A GRAPHICAL TOOL FOR TEST GENERATION FROM STATE MODELS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Qipeng Wu

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

December 2009

Fargo, North Dakota

North Dakota State University
Graduate School

Title

A GRAPHICAL TOOL FOR TEST

GENERATION FOR STATE MODELS

By

QIPENG WU

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Wu, Qipeng, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, December 2009. A Graphical Tool for Test Generation from State Models. Major Professor: Dr. Jun Kong.

This paper presented a graphical tool for generating test cases based on state models. The tool provides users with a user-friendly model editor to create their state model in a tree view structure. The tree based state model can then be saved to a disk in the form of an xml file and any existing model file can be loaded back into the tool. Two different traverse algorithms are explored by this tool, state based coverage and transition based coverage. The tool implements both algorithms and is capable of generating test paths based on different traversal algorithms. The tool also provides a code generation process that walks through these test paths and generates test cases in any one of the supported .NET based programming languages specified by the user. Lastly, the tool can generate a Visual Studio compatible model file based on the same state model created by the user. This model serves as a good visual representation of the state model created by the user in the model editor. The same state model is represented in three different forms, tree based state model in model editor, xml based state model in an xml file and graphical based state model in Visual Studio. An example is used to demonstrate the usage of this tool and the algorithms used behind the scene.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

1. INTRODUCTION

Testing is becoming more and more important in the modern software development process. As object oriented design is becoming the mainstream design mechanism, UML diagrams are used in different phases throughout the development process [1]. One of the UML models, the state model, is not only used during design, but also used in testing. In a popular software development methodology, test driven development, the state model plays a vital role during the entire life cycle of the software construction [2]. Subsystems\components are derived from the state model during design and the same model is used during testing for generating test cases.

1.1. Problem Definition

A huge amount of development time and effort is spent on documenting these state models. Sometimes these models are only used heavily during the design phase of the development process and neglected during other development phases. Even when state models are used throughout the development process, it is not an integrated experience. State models are created during the design phase using UML modeling tools. The design of classes is derived from the state model and code representing those classes and operations among classes are written by hand. During testing, test engineers come up with test cases by analyzing state models. Test cases are then hand written based on identified test scenarios. Due to resource constraints, not all possible paths

through the transition tree are covered. Only those paths considered top priority will be covered by test cases. There are model based test generation tools that do test generation by traversing state models. However, users will need to reformat their graphical state models into a program-friendly format, e.g. an xml document or a text file [3]. This not only costs the extra effort of making a new format of the same state model, but also incurs dual maintenance cost throughout the project's lifetime and in the products' after life.

Another problem with graphical state models, or with any other UI modeling tool, is that it is normally easy to use when few states are involved. However, when the number of states gets larger, it is a tedious process dragging and connecting numerous states in a single state model. Studies have indicated that humans are only capable of tracking up to 7 items simultaneously. Having to keep track of huge number of states is definitely an error-prone process.

Software is becoming more and more involved with our daily work and personal lives. We rely on software to tell us what the weather is going to be like. We rely on software to manage the transportation system that takes us to and from work. We rely on software to inform us what is happening near us and around the world. We rely on software to manage our personal, professional and financial lives. Our lives are run on top of software. As more and more software are developed each day by different software companies, the rate of software growth has not slowed down for decades. People always want the latest, the fastest and the newest software in their hands.

Companies spend huge amount of resources developing software to meet the needs of the ever growing fast paced model life style. Due to time and other resource constraints, large amounts of software used by everyday people are not well tested. A lot of development teams do not have a properly implemented development life cycle. Code quality maintenance is a big issue for development teams to manage [4]. A lot of source code repositories are un-guarded. Source codes are checked in at will when the code author thinks he/she is done implementing the code. Some better practices enforce peer code reviews before code submissions. While peer code review is an effective way of catching mistakes and bugs in the code, it is not a reliable way of enforcing source code integrity and correctness. Lots of time is spent fixing bugs introduced by these unguarded source code submissions. It might seem fast to check in source code at will and patch things up later, eventually, the bugs introduced would come back and bite the development team. More time would be wasted identifying and fixing the unintentional bug. If every source code submission had to run through a set of simple tests before going into the source code repository, we would have caught the bug in the first place and eliminated the need of coming back to identify and fix the bug. It is hard to strike a perfect balance between time spent developing the software and time spent testing it. Existing studies already indicate that model based testing improves bug detection process and actually reduce testing cost [5].

1.2. Tool Overview

This tool is intended to solve the problems listed above by introducing a new way of creating state models, exploring automated test generation from state models and experimenting the use of unified state model throughout the development process.

Phase one is the construction of the state model. Pictures are worth a thousand words. Graphical state models are very common and useful, because they are visually appealing and very easy to read and understand. However, as mentioned in the problem statement, it is always cumbersome to construct these state models graphically by using standard UML state model paradigms manually, because the sheer number of states and transitions one needs to deal with. Layout out huge number of states and event huger set of intermingled transitions on a piece of paper in a visible and organized fashion is no easy task.

Some resolves to construct non-graphical based state model by representing states and transitions with nodes in a XML document. It avoids having to deal with arrangement of graphical states and transitions, but it also loses the appealing visual elements possessed by its graphical counterpart. Not to mention the fact that it is virtually impossible for a normal human being to make any sense out of the page long XML document.

The above problem is solved by introducing the tree view based state model construction tool. Users of the tool start with a single root node, States, in their model.

As shown in Figure 1, a collection of child nodes can be easily added to the root node, which represents all states in a given state model. Transitions can be easily added to each state by adding child nodes under the Transition node under each state node. Properties on a transition can be set to indicate the starting and ending state of a transition. Constraints can also be specified on each transition. Details of how to constructing a state model will be covered in the Scenario and UI Step through section.

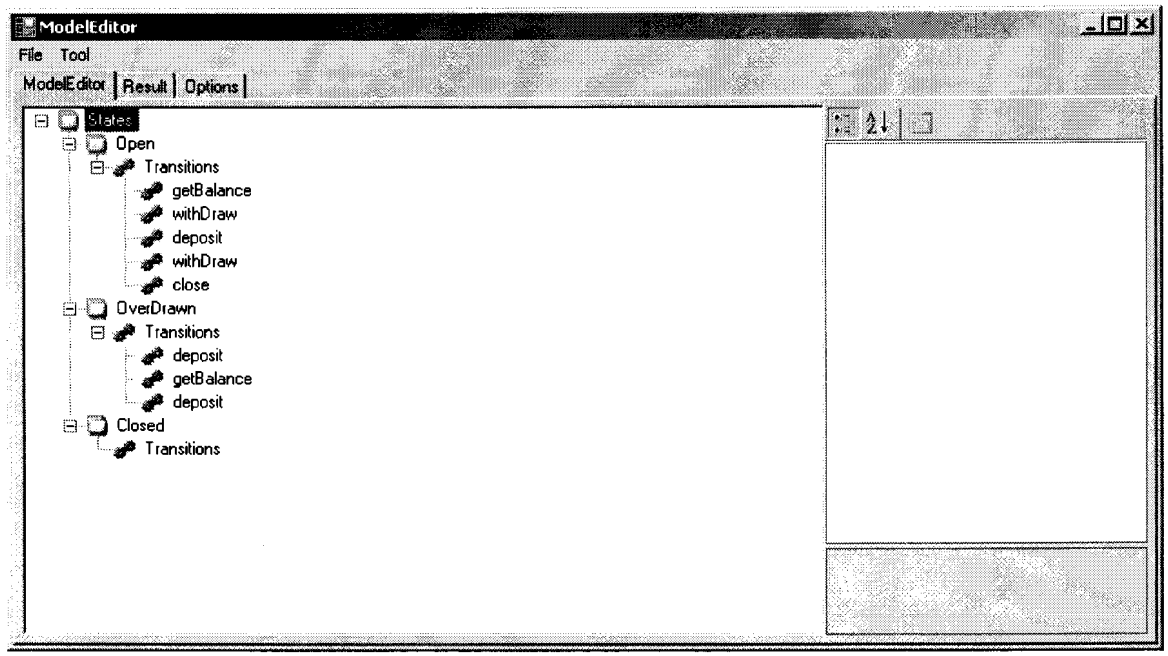

Figure 1. Construct a state model

When a state model is created in the model editor, it can be saved to the hard disk. The tools will traverse the state model and convert the structure to an XML document. The XML document can be saved to the hard disk and loaded back into the tool. Figure 2 shows an example of an XML representation of a state model.

```
<StateModel>
 <Open>
  <getBalance StartState="Open" EndState="Open" Constraint="" />
  <withDraw StartState="Open" EndState="Open" Constraint="b-amt>=0" />
  <deposit StartState="Open" EndState="Open" Constraint="amt>=0" />
  <withDraw StartState="Open" EndState="OverDrawn" Constraint="b-amt<0" />
  <close StartState="Open" EndState="Closed" Constraint="" />
 </Open>
 <OverDrawn>
  <deposit StartState="OverDrawn" EndState="Open" Constraint="b+amt>=0" />
  <getBalance StartState="OverDrawn" EndState="OverDrawn" Constraint="" />
  <deposit StartState="OverDrawn" EndState="OverDrawn" Constraint="b+amt<0"
/>
 </OverDrawn>
 <Closed />
</StateModel>
```

Figure 2. XML representation of a state model

After the state model is constructed, this tool can generate a graphical state model based on the tree view based state model. This way we achieved the simplicity of constructing non-graphical state models, avoided the clutterness of constructing graphical state models by hand and harnessed the benefits we get from visual representation. Result graphical model is shown in Figure 3.

The state model is a very good starting point to the entire development process. Not only we can base our design on the state model, we can also use the same model for testing. This is the perfect way of aligning the design process and the test process to identify good testing scenarios, which leads to good coverage on source code and higher quality software. That leads to the second problem.

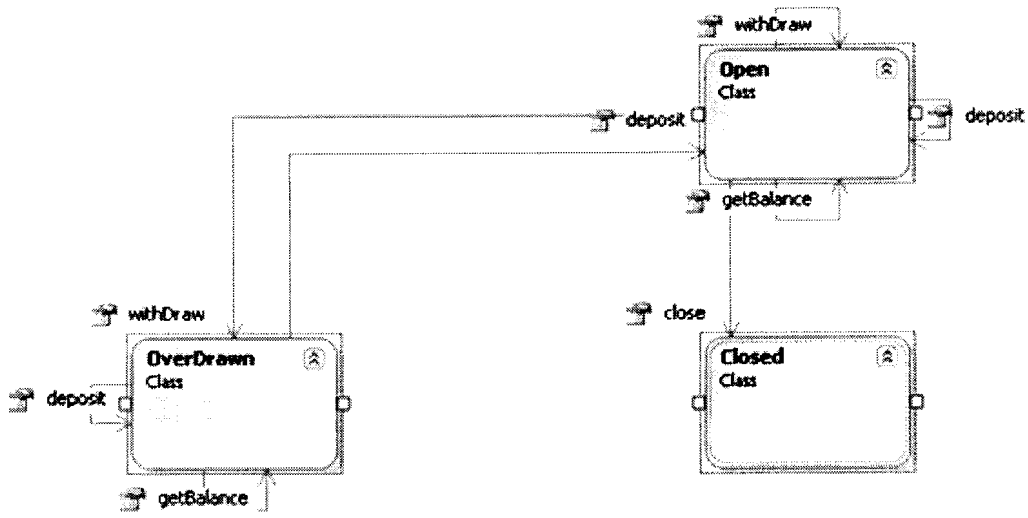

Figure 3. A graphical state model

By using the state model generated from phase one, the tool can generate test cases based on two different test coverage criteria, state based coverage and transition based coverage. State based coverage is a test generation algorithm in which test paths are generated in such fashion that all states in the state model are traversed at least once. Transition based coverage is a test generation algorithm in which test paths are generated in such fashion that all transitions in the state model are traversed at least once. These different test generation algorithms will be covered in more detail in the algorithm section.

In general, the number of transitions in a state model is larger than the number of states in a state model. Therefore, state based coverage provides a relatively smaller number of test cases that offers a good coverage for the source code. Since this small suite of test cases is quick and easy to execute, they can be used as a good suite of check in tests emplaced as part of a gated check in process. A developer would runs these

check in tests before their code submission to make sure the code they are checking in does not break major test scenarios. These tests provide good coverage for major functionalities while remain relatively small in size, quick and easy to execute. If a bug is introduced in the new source code that breaks an existing test case, it is relatively easier to debug because the size of the test suite is smaller. It also provides more incentives for developers to use before checking in for they are relatively less time consuming to run.

Transition based coverage provides a relatively larger number of test cases that offers more complete coverage for the source code. Since it is relatively bigger suite of test cases and it may take longer to run, this suite of test cases can be used as a good suite of regression tests that runs daily to identify possible bugs that may have been checked in. Although new source code passed through gated check in tests, it does not mean that the new source code is bug free. Daily ran regression tests provide more extensive test coverage [6]. Bugs not caught by the gated check in test could be caught by running the larger suite of test cases generated by transition based algorithm. As the result, we can identify bugs early in the development cycle. It is hard to debug a problem if there are a large number of bugs existing in the software. By identifying bugs early, we not only make it easier to debug problem, but also eliminate the possibility of duplicated dev effort. Because, sometimes, one bug can cause different issues to surface which not only blocks the owner of the bug, but also blocks other developers working on dependent or related components. This suite of generated test

cases can serve as a good base for regression tests which could be further extended by manually adding additional test cases. It is a very cost effective way to develop software and leads to better quality software.

1.3. Organization of the Paper

The remainder of this paper is organized as follows. Chapter 2 provides background and related work on model based testing and automated test generation. Chapter 3 introduces the algorithms used by the tool to traverse state models, some background information on visual studio integration with the tool. Chapter 4 covers the overall architect of the tool. Chapter 5 presents design details. Chapter 6 presents use cases and user scenarios of the tool. Chapter 7 explains UI components of the tool. Chapter 8 evaluates the tool by walking through an example. Chapter 9 summarizes what this paper has covered and discusses some open issues for future work.

2. BACKGROUND AND RELATED WORK

2.1. Model Based Test Generation

"A classical estimate relates up to 50% of the overall development cost to testing. Although this is likely to also include debugging activities, testing does and will continue to be one of the prevalent methods in quality assurance of software systems" [7]. Since the object oriented development has become the predominant methodology for software development, more and more attention has been paid to the model based testing.

The software development starts with gathering requirements. Domain experts meet with software program managers so that program managers can learn more about the specific domain and are able to define the problem the software is supposed to solve. A set of requirements is written by the program manager to describe what exactly the software should behave to solve the domain problem. These set of requirements are written in human readable language so that both customers and software development teams can read and understand. Customers and software development teams then come to an agreement on these set of requirements. However, since requirement specifications are written in human readable language, they have the tendency to be incomplete, ambiguous and sometimes contradictory. Most of the development teams have hand crafted tests based off of these requirement specifications. Inevitably, these tests could be incomplete and not 100% reflection of

what exactly the customers really required. State models are constructed during requirement/design phase to help clarify requirements.

Since state models are already widely used in the design phase, it is only logical to reuse these set of state models for testing implementation based on these designs. The model based testing makes scenarios under test more explicit, since system state models accurately describe different states the object can be in and the transitions among these states.

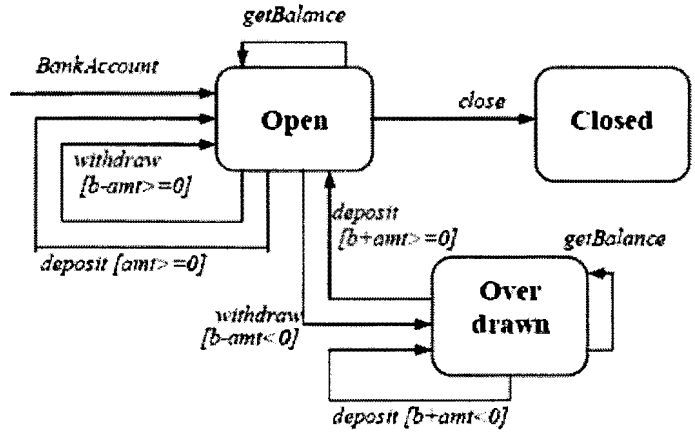

Figure 4. Bank account state model

As shown in Figure 4, each state in the above state model accurately describes the state the object is in. For example, the state Open in the BankAccount state model in figure below. It indicates that when a BankAccount object is initialized and its balance is greater than 0, its state is open.

Each transition consists of four parts of information. Take the transition from the state Open to the state Overdrawn for example.

1. The start state of the transition is Open.

2. The end state of the transition is Overdrawn.
3. The event that triggers the transition is withdraw, represented by the arrow going from Open to Closed.
4. The constraint on the event is b – amt < 0, which indicates that the BankAccount object will only transit from the state Open to the state Overdrawn when the balance of the BankAccount object is smaller than the amount withdrawn from the account.

Not only do state models provide us with a rigorous way for defining problem domains and designing software, they also provide us with a good foundation for the automated test generation. A test path is a collection of transitions that starts with object creation [8]. With the invocation of a given event, if the constraint of the event is satisfied, the state of the object transforms from the start state to the end state, therefore, completes the transition. Collection of test paths can be generated by traversing through state models. Based on different traversing logic, different collections of test paths can be generated. The collection of test paths can then be fed to a code generator. The code generator can iterate through the test paths and generate test cases. More advanced techniques can transfer state models into Constraint Logic Programming language and generate test cases with test input parameters [9].

2.2. Related Work

Using formal methods to model system has long been researched and utilized since the dawn of software industry. Using formal specifications to define system behavior eliminates ambiguity and helps reducing the possibility of errors during the software development process [9]. One kind of the most commonly used model languages is finite state based language, in which states are defined from a finite set of values. Several testing techniques come into existence based on finite state machines, such as transition tour, modified T method, W-method, Wp-method, etc. These techniques construct transition tree, traverse the transition tree based on different traversing logic, and verifies each test path with predefined set of conditions. Many state-based test generation methods also use a state model to represent the system and then test whether or not the implementation and design models conform to each other [8]. El-Fakih et al. have recently adapted four of the well-known methods (W, Wp, UIOv, and HIS) for generating tests that would test only the modified parts of an evolving specification [12]. Unfortunately, none of these techniques is tailored towards the testing of object oriented systems.

State models are widely used in the design of object oriented systems. They provide us with an accurate way of defining component behaviors. Test generation techniques based on state models are also been researched and developed. Xu et al. [8] explored the model based test generation based on four different criteria, the state

coverage, the transition coverage, and the basic and extended round-trip coverage. Their tool is also capable of providing users with the ability to specify input parameters for each transition and reuse these parameters when state model is modified. Hong et al. [13] provide a way to derive extended state machines from state charts to devise test criteria based on control and data flow analysis. Offutt et al. [14] provide definitions for such test criteria as all transitions, all transition pairs, and full-predicate for guard conditions. While topics covered in this paper is based on existing modeling and test generation work, this paper concentrates on introduction of the tree based model editing tool, the algorithms used in traversing the state model and the capability of representing the same state model in three different formats, the tree based structure, XML and the graphical state model in Visual Studio.

3. ALGORITHMS

3.1. Test Generation Algorithms

As introduced in the tool overview section, the tool can generate test cases based on two different coverage criteria, the state based coverage and the transition based coverage. These two coverage criteria are state model based traversal algorithms. We start with the initial state in a state model, traverse through states in the state model based on different algorithms and generate a collection of test paths. These test paths are then used to generate test cases in the user preferred programming language.

3.1.1. State Based Traversal

```
FOR each state in the list of states
    IF the state is not marked as visited
        call Traverse (newTransition, state)
    ENDIF
ENDFOR
Function Traverse (transition, targetState)
    Mark targetState as visited
    Add transition/targetState pair to the test path
    FOR each transition, t, that goes out of the targetState
        IF the ending state of t is not marked visited
            Call Traverse(t, endingState)
        ENDIF
    ENDFOR
    IF all transition from targetState leads to visited ending states OR there is no
transition coming out of the targetState
        Add the test path to the collection of test paths
    ENDIF
    Remove the last transition/state pair added to the test path
END Function Traverse
```

Figure 5. State based traversal

The traverse algorithm shown in Figure 5 can be explained by examining the simple state model shown in Figure 6. It consists of 3 states, named A, B and C; and four transitions, named 1, 2, 3 and 4.

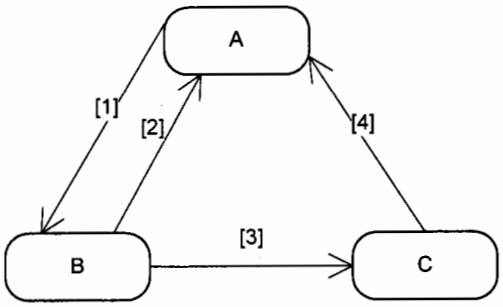

Figure 6. A simple state model

Here's how the traversal works in steps.

Traverse (x, A)

Mark A as visited and add (x, A) to test path, [(x, A)]

Traverse (1, B)

Mark B as visited and add (1, B) to test path, [(x, A), (1, B)]

Since transition 2 leads to state A and A is marked as visited, skip transition 2

Traverse (3, C)

Mark C as visited and add (3, C) to test path, [(x, A), (1, B), (3, C)]

Since transition 4 leads to state A and A is marked as visited, skip transition 4

Since all transition from State C leads to visited states, save test path to collection

Remove (3, C) from test path, [(x, A), (1, B)]

Done Traverse (3, C)

Remove (1, B) from test path, [(x, A)]

Done Traverse (1, B)

Remove (1, B) from test path, []

Done Traverse (x, A)

After the traversal, only 1 test path is generated for the simple state model above, i.e., [(x, A), (1, B), (3, C)]. The most important thing to take notice here is that all states in the state model, A, B and C, are covered in this test path. However, not all transitions are covered. Only transition 1 and 3 are included in the test path. We created a path of states and reached each state by recording the transition/state pair. Now, we can generate test cases based on the result test path collection. In this simple case, we have only 1 test path in the collection. As shown in Figure 7, there is only 1 test case generated.

```
public static void TestCase1()
{
    TestClass testObject = new TestClass();
    // TODO: verify if the current state is A
    //
    testObject.Transition1();
    // TODO: verify if the current state is B
    //
    test.Transition3();
    // TODO: verify if the current state is C
}
```

Figure 7. Generated test case - state based traversal

3.1.2. Transition Based Traversal

```
FOR each state in the list of states
    IF the state is not marked as visited
        call Traverse (newTransition, state)
    ENDIF
ENDFOR
Function Traverse (transition, targetState)
    Mark targetState as visited
    Add transition/targetState pair to the test path
    FOR each transition, t, that goes out of the targetState
        IF the ending state of t is marked as visited OR there is no transition coming
out of targetState
            Add t/endingState to the test path
            Add the test path to the collection of test paths
            Remove t/endingState from the test path
        ENDIF
        ELSE
            Call Traverse(t, endingState)
        ENDELSE
    ENDFOR
    Remove the last transition/state pair added to the test path
END Function Traverse
```

Figure 8. Transition based traversal

The traverse algorithm shown in Figure 8 can be explained by examining the same state model used for state based traversal, as shown in Figure 9.

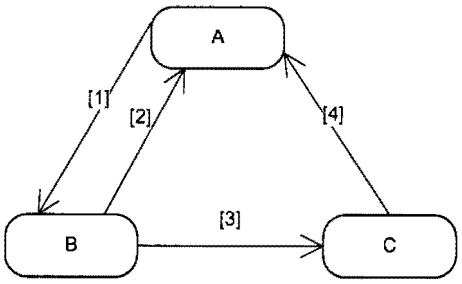

Figure 9. A simple state model

Here's how the traversal works in steps.

Traverse (x, A)

Mark A as visited and add (x, A) to test path, [(x, A)]

Traverse (1, B)

Mark B as visited and add (1, B) to test path, [(x, A), (1, B)]

Since transition 2 leads to state A and A is marked as visited, add (2, A) to test path, [(x, A), (1, B), (2, A)]

Add test path [(x, A), (1, B), (2, A)] to collection

Remove (2, A) from test path, result test path [(x, A), (1, B)]

Traverse (3, C)

Mark C as visited and add (3, C) to the test path, [(x, A), (1, B), (3, C)]

Since transition 4 leads to state A and A is marked as visited, add (4, A) to test path, [(x, A), (1, B), (3, C), (4, A)]

Add test path [(x, A), (1, B), (3, C), (4, A)] to collection

Remove (4, A) from test path, result test path, [(x, A), (1, B), (3, C)]

Remove (3, C) from test path, [(x, A), (1, B)]

Done Traverse (3, C)

Remove (1, B) from test path, [(x, A)]

Done Traverse (1, B)

Remove (x, A) from test path, []

Done Traverse (x, A)

After the traversal, 2 test paths are generated for the simple state model above, i.e., [(x, A), (1, B), (2, A)] and [(x, A), (1, B), (3, C), (4, A)]. The most important thing to

take notice here is that not only all states in the state model, A, B and C are covered by the collection of test paths, all transitions in the state model, 1, 2, 3 and 4, are also covered by the collection of test paths. Now, we can generate test cases based on the result test path collection. In this case, we have 2 test paths in the collection. As shown in Figure 10, there are two test cases generated.

Figure 10. Generated test cases - transition based traversal

3.2. Basics on Visual Studio Diagram

The diagram tool from Visual Studio is very intelligent and powerful tool. It has the ability to automatically arrange notes once they are added to the model. The visual studio diagram tool is used for modeling class diagrams. Visual studio does not have a diagram tool for state models. It is a bit confusing in the beginning that class diagrams

are used to represent state models. The simple example shown in Figure 11 illustrates why and how this diagram tool is used.

Figure 11. Visual Studio class diagram

At first glance, one may easily mistake the above class diagram for a state model. That is exactly why the class diagram in VS is used to represent the graphical state model. State0 and State1 in the above diagram are actually C# classes. Transition0 is a property on State0. The return type of Transition0 is State1. That's how VS represent the association between two classes. A link from State0 to State1 represents a property on the State0 whose return type is State1. Figure 12 shows the code behind the above class diagram.

```
class State0
{
    public State1 Transition0 {}
}
class State1
{
}
```

Figure 12. C# code behind the diagram

4. SYSTEM ARCHITECTURE

4.1. Class Diagrams

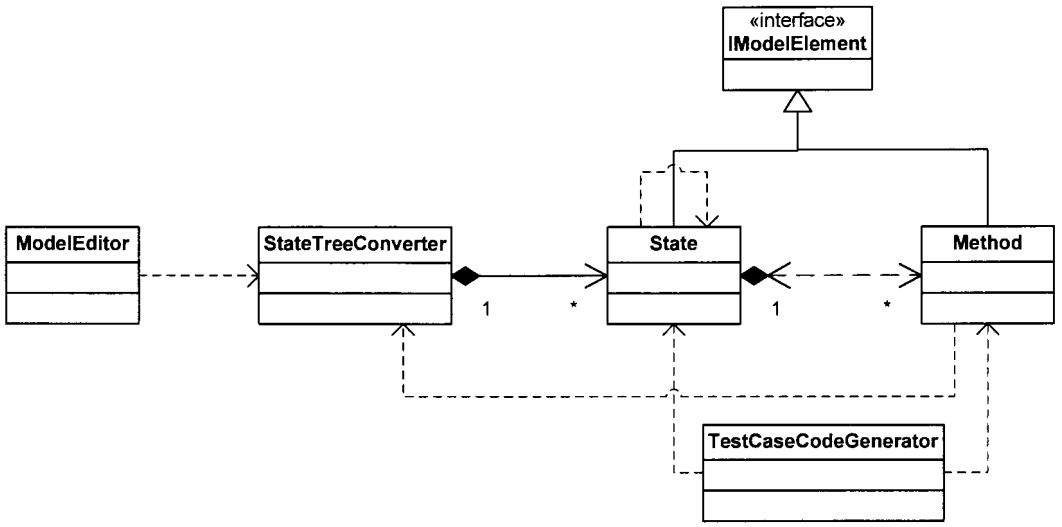

Figure 13. System architecture

Figure 13 shows the class diagram of the test generation tool.

ModelEditor

This class contains the UI element of the tool. It is responsible for creating the state model. It also contains the state model diagram generating logic. State models can be created and modified in the tree view structure in the model editor. When selecting a node in the tree, the property pane on the right side of the model editor displays related property information to the selected node. User can than modify properties of the selected node, state or transition, through the property window. This development experience should be very familiar with users of Visual Studio.

StateTreeConverter

This class is the container for all user created states. It also has the logic to convert the state model into transition trees.

IModelElement

This interface provides some common properties that are shared between the State class and the Method class.

State

This class is an abstraction of a state defined in the state model. It has a transition table in the form of a collection of key value pairs. Each key value pair consists of a transition and the ending state of that transition.

Method

This class is an abstraction of a transition in the state model. It has two important properties, the start state and the end state. Start state is automatically defaulted by the tool since transitions are created under a state. The start state of a given transition would be the state under which the transition is created. When user set the value of the end state on a given transition, the end state is looked up in the collection of states and the transition entry on the start state is updated with the corresponding end state.

TestCaseCodeGenerator

This class is responsible for iterating through the collection of test paths and generating test cases based on each test path. Since the tool uses .NET CodeDom APIs

for code generation, it is capable of generating test cases in different .NET based programming languages [10].

4.2. Sequence Diagrams

Generate State Model Diagram

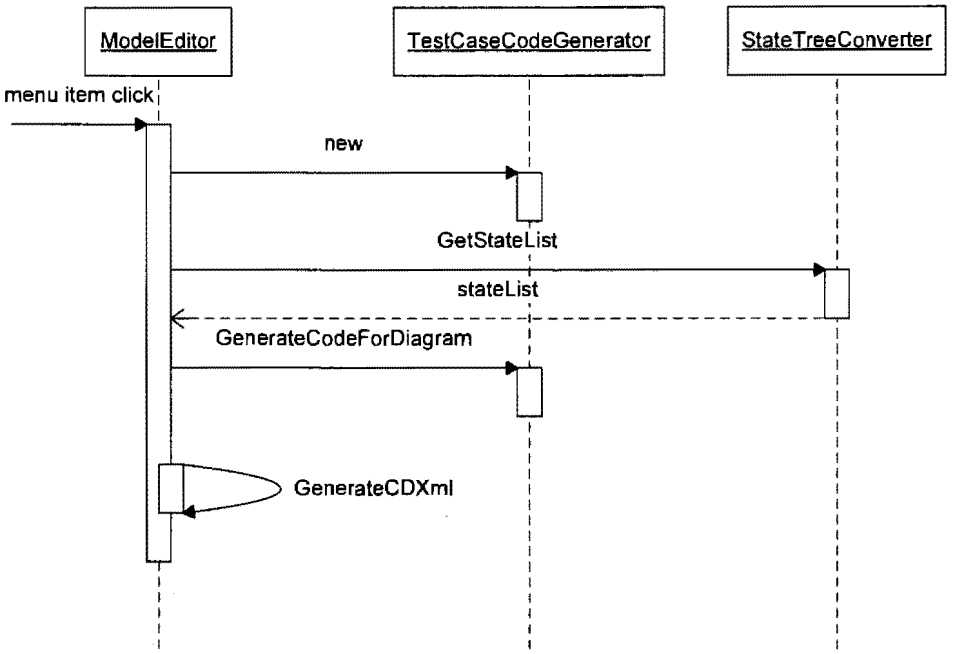

Figure 14. Sequence diagram - generate state model

Figure 14 shows the sequence diagram for generating the state model. After the state model is constructed in the model editor and the user clicks on the menu item "Generate state diagram", an instance of TestCaseCodeGenerator is created. The tree view based state model from the state editor is traversed and C# code files are generated by calling GenerateCodeForDiagram on TestCaseCodeGenerator. After C# code files are generated, the same stateList is traversed again by calling GenerateCDXml to generate

the VS compatible model xml file. By adding the result code file and the model file to a VS project, we can open the model file and view the generated graphical state model.

Generate Test Case Based on State Model

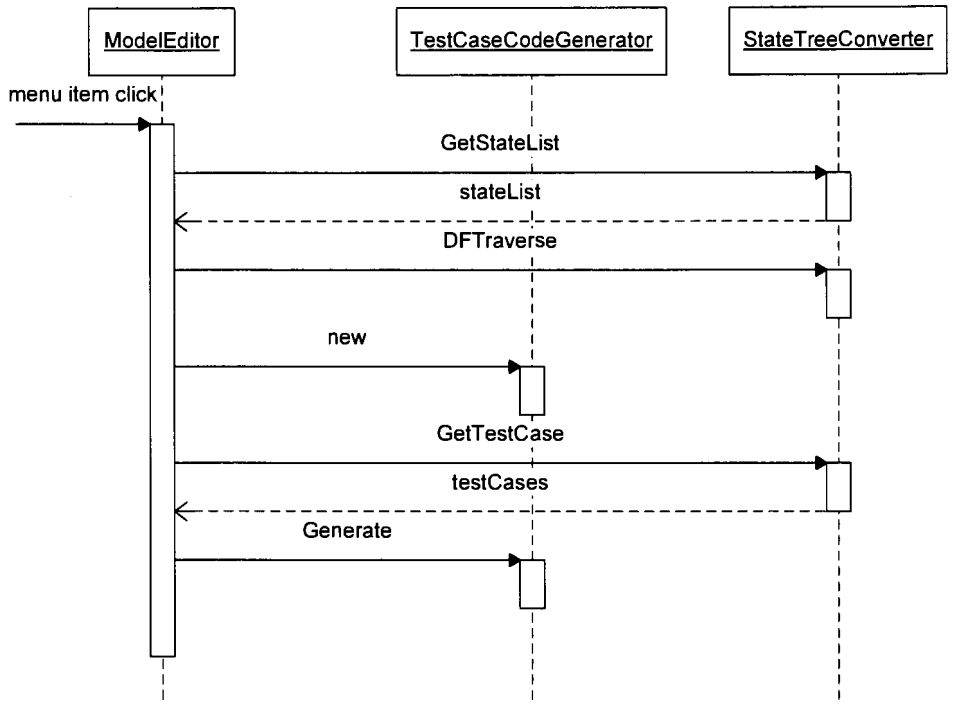

Figure 15. Sequence diagram - generate test case

Figure 15 shows the sequence diagram for generating test cases. After the state model is constructed in the model editor and the user clicks on the menu item "Generate test cases", the state model is traversed based on the coverage criteria chosen by the user. A collection of test paths are generated as the result of the traversal. An instance of TestCaseCodeGenerator is created. The collection of test paths is passed to TestCaseCodeGenerator::Generate method to generate code files that contains actually test cases based on the programming language user have chosen.

5. DESIGN DETAILS

5.1. ModelEditor

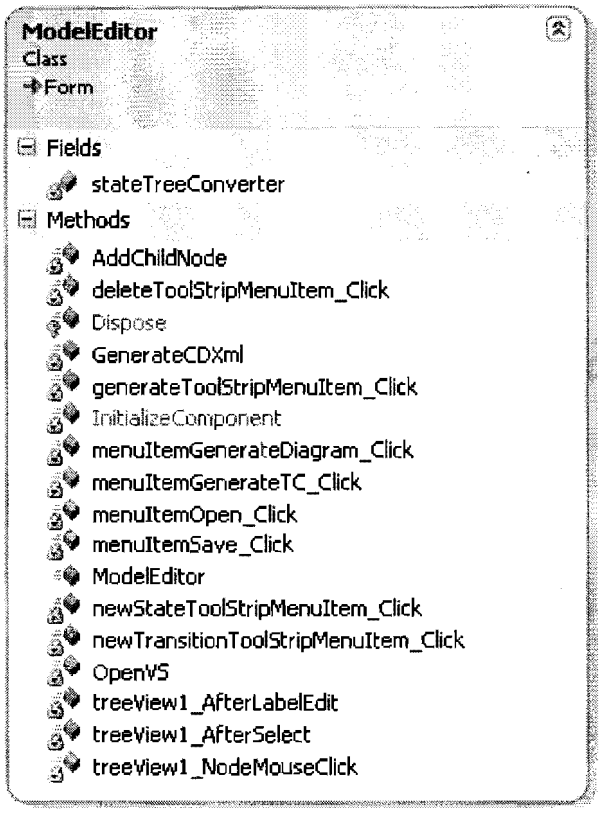

Figure 16. ModelEditor class diagram

Figure 16 shows the class diagram for the ModelEditor class.

Fields

---

stateTreeConverter – an instance of StateTreeConverter class that manages all user created states.

Methods

```
void AddChildNode(IModelElement newModelElement)
```

Description: Adds a child node to the current selected node and hook up the newly created child node to represent the new model element.

Parameters

newModelElement – the model element that represented by the newly created child node.

```
deleteToolStripMenuItem_Click(object sender, EventArgs e)
```

Description: this method handles the click event for clicking the "Delete" menu item on a selected node.

```
void GenerateCDXml()
```

Description: generate an xml file that has the format for displaying the state model in Microsoft Visual Studio.

```
void menuItemGenerateDiagram_Click(object sender, EventArgs e)
```

Description: this method handles the click event for generating the state model diagram. The outcome of this method is two generated files. A CSharp file is generated to contain the structures that support Microsoft Visual Studio display of the state diagram. An xml file is generated to contain the same information but in a different form required by Microsoft Visual Studio for displaying.

```
void menuItemGenerateTC_Click(object sender, EventArgs e)
```

Description: this method handles the click event for generating test cases. The outcome of this method is a code file containing the generated test cases and a test driver. The language used in the code depends on the option the user set in the "Options" tab.

```
menuItemOpen_Click(object sender, EventArgs e)
```

Description: this method handles the click event for the file open dialog.

```
void menuItemSave_Click(object sender, EventArgs e)
```

Description: this method handles the click event for the file save dialog.

```
public ModelEditor()
```

Description: this is the constructor of the ModelEditor class. It creates an instance of StateTreeConverter class and hook up some event handler methods.

```
void newStateToolStripMenuItem_Click(object sender, EventArgs e)
```

Description: this method handles the click event for creating a new state.

```
void newTransitionToolStripMenuItem_Click(object sender, EventArgs e)
```

Description: this method handles the click event for creating a new transition.

```
void treeView1_AfterLabelEdit(object sender, NodeLabelEditEventArgs e)
```

Description: this method handles the event when user edits the name of a node in the model editor.

```
void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
```

Description: this method handles the event when user selects a node in the model editor. It hooks into the property window so that the property window displays the content of the selected node.

```
void treeView1_NodeMouseClick(object sender, TreeNodeMouseClickEventArgs
e)
```

Description: this method handles the event when a node in the mode editor is clicked. It enables user to edit the name of the selected node.

5.2. IModelElement

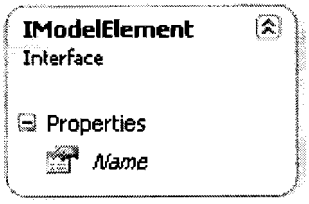

Figure 17. IModelElement

Figure 17 shows the class diagram for the IModelElement interface.

Property

Name – this property is shared by the State class and the Method class which are both model elements. This interface provides a common place to host similar properties shared by both model elements.

5.3. State

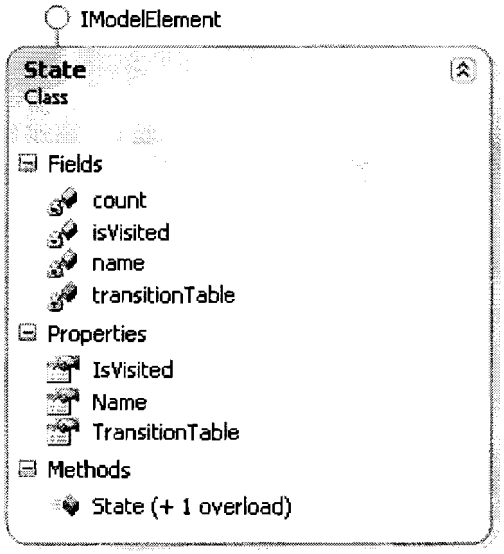

Figure 18. State class diagram

Figure 18 shows the class diagram for the State class.

Fields

count – this acts as the unique id for the states created.

isVisited – this field indicates whether this state has been visited by the state tree traverse algorithm.

name – the name of the current State.

transitionTable – a hash table that contains the transitions associated with the current State. Each transition contains information on the end State.

Properties

IsVisited – returns whether the current State is visited by the traverse algorithm.

Name – returns the name of the current State.

TransitionTable – returns the transitionTable.

Methods

```
public State()
```

Description: this is the public constructor for the State class.

5.4. Method

Figure 19 shows the class diagram for the Method class.

Fields

constraint – the constraint user can put on a Method.

count – this acts as the unique id for the Methods created.

end – this field indicates the end state of the current Method.

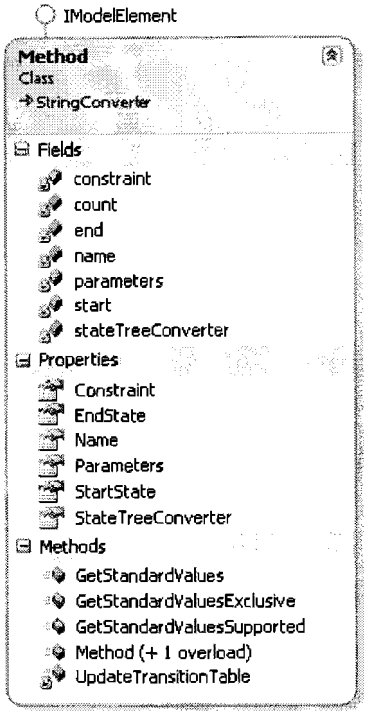

Figure 19. Method class diagram

name – the name of the current Method.

parameters – parameters taken by the current Method.

start – this field indicates the owner state of the current Method.

stateTreeConverter – the stateTreeConverter that contains all available States.

Properties

Constraint – returns the constraint of the current Method.

EndState – returns the name of the end State associated with the current Method.

Name – returns the name of the current Method.

Parameters – returns the parameters of the current Method.

StartState – returns the owner state of the current Method.

stateTreeConverter – return the stateTreeConverter that contains all available States.

Since it is static, there is only one instance across the entire application.

Methods

```
public Method()
```

Description: this is the public constructor for the Method class.

```
override bool GetStandardValuesSupported(ITypeDescriptorContext context)

override TypeConverter.StandardValuesCollection

GetStandardValues(ITypeDescriptorContext context)

override bool GetStandardValuesExclusive(ITypeDescriptorContext context)
```

Above three methods are implemented for the base class StringConverter. Method class dervies from the Microsoft .NET StringConverter class to support the pull down menu filling behavior observed in the property window [11].

```
void UpdateTransitionTable()
```

Description: this method updates the transition table of the owner state if the user changes the end state by changing the end state property in the property window.

5.5. StateTreeConverter

Figure 20 shows the class diagram for the StateTreeConverter class.

Fields

---

modelName – the name of the current model.

stateList – a list of all available States.

statesVisited – this keeps track of all the States that have been traversed by the traverse algorithm.

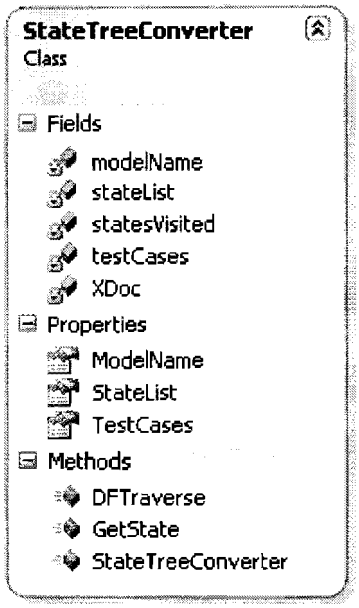

Figure 20. StateTreeConverter class diagram

testCases – the internal structure defined for the test case generation logic. Each test case is represented by a list of States and the transitions they are associated with.

xDoc – an xml representation of the States and their associated transitions tracked by the stateTreeConverter.

Properties

ModelName – returns the name of the current model.

StateList – returns a list of all available States.

TestCases – returns the internal structure representing test cases.

Methods

```
public StateTreeConverter()
```

Description: this is the public constructor of the StateTreeConverter class.

```
void DFTraverse(KeyValuePair<Method, State> visitingState)
```

Description: this method traverse the transition tree and adds series of States together to form test cases.

Parameters

visitingState – the state the traversing algorithm is currently visiting.

```
State GetState(string stateName)
```

Description: this method gets the corresponding State identified by the name passed in by the caller.

Parameters

stateName – the name of the state user is looking for.

Return

The state identified by the name passed in by the caller

5.6. TestCaseCodeGenerator

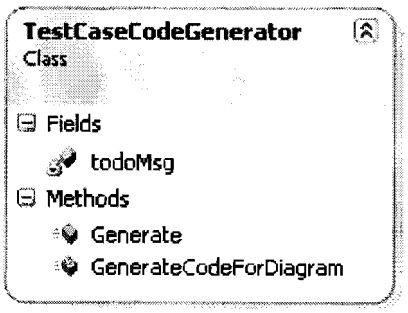

Figure 21. TestCaseGenerator class diagram

Figure 21 shows the class diagram for the TestCaseCodeGenerator class.

Fields

todoMsg – the TODO string used throughout the generated test case to remind tester to fill in the test data.

Methods

```
string Generate(string modelName, string typeName,
List<List<KeyValuePair<Method, State>>> stateGraph)
```

Description: this method generates the test cases based on the graph of states passed in.

Parameters

modelName – the name of the model

typeName - the type of language the code generation logic should use to generate the test case.

stateGraph – this is a list of KeyValuePairs of Method and State. Each second level list identifies a sequence of transition and state changes that indicate a test case.

Return

A string representing the full path of the file all the test cases are generated into.

```
string GenerateCodeForDiagram(string modelName, List<State> stateList)
```

Description: this method generates the CSharp code needed by the Microsoft Visual Studio to display the state diagram. Each state in the stateList is generated into a C# class. Each transition under that state is generated into a property on the generated class. The owner class of the property represents the start state of the transition and the return type of the property represents the end state of the transition.

Parameters

Modelname – the name of the model.

stateList – a list of all available States.

Return

A string representing the full path of the generated CSharp file.

6. USE CASES AND SCENARIOS

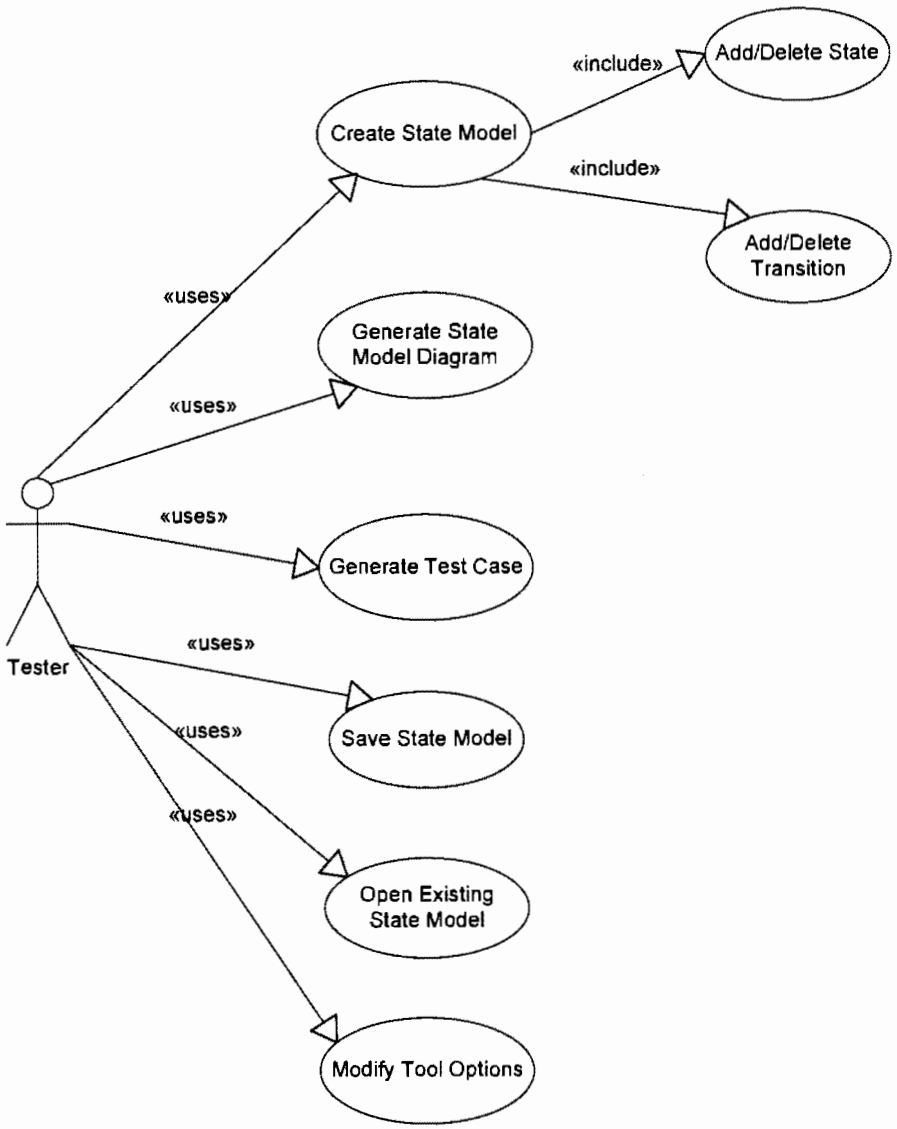

Figure 22. Use cases

Figure 22 shows the use cases diagram for this tool.

6.1. Actors

Tester is the direct user of the tool who uses this tool to model their state model, create state model diagrams, and generate test cases from state models and other related activities.

6.2. Scenarios and UI Step Through

Before we proceed to our first scenario, the tool is started by double click on the tool exe ModelEditor.exe. Figure 23 shows the UI of tool in its starting state.

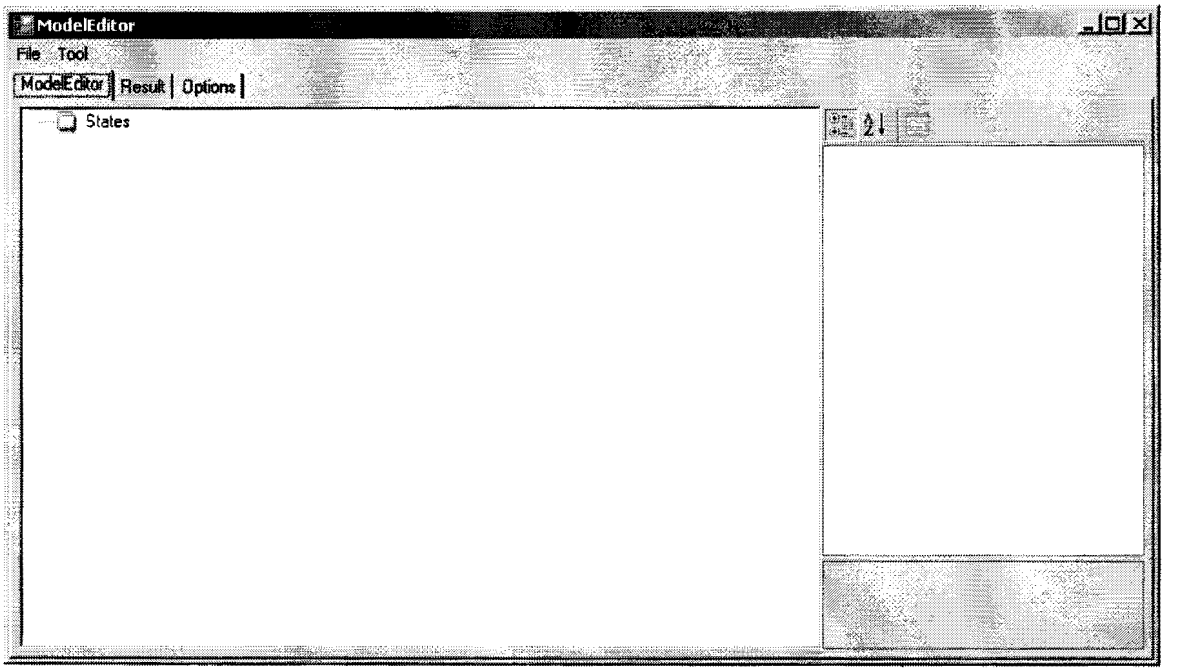

Figure 23. ModelEditor

Create State Model

This scenario consists of two sub scenarios, add/delete state and add/delete transition. In order to add a new state to the state model, the tester can select the node

in the main workspace labeled "States" and right click. A context menu should pop up and "New State" should appear as the only context menu item, as shown in Figure 24.

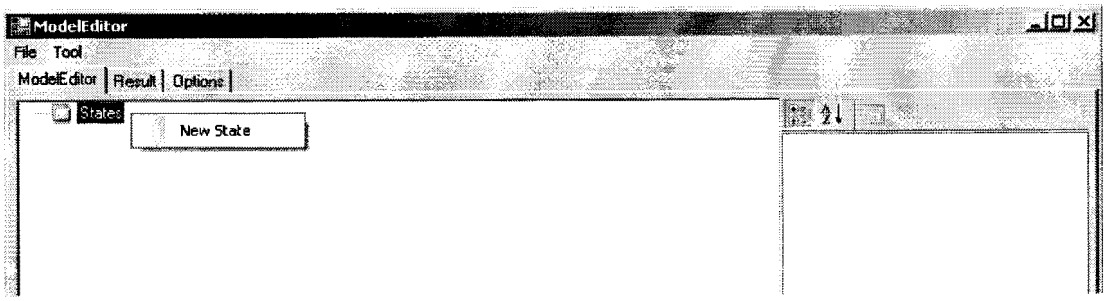

Figure 24. Create new state 1

The tester can click on "New State". A child node will be added to the "State" node. The name of the state is defaulted to "State", but tester can rename the node to a state name that makes sense in real life scenario. In this case, we'll name our new state "Open". Hit enter when finish and a new state is added. Notice the property window on the right side of the tool window is now displaying properties for the new state, as shown in Figure 25.

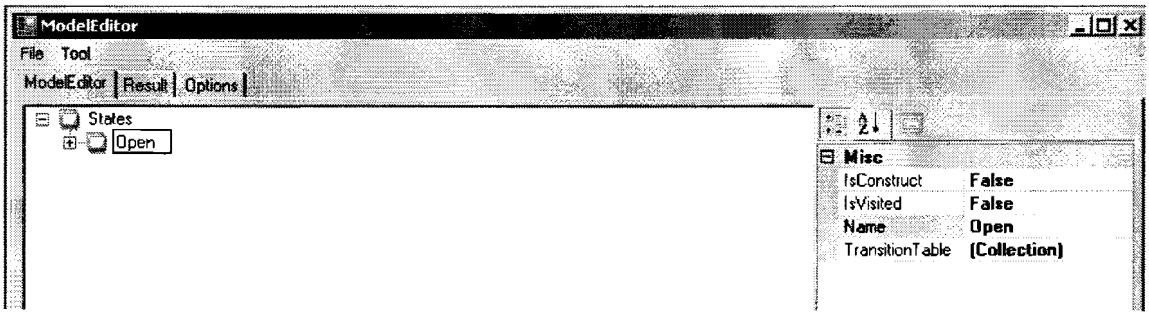

Figure 25. Create new state 2

To add a transition for a state, expand the state you want to add the transition to. A transition node will appear under the selected state node, as shown in Figure 26.

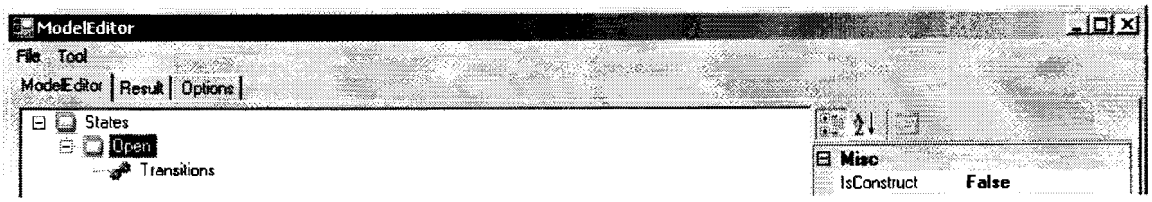

Figure 26. Add transition 1

The tester clicks on "Transitions" node and right click to bring up the context menu that has one context menu item reads "New Transition", as shown in Figure 27.

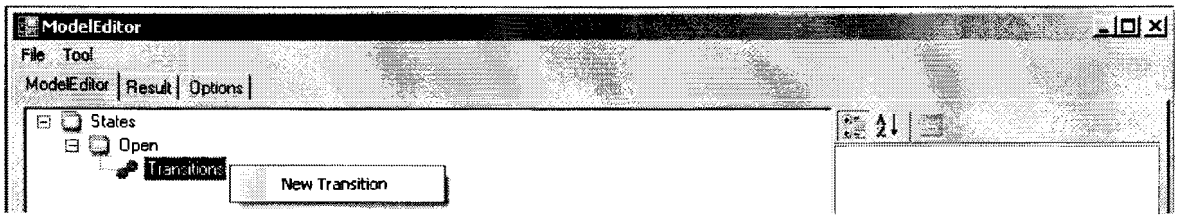

Figure 27. Add transition 2

The tester clicks on "New Transition" and a child node appears under "Transitions". The name of the child node is defaulted to be "Transition". The tester can rename the node to a transition name that makes sense in a real life scenario. In this case we name this transition "withdraw", as shown in Figure 28. Hit enter when finish and a new transition is added. Notice the property window is displaying properties of the newly created transition. Property named "StartState" should be set to the state to which the transition belongs.

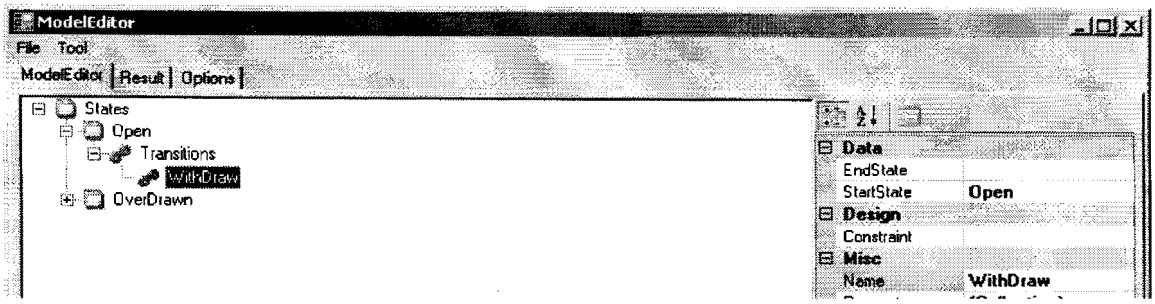

Figure 28. Add transition 3

As shown in Figure 29, one more state called "Overdrawn" can be added.

Figure 29. Add the Overdrawn state

Now, we can complete the transition we created under "Open" by setting the "EndState" property to "Overdrawn". In order to accomplish that, the tester can first click on the transition "withdraw". On the property window on the right side of the tool window, the tester can click into the "EndState" property. A drop down triangle appears on the right side of the control. Clicking on the drop down button, a list of the available states is displayed, as shown in Figure 30.

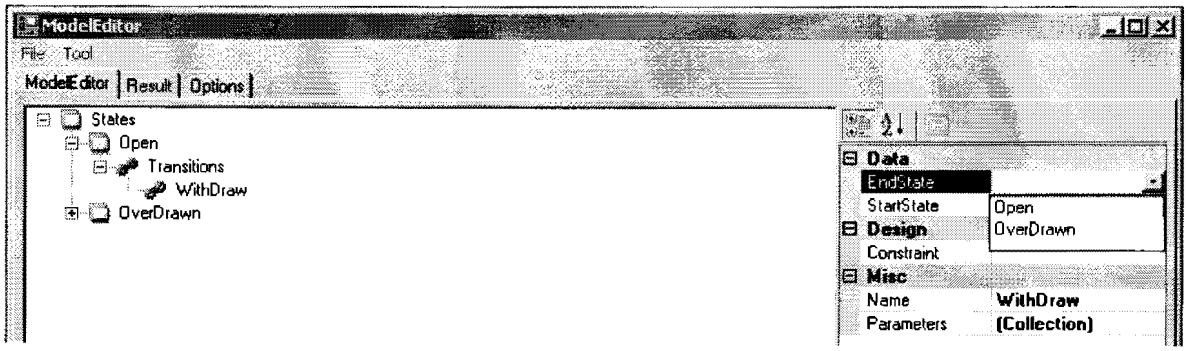

Figure 30. Display list of all states

The tester clicks on "Overdrawn" and insert "b-amt < 0 and b-amt>=-1000" under property "Constraint". Now, we completed adding two states and a transition connecting the two states, as shown in Figure 31.

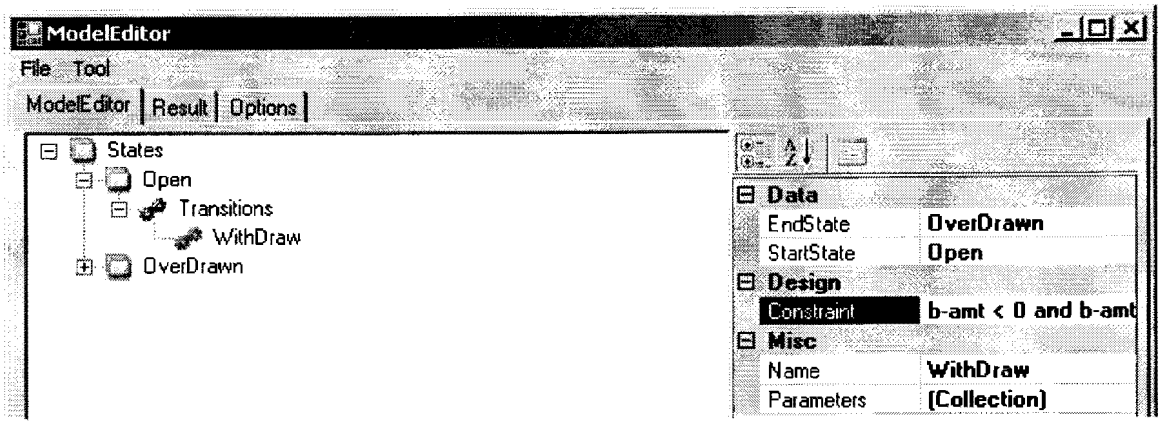

Figure 31. Two states and connection transition

The tester can delete any transition instance by right clicking on the target transition instance and a context menu with one menu item "Delete" will appear, as shown in Figure 32.

Figure 32. Delete transition 1

Clicking on "Delete" removes the selected transition, as shown in Figure 33.

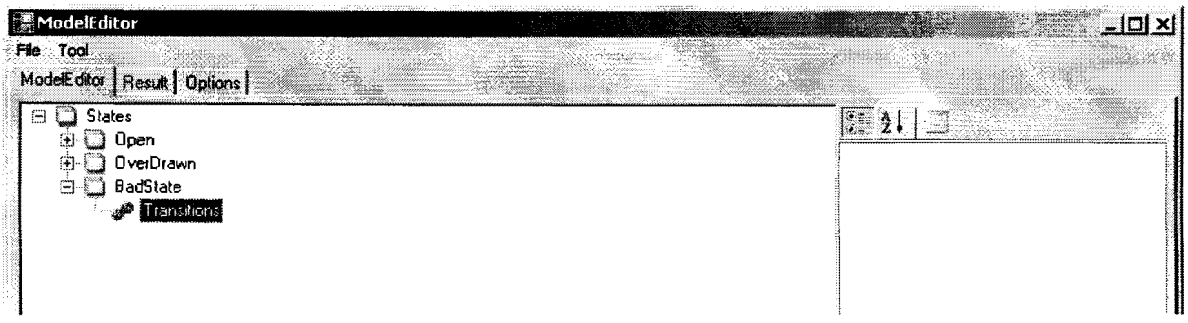

Figure 33. Delete transition 2

The tester can delete any state instance by right clicking on the target state instance and a context menu with one menu item "Delete" will appear, as shown in Figure 34.

Figure 34. Delete state 1

Clicking on "Delete" removes the selected state, as shown in Figure 35.

Figure 35. Delete state 2

Generate State Model Diagram

The tester can get a graphical representation of the state model using this tool. This is where this tool integrates with Microsoft Visual Studio 2008. Tester clicks on "Tool" from the menu bar and clicks on "Generate State Diagram", as shown in Figure 36.

Figure 36. Generate state diagram

As shown in Figure 37, the generated graphical state model can be opened in a VS project and rearranged to look like the following.

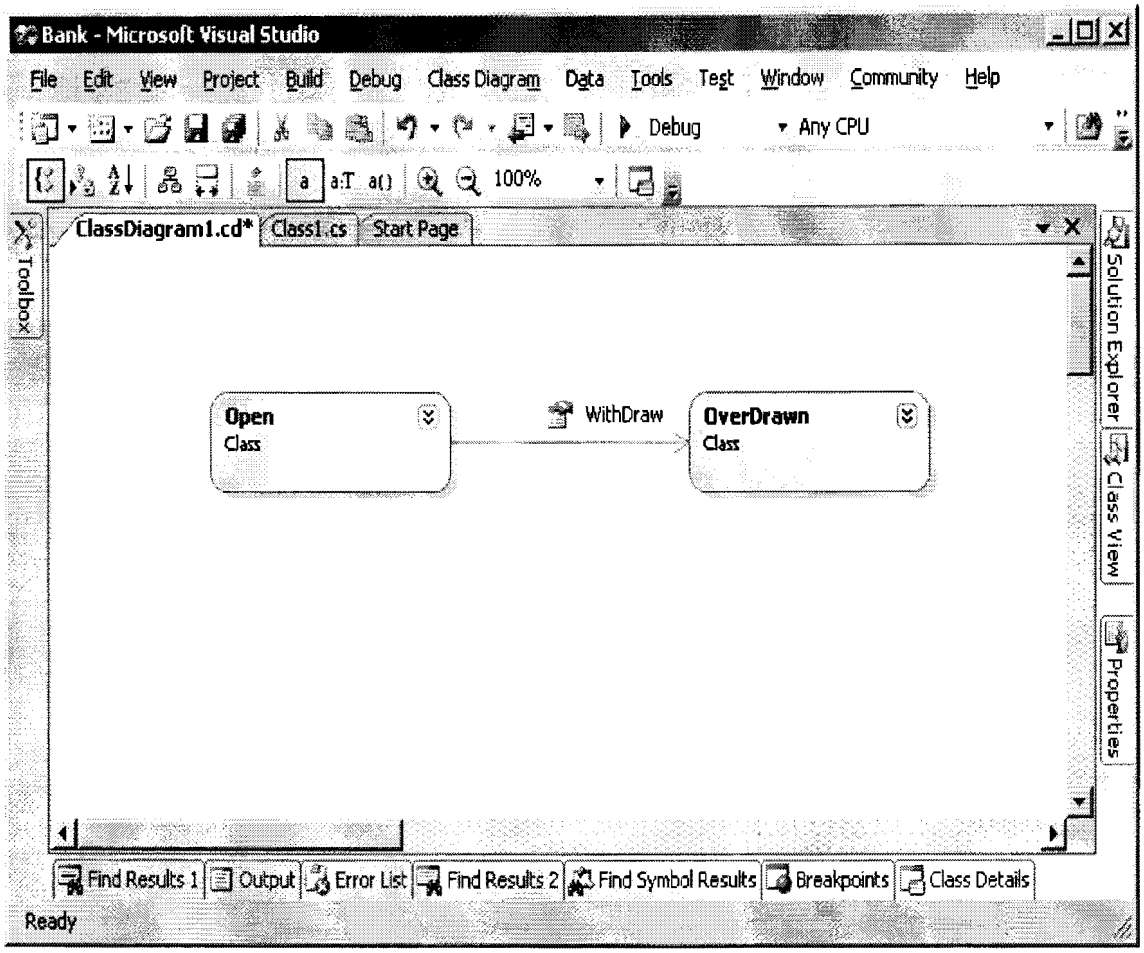

Figure 37. State diagram - Open, Overdrawn

Generate Test Case

The tester can generate test case based on the state model created in the tool by clicking on "Tool" on the menu bar, then clicking on "Generate Test Case". There are two different test generation algorithms to choose from, state coverage based generation or transition coverage based generation. The tester can pick one based on the purpose of the test suite to generate tests, as shown in Figure 38.

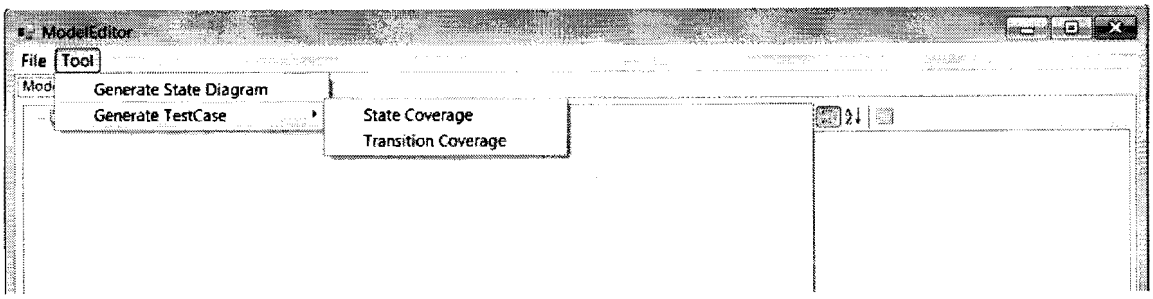

Figure 38. Genenerate test cases

To see the generated test cases, the tester can click on the "Result" tab. The generated test file containing test cases and a test driver is displayed, as shown in Figure 39.

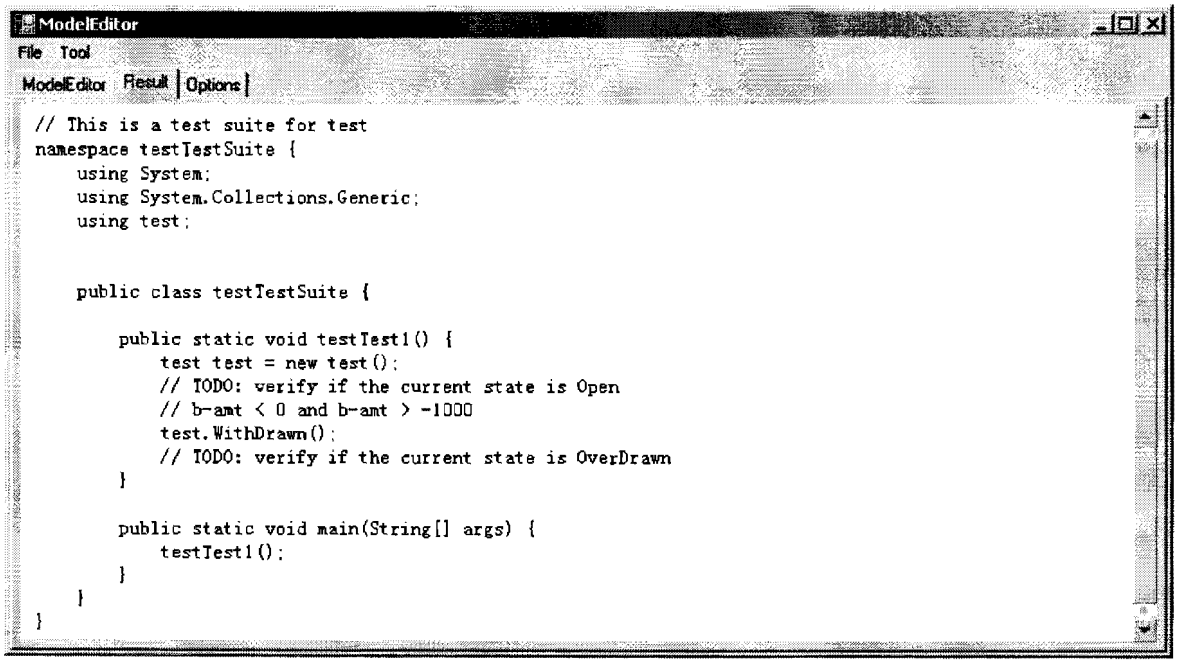

Figure 39. Generated test case

Save State Model

The tester can save the state model by go to "File->Save" on the menu bar, as shown in Figure 40.

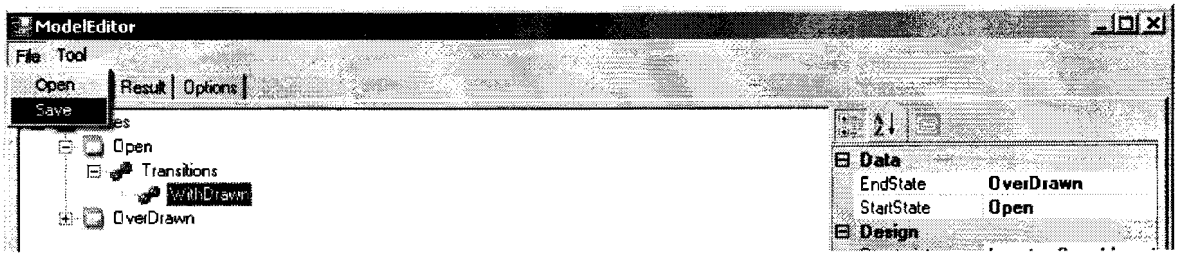

Figure 40. Save state model

The following is an example of a saved model xml file, as shown in Figure 41.

```
<StateModel>
  <Open>
    <getBalance StartState="Open" EndState="Open" Constraint="" />
    <withDraw StartState="Open" EndState="Open" Constraint="b-amt>=0" />
    <deposit StartState="Open" EndState="Open" Constraint="amt>=0" />
    <withDraw StartState="Open" EndState="OverDrawn" Constraint="b-amt<0" />
    <close StartState="Open" EndState="Closed" Constraint="" />
  </Open>
  <OverDrawn>
    <deposit StartState="OverDrawn" EndState="Open" Constraint="b+amt>=0" />
    <getBalance StartState="OverDrawn" EndState="OverDrawn" Constraint="" />
    <deposit StartState="OverDrawn" EndState="OverDrawn" Constraint="b+amt<0"
/>
  </OverDrawn>
  <Closed />
</StateModel>
```

Figure 41. Saved xml model file

Open Existing State Model

The tester can save the state model by go to "File->Open" on the menu bar, as shown in Figure 42.

Figure 42. Open state model

Modify Tool Options

The tester can modify tool options by navigating to the Options tab, as shown in Figure 43.

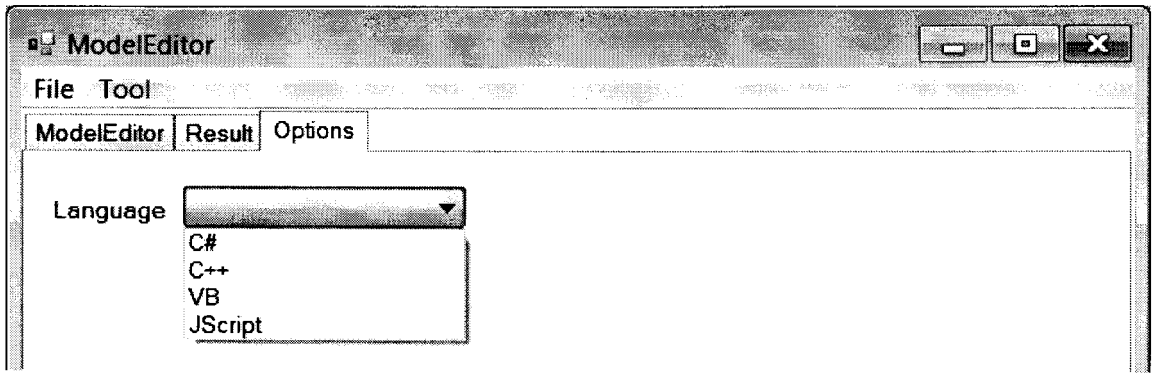

Figure 43. User options

7. UI OVERVIEW

This tool consists of two major UI components: the menu bar and the tab pages.

7.1. Menu Bar

File -> Open

Opens an existing state model file and load it into the tool.

File -> Save

Saves the current state model to the user specified location.

Tool->Generate State Diagram

Generates a state mode diagram and open it in Microsoft Visual Studio.

Tool->Generate Test Case->State Coverage

Generates test cases based on the state model constructed in the editor using state based coverage.

Tool->Generate Test Case->Transition Coverage

Generates test cases based on the state model constructed in the editor using transition based coverage.

7.2. Tab Pages

There are three tab pages in the tool, as shown in Figure 44.

ModelEditor Tab

This tab is the workspace for create state model. The property window on the rightmost side of the tab page display properties of the selected node in the workspace.

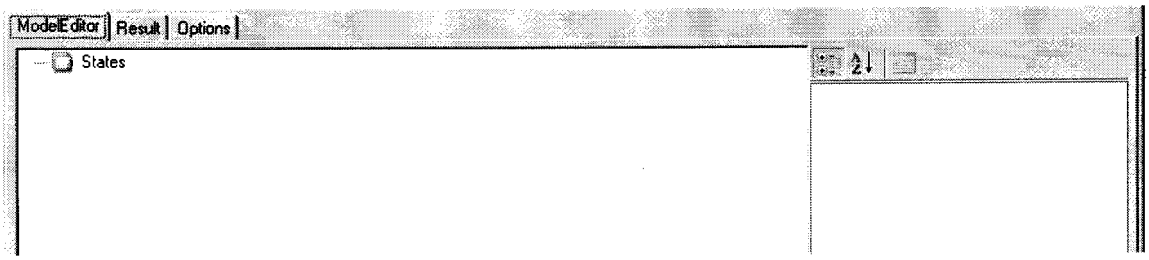

Figure 44. Workspace

Result Tab

Figure 45. Result tab

As shown in Figure 45, this tab page displays the test cases and the test driver generated by the test generation component.

Options Tab

Figure 46. Options tab

As shown in Figure 46, this tab page provides tool options for users to modify.

Language – this pull down menu contains all the possible languages the tool can use to generate test cases. The available languages include C#, C++, Visual Basic and JScript.

8. CASE STUDIES

8.1. The Bank Account State Model

This state model documents state changes for a given bank account. There are three valid states for a bank account, Open, Overdrawn and Closed, as shown in Figure 47.

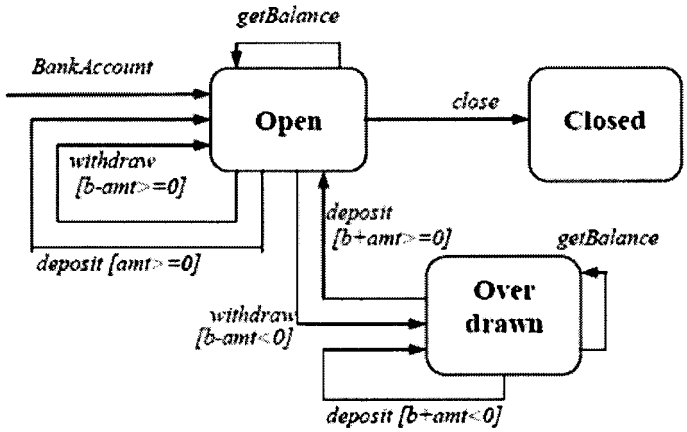

Figure 47. The bank account state model

For state Open, there are five different transitions coming out of it.

1. One can perform getBlanace on a bank account. This transition does not affect the state of the bank account; therefore, it ends up back on the same state.
2. One can deposit into a bank account. Noted by the constraint on the transition, as long as the deposit amount is not a negative amount, the end state is still Open.
3. One can withdraw from a bank account. If the balance after withdraw is not negative, the end state is still Open.

4. If the balance after withdraw is negative, the end state is Overdrawn.

5. One can also close the account. The end state is Closed

For state Overdrawn, there are three transitions coming out of it.

1. Deposit can be made to the bank account. If the ending balance after the deposit is positive, the end state is Open.

2. If the ending balance after a deposit is negative, the end state is still Overdrawn.

3. One can always get balance on an overdrawn account. The end state is still Overdrawn.

For state Closed, there is no transition coming out of it.

8.1.1. Step 1 – Create the Model

Create model elements representing the above state model as shown in Figure 48.

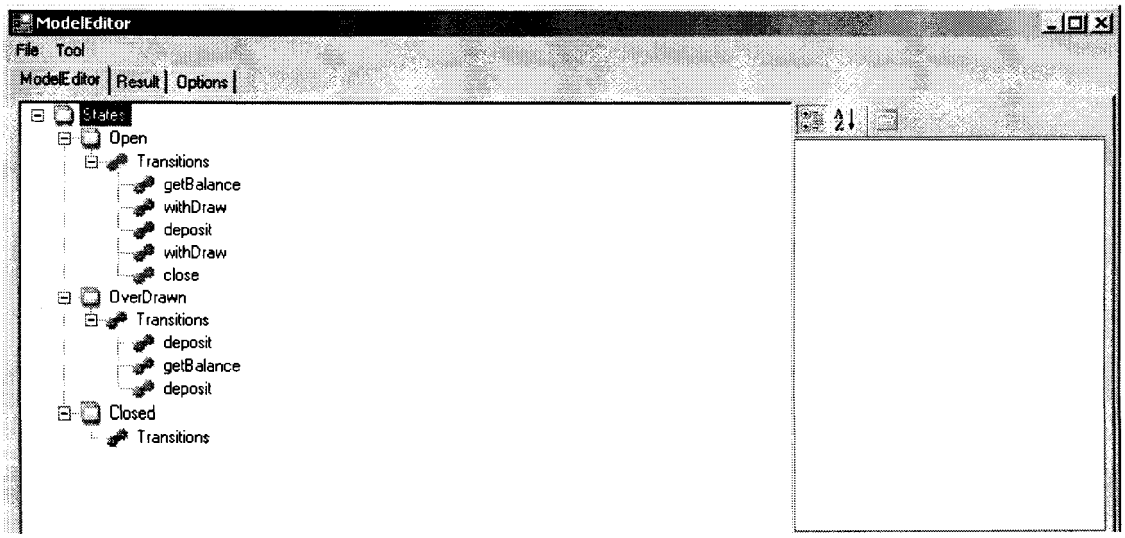

Figure 48. The bank account state model in model editor

8.1.2. Step 2 – Generate State Diagram in VS

Generate state diagram, include generated code files and model files in a C# project. The model file can be opened in VS as shown in Figure 49.

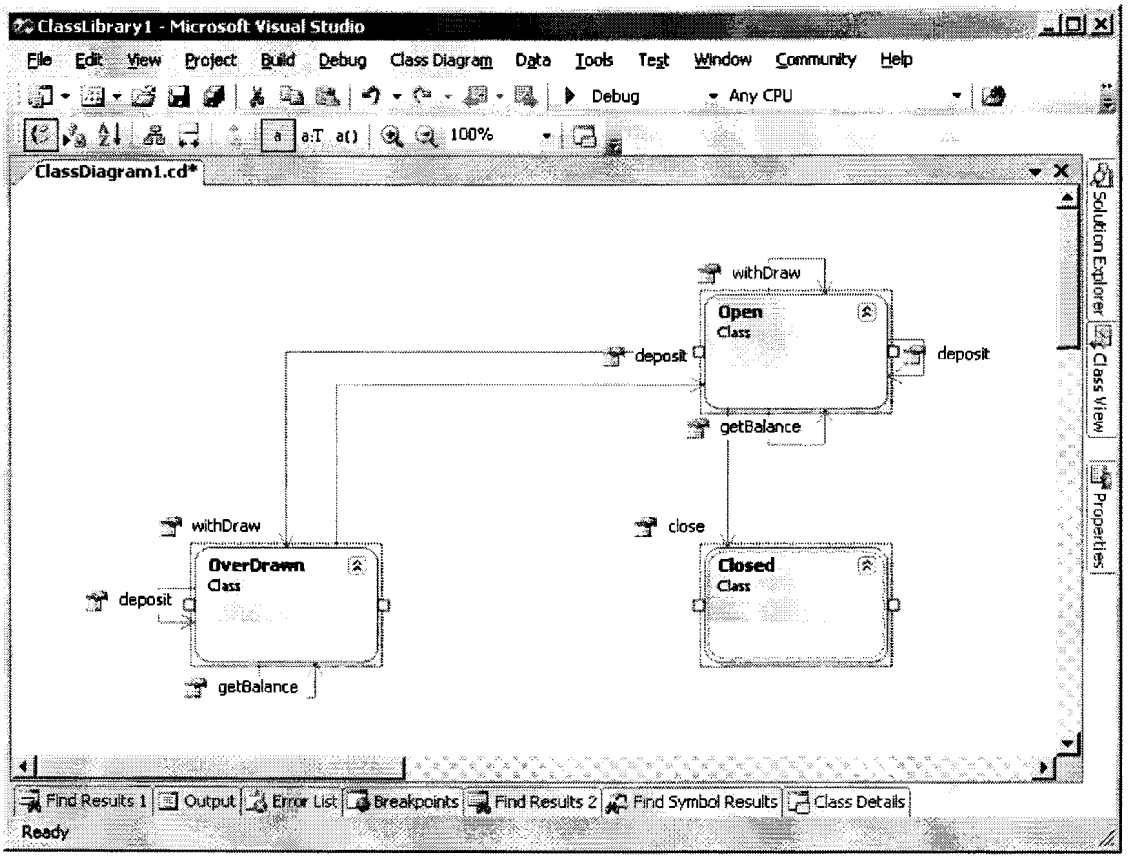

Figure 49. The bank account state model in VS

Figure 50 shows the generated CSharp file, class1.cs

```
class Open
{
   public Open getBalance {}
   /// <value>b-amt>=0</value>
   public Open withdraw {}
   /// <value>amt>=0</value>
   public Open deposit {}
   /// <value>b-amt<0</value>
   public Overdrawn withdraw {}
```

```
    public Closed close {}
}

class Overdrawn
{
    /// <value>b+amt>=0</value>
    public Open deposit {}
    public Overdrawn getBalance {}
    /// <value>b+amt<0</value>
    public Overdrawn deposit {}
}

class Closed
{
}
```

Figure 50. Generated code file for VS state model

Figure 51 shows generated xml model file, class1Diagram.cd

```
<ClassDiagram>
 <Font Name="Tahoma" Size="8.25" />
 <Class Name="Open">
  <Position X="0" Y="0" Width="0" />
  <TypeIdentifier>
   <FileName>Class1.cs</FileName>
  </TypeIdentifier>
  <ShowAsAssociation>
   <Property Name="getBalance" />
   <Property Name="withdraw" />
   <Property Name="deposit" />
   <Property Name="withdraw" />
   <Property Name="close" />
  </ShowAsAssociation>
 </Class>
 <Class Name="Overdrawn">
  <Position X="0" Y="0" Width="0" />
  <TypeIdentifier>
   <FileName>Class1.cs</FileName>
  </TypeIdentifier>
  <ShowAsAssociation>
   <Property Name="deposit" />
   <Property Name="getBalance" />
   <Property Name="deposit" />
  </ShowAsAssociation>
 </Class>
 <Class Name="Closed">
  <Position X="0" Y="0" Width="0" />
```

```
    <TypeIdentifier>
      <FileName>Class1.cs</FileName>
    </TypeIdentifier>
  </Class>
</ClassDiagram>
```

Figure 51. Generated model file for VS state model

8.1.3. Step 3 – Generate Test Cases – State Based Coverage

Following the state based traverse algorithm, the following collection of test paths are generated from the state based coverage.

{[new, Open], [withdraw, Overdrawn]}

{[new, Open], [close, Closed]}

There are two test cases generated based on the collection of test paths. Notice all three states are covered by the two test cases, as shown in Figure 52.

```
public static void TestCase1()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    // b-amt<0
    bankAccount.withdraw();
    // TODO: verify if the current state is Overdrawn
}

public static void TestCase2()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    //
    bankAccount.close();
    // TODO: verify if the current state is Closed
}
```

Figure 52. Generated test case - State coverage

TestCase1 starts with instantiating the BankAccount object and verifying that its state is Open. By calling withdraw with the constraint of b-amt < 0, the state of the object is changed from Open to Overdrawn, therefore, we need to verify that the state of the object is Overdrawn. TestCase2 also starts with instantiating the BankAccount object and verifying that its state is Open. By calling close, the state of the object is changed from

Open to Closed, therefore, we need to verify that the state of the object is Closed. By executing these two test cases, all three states, Open, Overdrawn and Closed are covered. However, not all possible transition among these states are covered, therefore, this suite of test cases can be used as a good check in tests that runs before every developer's code submission to provide basic coverage.

8.1.4. Generate Test Case Using Transition Based Coverage

Following the transition based traverse algorithm, the following collection of test paths are generated from the transition based coverage.

{[new, Open], [getBalance, Open]}

{[new, Open], [withdraw, Open]}

{[new, Open], [deposit, Open]}

{[new, Open], [withdraw, Overdrawn], [deposit, Open]}

{[new, Open], [withdraw, Overdrawn], [getBalance, Overdrawn]}

{[new, Open], [withdraw, Overdrawn], [deposit, Overdrawn]}

{[new, Open], [close, Closed]}

As shown in Figure 53, there are seven test cases generated based on the collection of test paths. Notice not only all three states are covered; all transitions are also covered with the generated test cases. This suite of test cases can be used as foundation for a good set of regression tests that runs daily to guarantee the quality of the software product.

```
public static void TestCase1()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    //
    bankAccount.getBalance();
    // TODO: verify if the current state is Open
}
```

```
public static void TestCase2()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    // b-amt>=0
    bankAccount.withdraw();
    // TODO: verify if the current state is Open
}

public static void TestCase3()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    // amt>=0
    bankAccount.deposit();
    // TODO: verify if the current state is Open
}

public static void TestCase4()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    // b-amt<0
    bankAccount.withdraw();
    // TODO: verify if the current state is Overdrawn
    // b+amt>=0
    bankAccount.deposit();
    // TODO: verify if the current state is Open
}

public static void TestCase5()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    // b-amt<0
    bankAccount.withdraw();
    // TODO: verify if the current state is Overdrawn
    //
    bankAccount.getBalance();
    // TODO: verify if the current state is Overdrawn
}

public static void TestCase6()
{
    BankAccount bankAccount = new BankAccount();
    // TODO: verify if the current state is Open
    // b-amt<0
    bankAccount.withdraw();
    // TODO: verify if the current state is Overdrawn
    // b+amt<0
    bankAccount.deposit();
    // TODO: verify if the current state is Overdrawn
}

public static void TestCase7()
{
    BankAccount bankAccount = new BankAccount();
```

```
    // TODO: verify if the current state is Open
    //
    bankAccount.close();
    // TODO: verify if the current state is Closed
}
```

Figure 53. Generated test cases - transition coverage

8.2. The Spacecraft Ascent State Model

A model that has more states and transitions is shown in Figure 54. This example examines how the tool does in generating test cases based on different coverage criteria. This model describes the ascent and earth orbit flight phases of a spacecraft [15].

8.2.1. Step 1 – Create the Model

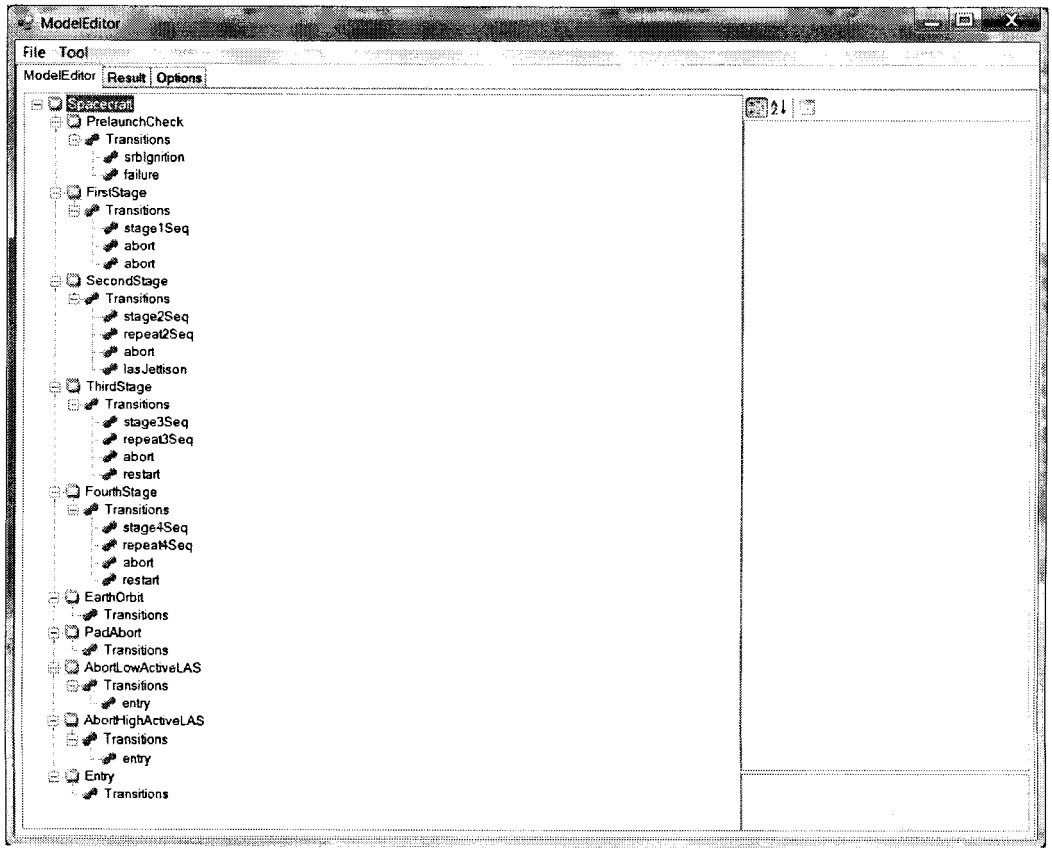

Figure 54. The spacecraft state model in model editor

8.2.2. Step 2 – Generate State Diagram in VS

Figure 55 shows the generated state diagram in Visual Studio.

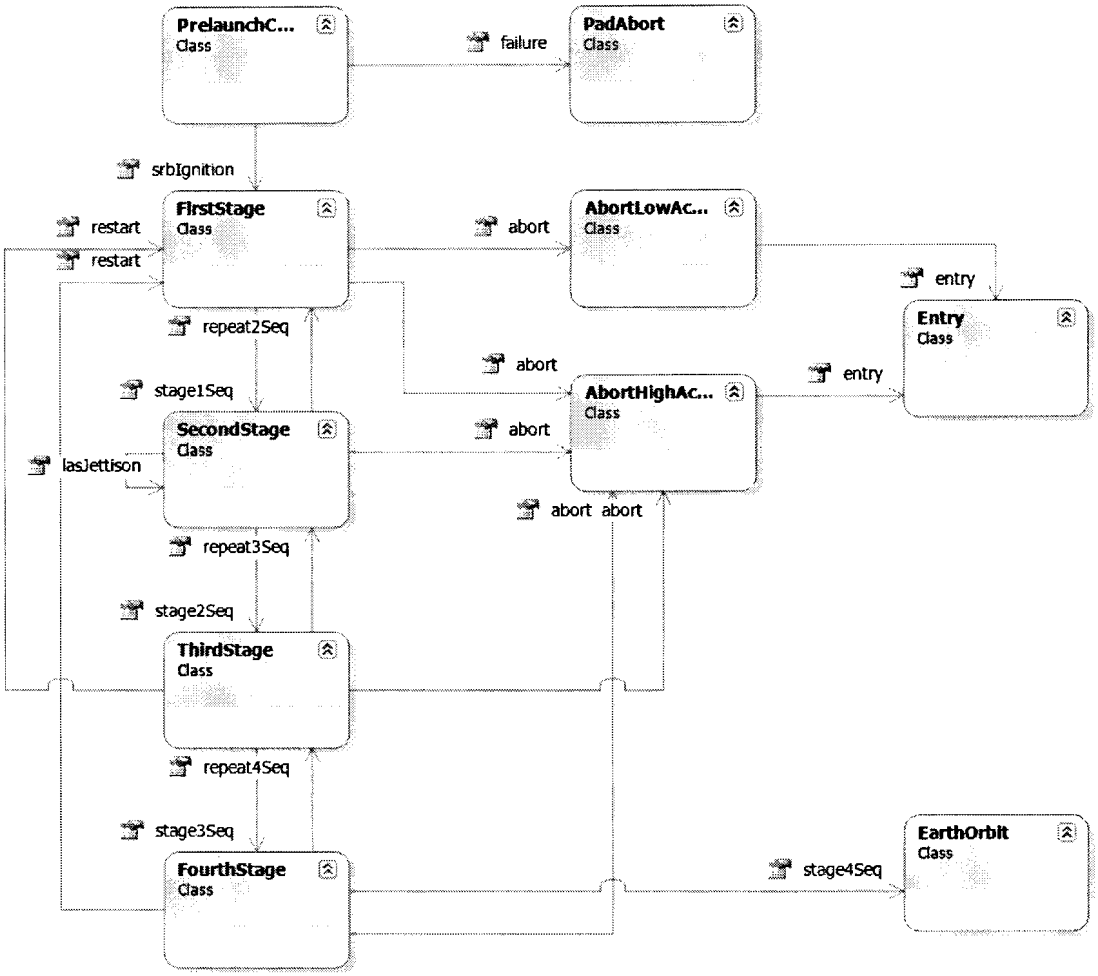

Figure 55. Generated spacecraft graphical state model in VS

8.2.3. Step 3 – Generate Test Cases – State Based Coverage

Figure 56 shows the generated test cases with the state based coverage.

```
public static void SpacecraftTest1() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
```

```
    //
    spacecraft.stage3Seq();
    // TODO: verify if the current state is FourthStage
    //
    spacecraft.stage4Seq();
    // TODO: verify if the current state is EarthOrbit
}

public static void SpacecraftTest2() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.stage3Seq();
    // TODO: verify if the current state is FourthStage
    //
    spacecraft.abort();
    // TODO: verify if the current state is AbortHighActiveLAS
    //
    spacecraft.entry();
    // TODO: verify if the current state is Entry
}

public static void SpacecraftTest3() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    // altitude <= 1.2e5
    spacecraft.abort();
    // TODO: verify if the current state is AbortLowActiveLAS
}

public static void SpacecraftTest4() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.failure();
    // TODO: verify if the current state is PadAbort
}
```

Figure 56. Generated spacecraft test cases - state coverage

8.2.4. Generate Test Case - Transition Based Coverage

Figure 57 shows the generated test cases with the transition based coverage.

```
public static void SpacecraftTest1() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.stage3Seq();
    // TODO: verify if the current state is FourthStage
    //
    spacecraft.stage4Seq();
    // TODO: verify if the current state is EarthOrbit
}

public static void SpacecraftTest2() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.stage3Seq();
    // TODO: verify if the current state is FourthStage
    //
    spacecraft.repeat4Seq();
    // TODO: verify if the current state is ThirdStage
}

public static void SpacecraftTest3() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.stage3Seq();
    // TODO: verify if the current state is FourthStage
    //
    spacecraft.abort();
```

```
    // TODO: verify if the current state is AbortHighActiveLAS
    //
    spacecraft.entry();
    // TODO: verify if the current state is Entry
}

public static void SpacecraftTest4() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.stage3Seq();
    // TODO: verify if the current state is FourthStage
    //
    spacecraft.restart();
    // TODO: verify if the current state is FirstStage
}

public static void SpacecraftTest5() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.repeat3Seq();
    // TODO: verify if the current state is SecondStage
}

public static void SpacecraftTest6() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.abort();
```

```
    // TODO: verify if the current state is AbortHighActiveLAS
}

public static void SpacecraftTest7() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.stage2Seq();
    // TODO: verify if the current state is ThirdStage
    //
    spacecraft.restart();
    // TODO: verify if the current state is FirstStage
}

public static void SpacecraftTest8() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.repeat2Seq();
    // TODO: verify if the current state is FirstStage
}

public static void SpacecraftTest9() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
    //
    spacecraft.abort();
    // TODO: verify if the current state is AbortHighActiveLAS
}

public static void SpacecraftTest10() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    //
    spacecraft.stage1Seq();
    // TODO: verify if the current state is SecondStage
```

```
    //
    spacecraft.lasJettison();
    // TODO: verify if the current state is SecondStage
}

public static void SpacecraftTest11() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    // altitude <= 1.2e5
    spacecraft.abort();
    // TODO: verify if the current state is AbortLowActiveLAS
    //
    spacecraft.entry();
    // TODO: verify if the current state is Entry
}

public static void SpacecraftTest12() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.srbIgnition();
    // TODO: verify if the current state is FirstStage
    // altitude >= 1.2e5
    spacecraft.abort();
    // TODO: verify if the current state is AbortHighActiveLAS
}

public static void SpacecraftTest13() {
    Spacecraft spacecraft = new Spacecraft();
    // TODO: verify if the current state is PrelaunchCheck
    //
    spacecraft.failure();
    // TODO: verify if the current state is PadAbort
}
```

Figure 57. Generated spacecraft test cases - transition coverage

In the state based coverage, there are only four test cases generated. However, all states are covered by this suite of test cases. Based on our rationale, this suite of tests can be used as a good suite of check in gated tests, for their relatively smaller number of test cases and coverage of all states.

In the transition based coverage, there are thirteen test cases generated. Every states and every transition is covered by this suite of test cases at least once. Based on

our previous reasoning, this suite of tests can be used as a good suite of regression test that can be ran daily, for their relatively larger number of test cases and relatively complete coverage of the source code.

9. CONCLUSION AND FUTURE WORK

9.1. Conclusion

The tool presented by this paper is a graphical tool for generating tests based on state models. As mentioned in the introduction section, testing is a vital part of the development process. Traditionally in a waterfall development methodology, testing is only one part of the development life cycle. In the new development process, testing is involved in every single aspect of the software life cycle. Here are three problems one would run into during the development process.

1. It is often not easy to create and maintain an unified state model that would be used throughout the entire development life cycle.
2. It is hard to create graphical based state models once there are too many states and too many transitions involved.
3. During the implementation phase, due to limited resources, source code submissions are sometimes unguarded. This not only produces poor quality software, but also does not guarantee conservation of resource. Sometimes it even leads to more resources wasted in identifying and fixing bugs that could have been caught much easily if a suite of tests were emplaced as part of the source code management process.

The first issue is solved by starting with the tree view based model editor. It is easy to construct a state model in the model editor. Anyone who is familiar with the

visual studio experience of property window should have no problem getting use to use the model editor to add/delete/edit states and transitions. The property window provides the user with a quick and error-resistant way to set start and end state on a transition. The tool also enables the user to save constructed model to an xml file on the disk and reload an existing xml file back into the tool. The xml style model file can also be viewed in IE or any other xml viewer for those who prefer to view state models in xml format. It also provides experienced users with a quick way of creating and modifying the model. Experienced users can open the model file in any other xml editor, edit the model and reload the xml model file back into the tool.

Second issue is solved with the help of an existing functionality from visual studio, the class diagram. It may seem odd at the first that we are using a class model to represent a state model. The arrange engine used in visual studio class diagram viewer is quite intelligent. The tool can walk through the states and transitions created in the tree based state model. It generates visual studio compatible model files and corresponding code files. Then, one can open the generated model in the visual studio. After clicking on the arrange button in visual studio, a nice and clean graphical state model is presented, without the hassle of connecting dots and other visual and artistic challenges.

Lastly, this paper presents a way to quickly and accurately generate a suite of test cases that provide good coverage for source code. Two different coverage algorithms are explored by my tool. The state based coverage guarantees that every

state in the state model is traversed at least once by the generated test paths. The transition based coverage guarantees that every transition in the state model is traversed at least once by the generated test paths. In most state models, the number of transitions is greater than the number of states. The suite of test cases generated by the state based coverage serves as a good suite of check in tests that a developer can run before source code submission. The suite of test cases generated by the transition coverage serves as a good suite of regression tests that can be ran daily to guarantee the quality of the software and catch bugs at an early stage.

9.2. Future Work

Some improvements can be made to the tool.

To enhance the usability of the model editor, drag and drop support can be added for quick edits. User should be able to drag and drop states and transitions from one state to another. Copy/Cut/Paste operation can also be added to enable faster model editing. Tool tips and help text can be added to commands under the file menu to provide information on what the commands do. When the number of states gets too large and the model becomes harder to navigate, sub edit windows could be added so that user can edit a state in a fresh new window.

To have a better overall user experience, tighter integration with visual studio can be explored so that graphical state model can be opened without opening visual studio. A visual studio project that contains the generated model files can be generated

automatically. When the user updates the state model in the model editor, the generated visual studio state diagram should sync up these changes.

To enrich the functionality of the tool, other coverage criteria can be added to the tool. Since the tool already knows how to walk through the collection of generated test paths, integrating with other traversal algorithms should be relatively straight forward.

10. REFERENCES

1. Meyer, B. "Object-Oriented Software Construction". *Englewood Cliffs, N.J.: Prentice-Hall*, 1988.
2. A. Pretschner. "Model-based testing," in ICSE '05: *Proceedings of the 27th international conference on Software engineering*, (New York, NY, USA), pp. 722–723, ACM Press, 2005.
3. P. V. R. Murthy, P. C. Anitha, M. Mahesh, Rajesh Subramanyan. "Test ready UML statechart models", pp. 75-82, SCESM, 2006
4. Wong, W.E., Horgan, J.R., London, S., and Agrawal, H. "A Study of Effective Regression Testing in Practice", *Proc. of the Eighth IEEE International Symposium on Software Reliability Engineering* (ISSRE'97), pp. 522-528, November 1997.
5. Rothermel, G., Untch, R.H., Chu, C., and Harrold, M.J. "Prioritizing Test Cases for Regression Testing". *IEEE Trans. on Software Engineering*, vol. 27, no. 10, pp. 929- 948, 2001.
6. Pretschner, A., Slotosch, O., Aiglstorfer, E., and Kriebel, S. "Model-Based Testing for Real - The Inhouse Card Case Study". *J. Software Tools for Technology Transfer*, vol. 5, pp. 140-157, 2004.
7. Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., and Stauner, T. "One Evaluation of Model-Based Testing

and Its Automation". *Proc. of the 27th International Conference on Software Engineering* (ICSE'05), pp. 392 – 401, 2005.

8. Xu, D., Xu, W., and Wong, W.E. "Automated Test Code Generation from UML Protocol State Machines", *Proc. of the 19th International Conference on Software Engineering and Knowledge Engineering* (SEKE'07), Boston, July 2007.
9. Robert M. Hierons, Krill Bogdanov, Jonathan P. Bowen, etc. "Using formal specifications to support testing", *Computing Surveys* (CSUR), Vol. 41 Issue 2, Feb, 2009.
10. Microsoft Corporation (2009). *Dynamic Source Code Generation and Compilation.* Retrieved October 20, 2009, from http://msdn.microsoft.com/en-us/library/650ax5cx.aspx.
11. Microsoft Corporation (2009). *System.ComponentModel Namespace.* Retrieved October 20, 2009, from http://msdn.microsoft.com/en-us/library/system.componentmodel.stringconverter.aspx.
12. El-Fakih, K., Yevtushenko, N., and Bochmann, G.V. "FSM-Based Incremental Conformance Testing Methods", *IEEE Trans. on Software Engineering*, vol. 30, no. 7, pp. 425-436, July, 2004.
13. Hong, H.S., Kim, Y.G., Cha, S. D., Bae, D.H., Ural, H. "A Test Sequence Selection Method for Statecharts", *Journal of Software Testing, Verification and Reliability*, vol.10, no.4, pp. 203-227, 2000.

14. Offutt, J., Liu, S., Abdurazik, A., and Ammann, P. "Generating Test Data from State-Based Specifications". *Journal of Software Testing, Verification and Reliability*, vol.13, no.1, pp. 25-53, 2003.

15. Corina S. Pasareanu, Johann Schumann, Peter Mehlitz, Mike Lowry. "Model Based Analysis and Test Generation for Flight Software". *Third IEEE International Conference on Space Mission Challenges for Information Technology*, Volume 00, pp. 83-90, 2009.