APPLYING MODEL BASED TESTING TO NETWORK MONITOR USER INTERFACE

By

ASHISH PANDAY

BCA, IIPS, Devi Ahilya University, 2006

A REPORT

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF COMPUTING AND INFORMATION SCIENCE
COLLEGE OF ENGINEERING

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Approved by:

Major Professor
Dr. Robby

Abstract

This report is a case study of applying Model-Based testing approach using SpecExplorer, which is a model-based testing tool developed by Microsoft, to test a component of Microsoft Network Monitor. The system under test is the UI of the Network Monitor feature, Parser Profiles Management.

Model-Based testing is a methodology for automated testing which not only automates the test execution, but the test design and generation. This approach starts by expressing an abstract model of the system which is a smaller subset of the product behavior, but retains essential elements which form the focus of the testing problem. A model-based testing tool creates a finite state machine from the model which is traversed to produce test cases. Thus, it provides more efficient coverage and flexibility in developing and maintaining test cases.

Table of Contents

List of Figures:

List of Tables:

Acknowledgements

I owe my deep gratitude to Dr. Robby, whose teaching enabled me to develop an understanding of Model-Based Testing and this report would not have been possible without his support.

I am grateful to Dr. John Hatcliff and Dr. Scott A. Deloach for being part of my committee.

I would also like to thank my manager, Ralph Case for his unwavering support and guidance, and Dr. Wolfgang Grieskamp, Nico Kicillof and Robert DuWors of the SpecExplorer team at Microsoft for their encouragement and advice throughout this project.

Chapter 1: Introduction

1.1 Problem Statement

The objective of this study is to apply Model-Based Testing approach using SpecExplorer and UI automation to test a component of the tool Microsoft Network Monitor. This component is the user interface for Parser Profiles, a new feature introduced in Network Monitor 3.4.

1.2 Document Overview

This report is structured as follows:

- Chapter 2 gives a brief background of the problem and the tools. It starts with an overview of the problem and objectives, followed by a discussion of Model-Based testing methodology, its benefits and relevance to User Interface testing. It discusses the tools, SpecExplorer, Network Monitor and UI automation. It ends with a description of the problem.
- Chapter 3 describes the design approach used to solve the problem. It gives the outline of test generation and execution, followed by the design approach used to design the model and implementation of the project.
- Chapter 4 contains excerpts from the project code implementation, including the model, exploration of state machines and slicing of the model. It ends with a brief overview of the implementation using a test harness.
- Chapter 5 concludes the report with results, limitation, future work and conclusion.

Chapter 2: Background

2.1 Overview

The Network Monitor test team at Microsoft has used various automation technologies through the years to develop, run and maintain test cases covering the product. The team runs two test labs with different platforms covering automatic and manual testing.

This study explores the usefulness and efficiency of using Model-Based testing as applied to the Network Monitor UI. At the time of this exercise, multiple teams were investigating the tool SpecExplorer for UI testing, but we were not aware a complete study on UI automation testing using the tool, thus it also served as a feasibility study for the application of the SpecExplorer tool to UI testing and integration into the team's test framework.

2.2 Model-Based Testing

As software becomes more and more complex, the challenges of testing software efficiently become bigger day by day. A full range of strategies, technologies and tools are used in order to solve testing problems. This trend is fuelling a growing market for automated testing tools which can significantly reduce manual effort and be more efficient and maintainable.

Model-based testing pushes the level of automation even further by automating the design, not just the execution, of test cases (Utting & Legeard, 2007). The feasibility and effectiveness of mode-based technology has been established by years of research, and a range of test-generation tools and strategies have been developed.

The *specification* of system is expressed as a model which is a smaller and more abstract version of the system itself, which strips out the non-essential details but still is expressive enough to cover the behavior which forms the focus of the test specification. Model-based testing tools generate a Finite State Machine (FSM) while exploring the model. For *on-the-fly* testing of systems (where the test cases are generated as the system is being tested), the tool walks through the FSM while the system under test is actively engaged. For test-case *generation*, the FSM is traversed and test cases are generated for different paths through the FSM.

Model-Based testing offers a host of benefits over regular automated testing tools. A model can effectively act like an oracle, from which a small set of test cases may be generated ensuring maximum coverage. It greatly improves the flexibility – once a model is defined, different features (components) of the system may be separated for test generation, and models may be combined together using model-composition.

It also helps in analyzing the system. Models may be used for visualization; which apart from showing the different states and transition of the system, also serves to validate the specification of the system and the model itself. Safety analysis may be performed to see if the system reaches any forbidden states, as well as deadlocks and livelocks may be identified (Jacky, Veanes, Campbell, & Schulte, 2008).

Model-Based testing is particularly relevant to testing User Interfaces, since they are systems designed to respond to step-by-step actions from the user. Thus at any point in a sequence of atomic actions used to accomplish a task through the UI, the system generally responds in some manner which indicate at least part of its state, and the transition.

2.3 SpecExplorer

2.3.1 Introduction

SpecExplorer is one of the third-generation of tools developed inside Microsoft for Model-Based testing (Utting & Legeard, 2007). The first version, SpecExplorer 2004 was developed in Microsoft Research and has since been moved to an engineering group in the Windows Server organization where it is being maintained and developed. SpecExplorer's history may be found on Wolfgang Grieskamp's blog at http://blogs.msdn.com/wrwg/archive/2009/10/28/the-spec-explorer-story.aspx. It extends Visual Studio to provide Model-based testing, visualization and analysis functionality, and the first public version is expected to ship with Visual Studio 2010. It is already being used by multiple teams inside Microsoft for automated testing.

A system's behavior in SpecExplorer is modeled in two ways, by writing the rules in C# annotated with rule attributes, and by defining scenarios as action patterns in a regular-expression style. It also has the ability to compose models, apply different strategies for test

case generation and it supports symbolic and combinatorial interaction with a rich set of features (SpecExplorer), http://msdn.microsoft.com/en-us/devlabs/ee692301.aspx.

A graphic showing how SpecExplorer fits into the testing lifecycle is reproduced here:

(Nico Kicillof, 2009)

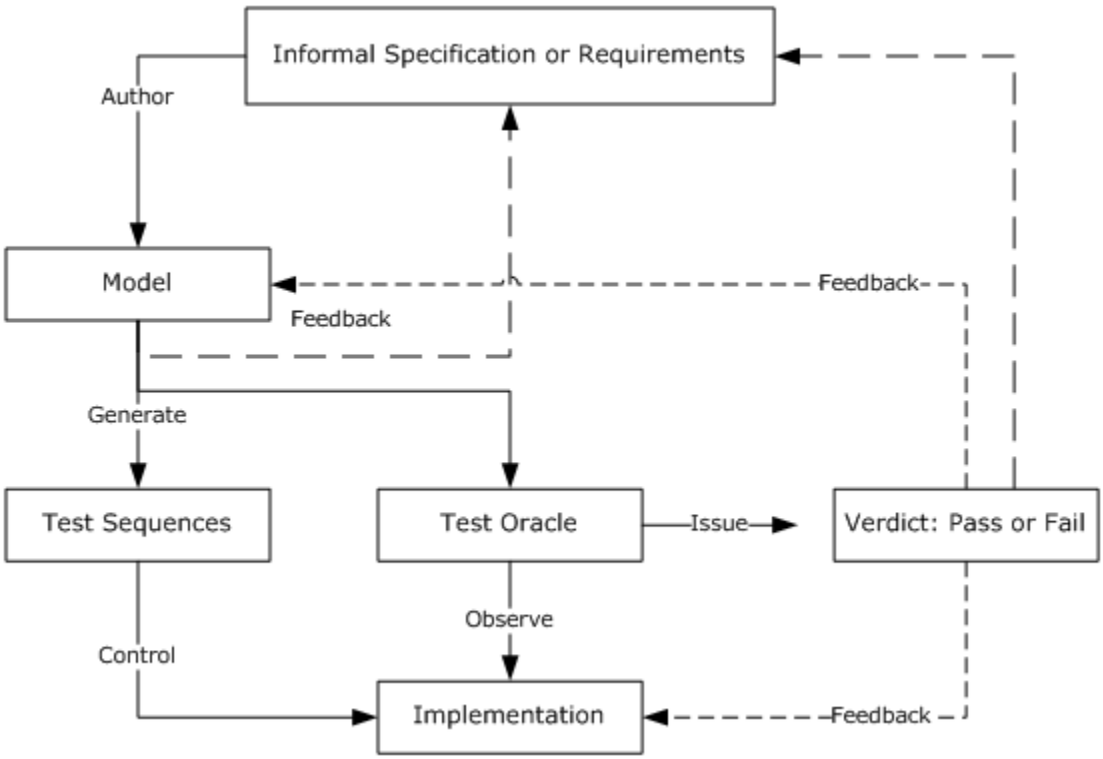

Figure 1: SpecExplorer in the test lifecycle.

By Nico Kicillof, Lead program manager for SpecExplorer on his blog:
http://blogs.msdn.com/SpecExplorer/

2.3.2 Model-Based Testing with SpecExplorer

Model-based testing solutions written with SpecExplorer have 3 basic components:

- Model
- Configuration
- Implementation

For the sake of a simple example, let's assume the system under test as a component of a media player, which has a button for Play/Pause, and a button for Stop, which is only enabled if the Play/Pause button has been pressed at least once, i.e. the player is not already stopped.

Model:

The model basically consists of a set of actions and preconditions which govern when a particular action is 'eligible' to run. These actions are written in C#, and annotated with attributes. SpecExplorer makes available different classes which facilitate parameter combination techniques, symbolic exploration, collection types, etc.

For the above SUT, we can design a model consisting of two actions, PlayOrPause and Stop, corresponding to the two buttons on the media player. It can be expressed in SpecExplorer in the following manner:

```
//Define State
enum PlayValues {Play, Pause, Stop};

//Declare State
static PlayValues State = PlayValues.Stop;

//The return value in this method is writen for the sole purpose of
displying it in the FSM exploration.
[Action("PlayOrPause()/result")]
static string PlayOrPause()
{
    if ((State == PlayValues.Stop) || (State == PlayValues.Pause))
    {
        State = PlayValues.Play;
        return "Play";
    }
    else
    {
        State = PlayValues.Pause;
        return "Pause";
    }
```

```
}

[Action("Stop()")]
static void Stop()
{
    Contracts.Requires(State != PlayValues.Stop);
    State = PlayValues.Stop;
}
```

Configuration:

The configuration is expressed in a scripting language called 'CORD'. A host of values may be configured to control the bound on number of states to explore, bound on path depth, test case generation etc. One may also create different configurations with a different set of actions for each (features of the system) to be explored and combined in various ways.

Cord scripts are used to define 'machines' which may be explored in the exploration manager. Machines may be used for the following important functions:

- Construct a model program from a Cord configuration.
- Bind parameter values to specific domains or expressions.
- Define 'Scenarios' – which are a specific sequence of actions defined in regular-expression style, generally in accordance with user scenarios in a Test Specification. Defining scenarios is at the core of test-generation role of SpecExplorer – it helps to contain the classic State-Explosion problem with Model-Based testing.
- Composition of different machines. For example, a scenario may be composed with a full model to produce a limited exploration.
- Generating test cases with a particular strategy for an exploration.

For the above media player example, we use Cord scripts to do the following in SpecExplorer:

1. Construct the configuration:

```
config Main
```

```
{
    bound steps = 1024;
    bound pathdepth = 1024;
    action static string Implementation.PlayOrPause();
    action static void Implementation.Stop();
}
```

2. Construct the machine for exploration:

```
machine ModelProgram() : Main
{
    construct model program from Main where namespace =
"SpecExplorerProject.Model"
}
```

On exploration, the above machine produces the following FSM:

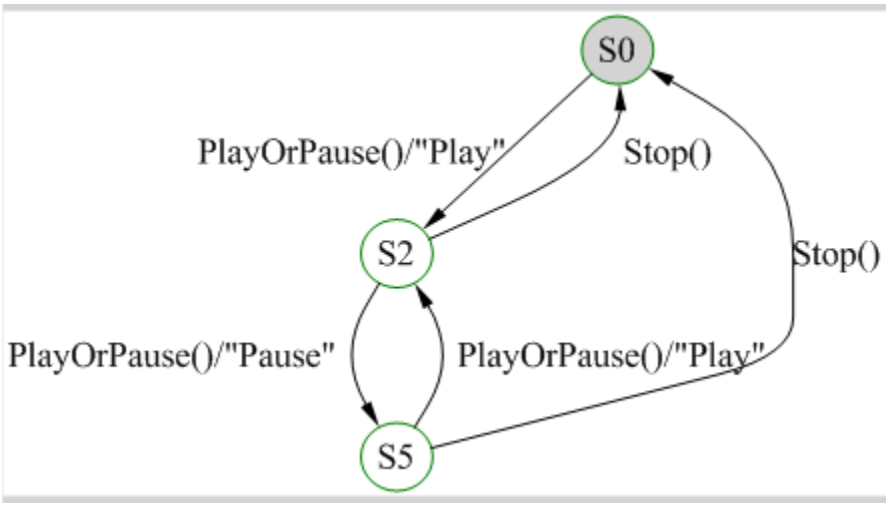

Figure 2: Exploration of the sample model code

3. Create test cases:

```
machine TestCases() : Main
{
   construct test cases for ModelProgram
}
```

SpecExplorer produces the following set upon exploration:

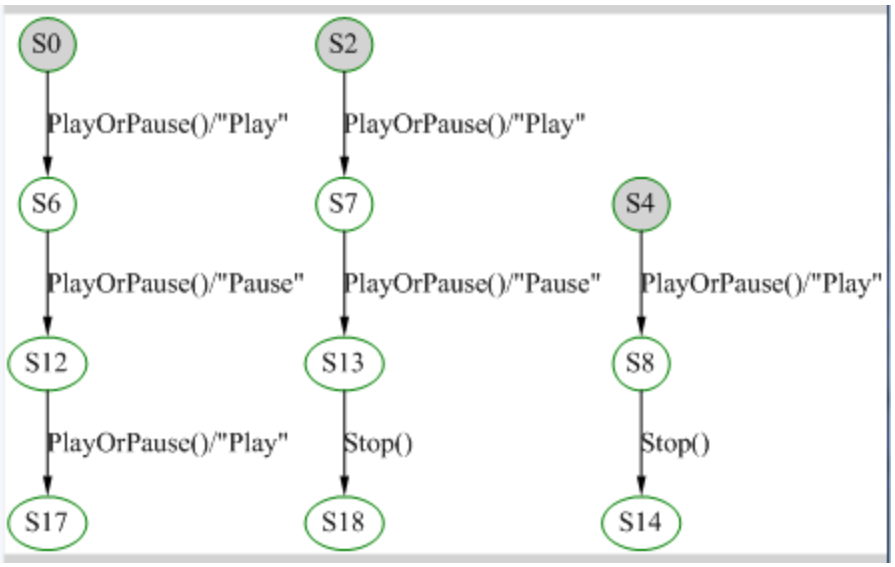

Figure 3: Exploring test cases for the sample model

4. Define a scenario: Since we have not defined any accepting states in this FSM, Spec explorer stops at the initial state and does not repeat. We can define a scenario to execute the actions according to our requirements:

```
machine PlayAfterStop() : Main
{
    PlayOrPause()*; Stop(); PlayOrPause(); Stop();
}
```

This machine ensures that the Play/Pause button is pressed once after each time Stop is used, preceded by any number of activating Play/Pause. Exploration is this machine yields:

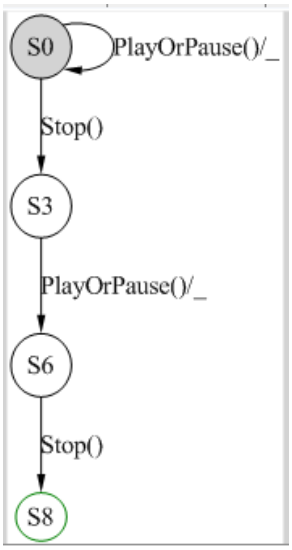

Figure 4: Sample scenario machine

5. Compose this scenario with the full model:

```
machine ScenarioSlice() : Main
{
   PlayAfterStop || ModelProgram
}
```

This exploration takes the scenario path as follows:

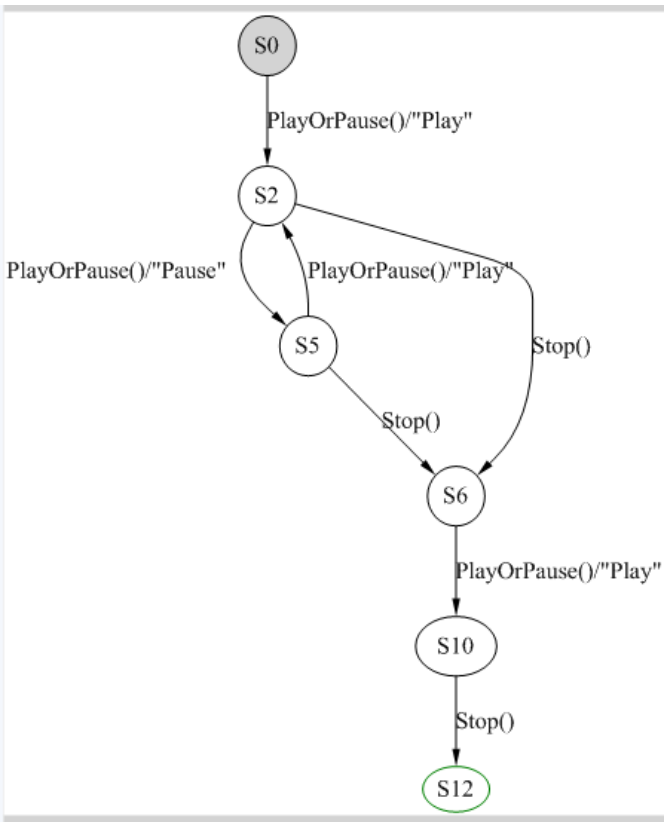

Figure 5: Composing the scenario with the full model

6. Generate test cases for the scenario:

```
machine SlicedCases() : Main
{
   construct test cases for ScenarioSlice
}
```

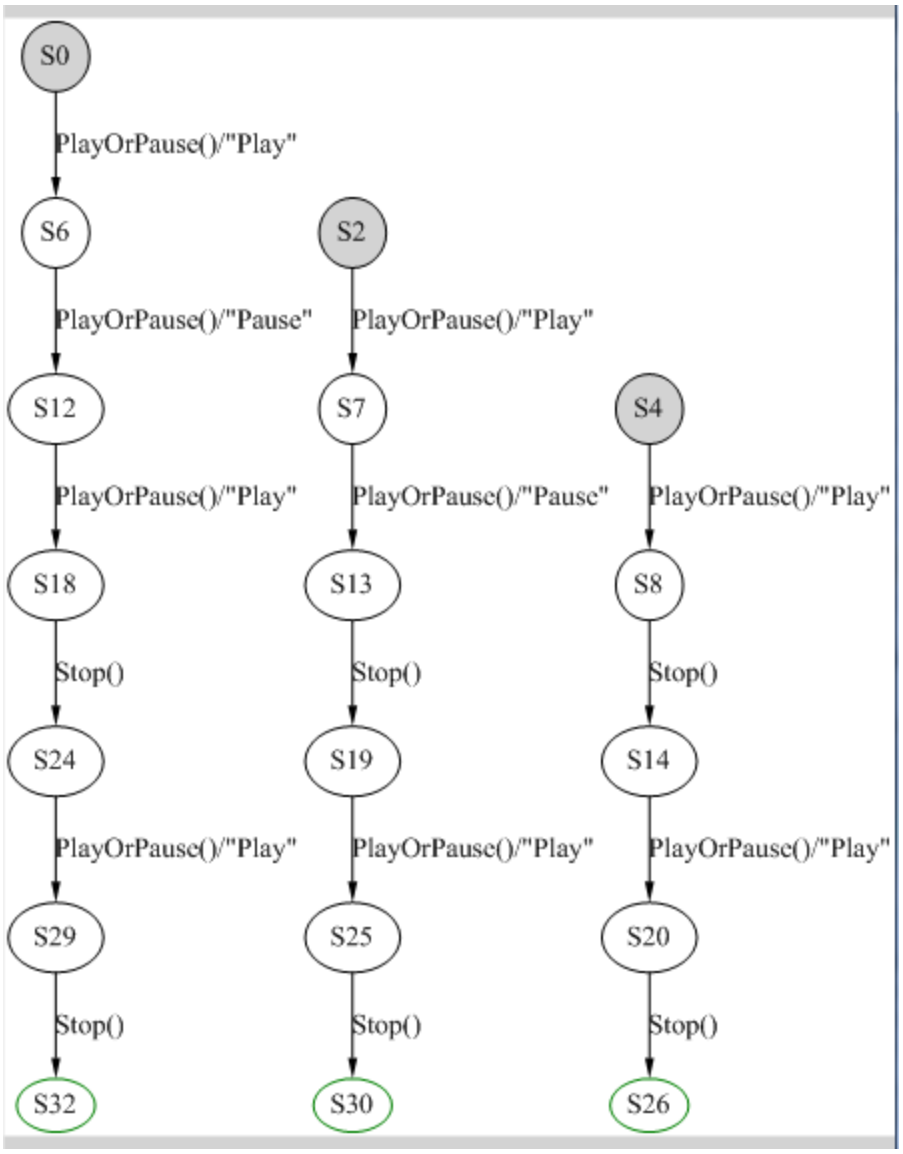

Figure 6: Exploration of test cases for the scenario composition

Test strategies for exploration:

SpecExplorer provides various test constructs along with parameter generation techniques in order to better control the generated tests. Some of the constructs are introduced below. For more information on these constructs, please refer to the SE help text (Microsoft, 2010).

Longtests: In this strategy, the exploration goes through a general tour of the graph, resulting in longer but fewer test cases. In order to demonstrate this, we add a condition to the model which defines an accepting state as one where the player is stopped:

```
[AcceptingStateCondition]
static bool IsStopped()
{
    if ((State == PlayValues.Stop))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Now if we write our test machine for the example above this way:

```
machine TestCasesLong() : Main
{
    construct test cases where strategy = "longtests" for ModelProgram
}
```

It would result in a single test case, which covers all the paths in the graphs:

Figure 7: Exploration of test cases with longtests strategy

The green outline signifies an accepting state in the above exploration.

Shorttests: This strategy results in a greater number of test cases. The exploration continues in a depth-first manner until an accepting state is reached, and there test case is cut, and another starts from the initial state, until the graph is completely covered.

```
machine TestCasesShort() : Main
{
    construct test cases where strategy = "shorttests" for ModelProgram
}
```

This results in the following exploration:

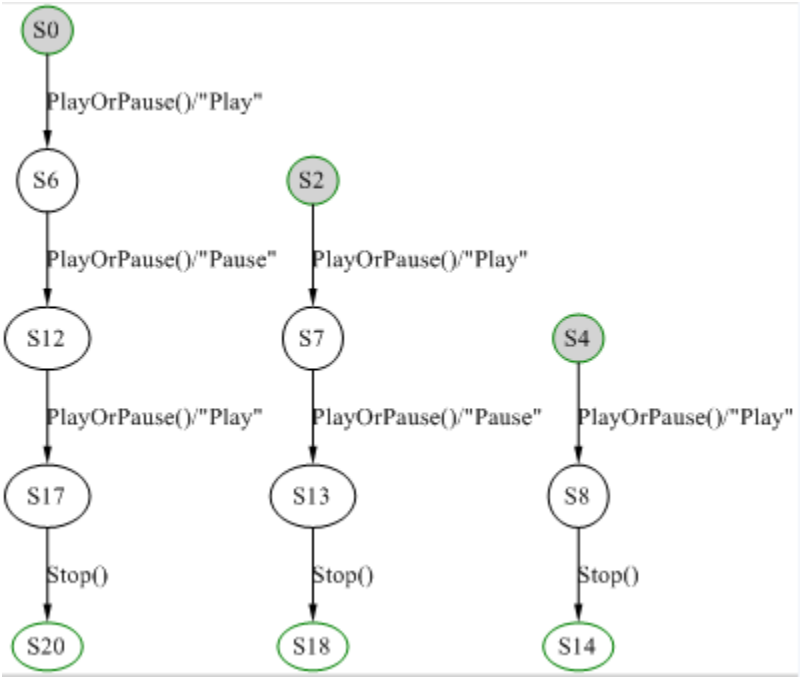

Figure 8: Exploration of test cases for the shorttests strategy.

The shorttests strategy also provides for two switches:

- StopAtAccepting: This switch has the effect of making each test case cut when any accepting state is found, and not only an accepting end state. If we use this switch, the result would be the same as the above exploration, because the accepting states are either the initial state or the end state.

- MinimumPathLength: An integer value may be provided for this switch, and the exploration then would stop either at an accepting end state, or an accepting state which is found at least that many steps farther from the initial state. To illustrate this, we remove the AcceptingStateCondition from our model to introduce other accepting states, and provide a value of 4:

```
machine TestCasesShortMinPath() : Main
{
    construct test cases where strategy =
"shorttests",minimumpathlength = 4 for ModelProgram
}
```

Since now all the states are accepting states, a minimum path length of 2 makes the exploration stop at any states after at least 2 actions (each action is 2 steps). This results in the following exploration:

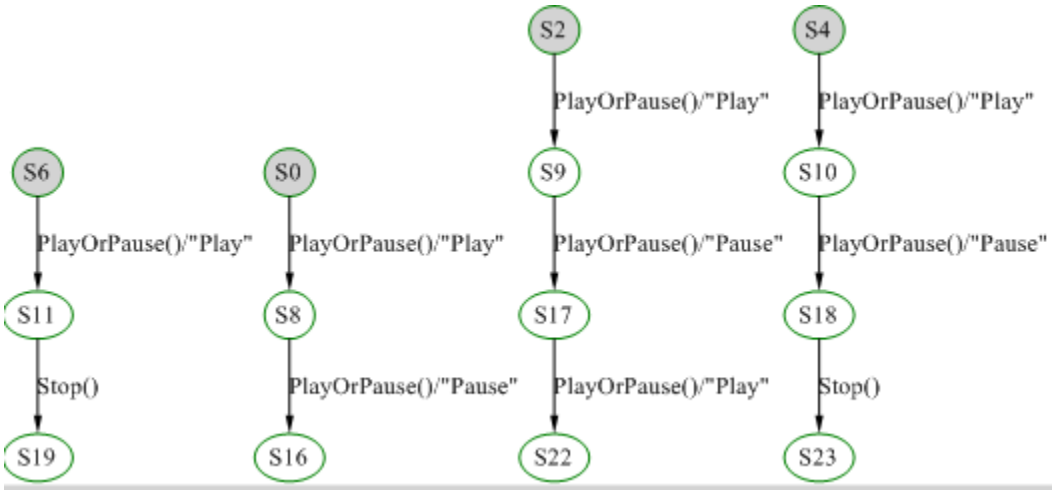

Figure 9: Using MinimumPathLength switch with shorttests.

Point and Shoot: The Point and Shoot construct provides a convenient way to control unmanageably large exploration graph towards a specific goal, as opposed to a scenario which is sliced using a sequence of pre-determined steps. For example, let us consider a test requirement for stress testing, where our media player is to be put through in an 'On' state for at least 12 hours. We achieve this, we add two state variables:

```
static int OnHours = 0;
static bool StressComplete = false;
```

We add a method to simulate the delay in our model, which accepts a value for the number of hours to delay, and only runs if the stress condition has not been met:

```
[Action("TimeDelay(Hours)/result")]
static string TimeDelay(int Hours)
{
    Contracts.Requires(Hours > 0);
    Contracts.Requires(!StressComplete);
    Contracts.Requires(State != PlayValues.Stop);
    OnHours += Hours;

    if (OnHours > 12)
    {
        StressComplete = true;
    }

    return OnHours.ToString();
}
```

And we modify the accepting state condition to reflect the requirement:

```
[AcceptingStateCondition]
static bool IsStopped()
{
    if ((State == PlayValues.Stop) && (StressComplete))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

We also constraint the domain of values for the value ‘Hours’ (the domain for an ‘int’ is theoretically infinite):

```
config Main
{
    action static string Implementation.PlayOrPause();
    action static void Implementation.Stop();
    action static string Implementation.TimeDelay(int Hours)
      where Hours in {2, 3, 5, 6, 8};
    bound states = 512;
    bound steps = 512;
    bound pathdepth = 512;
}
```

Even with the domain constraint, if we try to explore the model with these additions, it would result in a graph which is impossible to test exhaustively. The exploration would do needless repetitions of all possible hours, which do not add value to test coverage. To overcome this problem, we may use the Point and Shoot construct. The idea behind the construct is that we *Point* the exploration to go up a significant number of hours in the first step, and define *Shoot* as a way to reach the accepting end state after that step.

Although the above example is very simple and this can easily be achieved using scenarios, but in more complex systems, it may not be easy to define the first step so as to reach closer to the final state. In Point and Shoot, we define a machine to 'Point' as opposed to a sequence of actions.

We define a machine Point:

```
machine Point() : Main
{
    bind TimeDelay({6, 7})
    in
        construct model program from Main
    where
        namespace = "SpecExplorerProject.Model"
}
```

In order to create an idea of going towards completion, we define the following condition in the model:

```
        public static bool LongEnough()
        {
            if (OnHours > 8)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

And create another machine:

```
machine CompleteExploration() : Main
{
    bind TimeDelay(6)
```

```
    in
        construct model program from Main
    where
        namespace = "SpecExplorerProject.Model"
}
```

Exploration of the above machine:

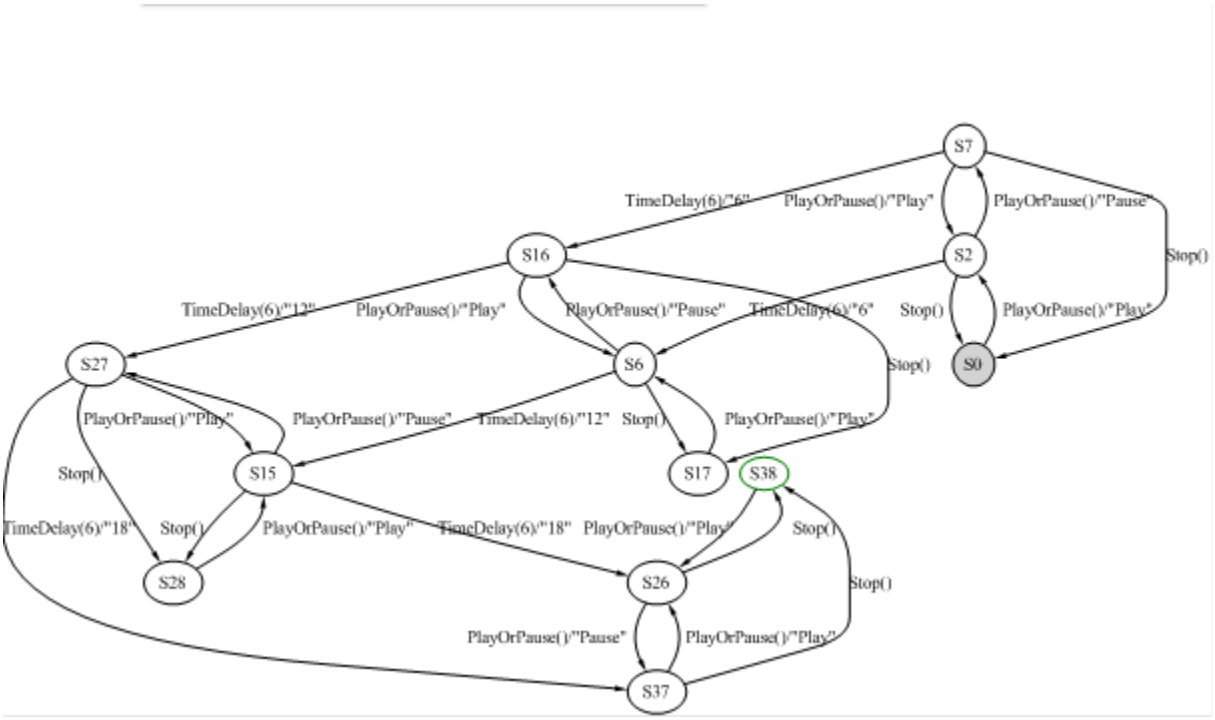

Figure 10: Exploration of CompleteExploration machine.

Now we use the complete construct:

```
machine PointAndShoot() : Main
{
    construct point shoot
    where
        PathDepth = 2,
        Shoot = "Shoot",
        Completer = "CompleteExploration" with
        (.
            SpecExplorerProject.Model.ModelProgram.LongEnough()
         .)
         for
            Point
}
```

The path depth ensures that the exploration is limited in the first steps. For details on the syntax of the construct, please see Cord Language Reference (Microsoft, 2010).

Finally we define a machine to construct test cases:

```
machine PointShootTest() : Main
{
    construct test cases where strategy = "shorttests" for PointAndShoot
}
```

Exploration of the above machine yields 36 test cases. The first few are reproduced here:

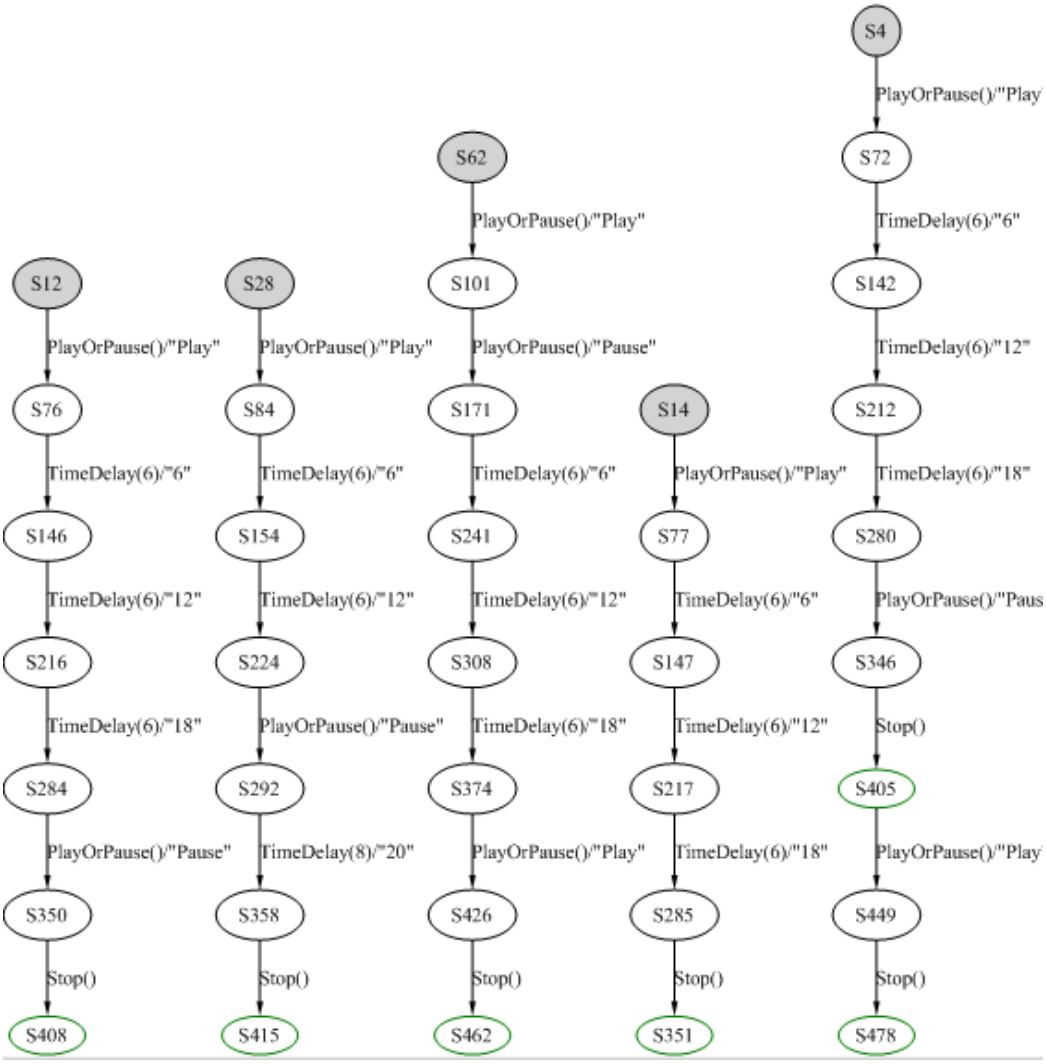

Figure 11: Portion of test cases exploration for PointAndShoot machine.

Implementation:

The implementation is written in C#, which acts as an *adapter* between the model and the system under test. Each action in the model has a corresponding C# method in the implementation. The implementation may be abstract if one only wishes to explore and analyze the model. When test cases are generated, SpecExplorer creates a test project which contains calls to the implementation methods in accordance with the model actions in the exploration.

For the example above, the implementation would have the necessary code to execute the Play/Pause and Stop actions on the SUT. Assuming the methods are exposed via a simple API in the library MediaPlayer, the implementation would be written as follows:

```
using MediaPlayer;
static MediaPlayer.Player Player1;

public static string PlayOrPause()
{
    Player1.PlayOrPause();
    return "";  //This value is disregarded.
}

public static void Stop()
{
    Player1.Stop();
}
```

2.4 Microsoft Network Monitor

2.4.1 Introduction

NetMon is a Network Traffic Analyzer. It facilitates capturing network data on the wire, deciphering network protocols, viewing the raw or parsed data and analysis. It is currently used by people internal and external to Microsoft for protocol development and testing, viewing network data, parsing protocol data and network troubleshooting.

NetMon comprises of three basic components:

- Capturing Engine, which allows capturing network traffic on the wire on different network adapters.
- Parsing Engine, this is used to parse raw network data into protocol data.
- User-Interface, which allows for viewing and analysis of the data.

2.4.2 Parser Profiles

The parsing engine in Netmon uses parsers written in a scripting language known as NPL. NPL is used to express network protocol structures. NPL parsers are hierarchical structures, where any parsing starts from the structure *Frame* and based on the traffic, it is parsed into different levels of protocols. The types are similar to a tree structure except for some inter-dependencies.

The parsing engine loads all available NPL parsers when it starts, in order to make the type structure. With network bandwidth increasing by the day, and parsers continuously being added for new and old protocols, loading all the parsers together becomes a performance problem for loading time as well as parsing time. The installer for parsers also grows larger.

To solve this problem, a new feature called *Parser Profiles* was proposed for Netmon 3.4. The idea is to identify groups of parses which are generally used together to solve a particular problem (e.g. protocols related to Microsoft Office, or Internet Explorer related traffic), and 'Stub out' the group from the core set by creating trivial parsers for all the dependent positions in the core set of parsers. This way, these groups may be combined together with the core set to make a parsers profile, and the users may be able to switch between them. Performance would be improved by not loading unnecessary parsers when the parsing engine is being used; and installers for different profiles may be added at any time.

2.5 Problem Description

Including the parser profile feature presented the challenge of testing all the combinations of actions that may be performed by the users on the UI; and the response of the system. The feature added new dialog boxes and buttons to provide the functionality, these are produced below.

2.5.1 UI components

Three new components were added to the UI:

A drop down box for quick access to switching between the profiles:

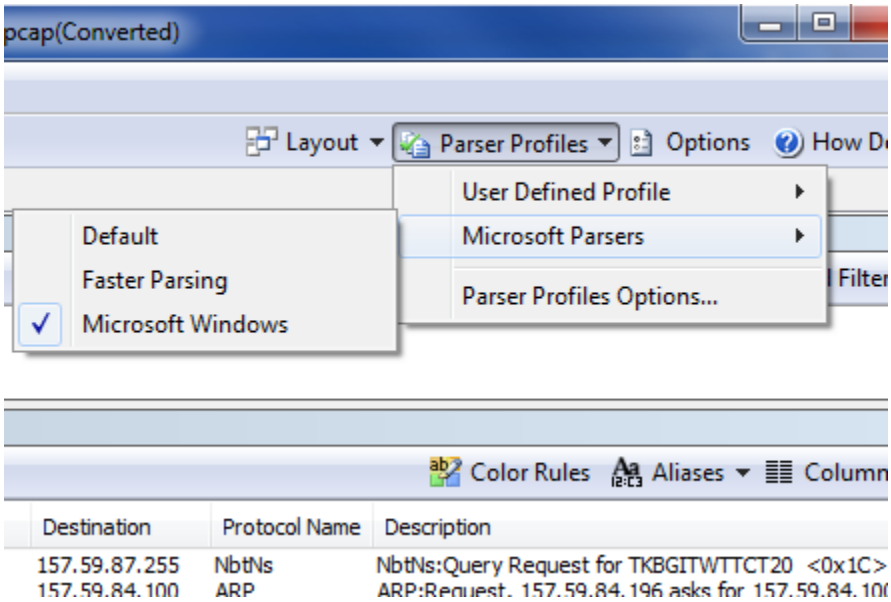

Figure 12: Quick access menu addition to the UI

A base dialog, which shows the list of available profiles, and makes available actions for creating new profiles, editing or deletion:

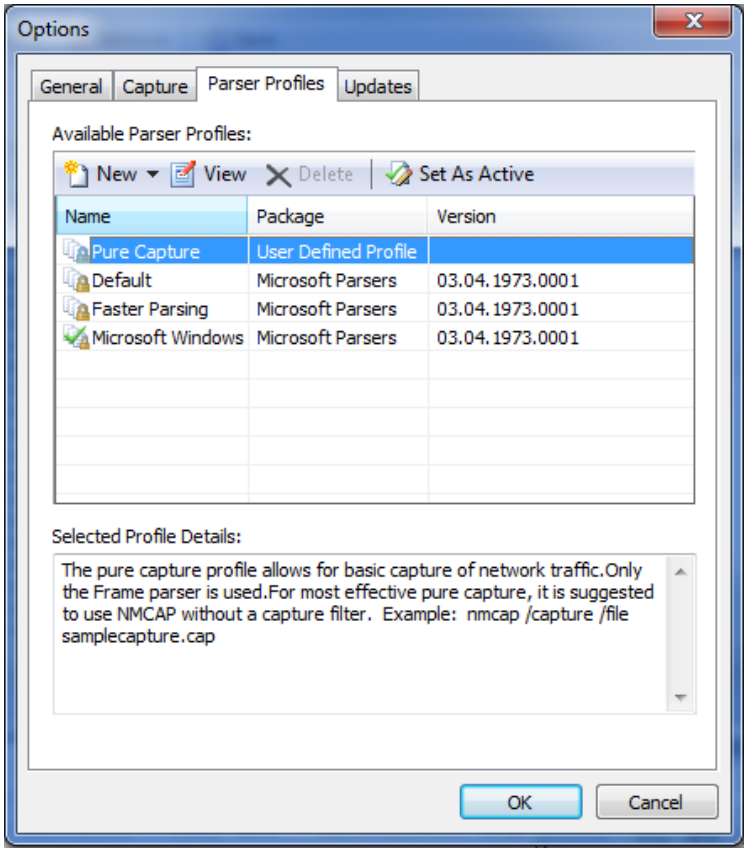

Figure 13: Base UI dialog for parser profiles

A child dialog which may be activated through the base dialog, which shows the attributes of a particular profile, and provides the editing actions:

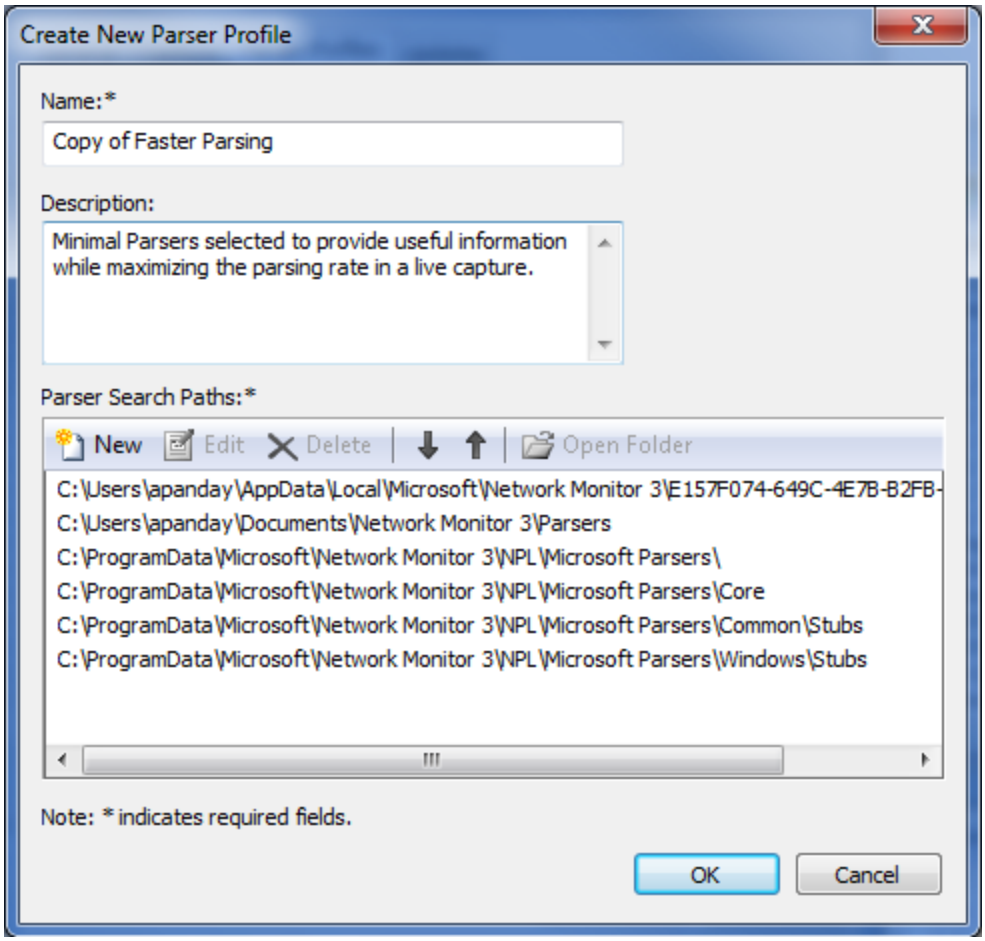

Figure 14: Child UI dialog for parser profiles

2.5.2 Test Specification and UI testing scope

The test specification includes:

- Network Monitor installer will create a set of packaged profiles, which may not be edited or deleted. A set of folders will be created on the computer, containing different groups of parsers, and the profiles would comprise of a combination of these folders.
- If it is an upgrade install, the previous parsers will be packaged into a new profile and included in the set of profiles.
- A user must be able to switch between profiles and Network Monitor should remember the selection.
- If a profile is set as ‘Active’; the parsers in that profile must be used by the parsing engine, and the user should be able to see the correctly parsed data through the UI.

- Each profile will be specified as having a name, a description, and a set of folder paths containing parsers. If the user sets it as active, it would be compiled by the parsing engine. If the compilation is successful, the profile is set as active, and the user sees data parsed with the specified set of parsers. If it is not successful, an error message is shown, and the current active profile selection is unchanged.
- A user should be able to create a new profile, which must have a name and at least one folder path in the list. Such a profile would be a user-defined profile.
- A user should be able to extend from existing default and user-defined profiles. Such a profile would also be a user-defined profile.
- A user should be able to edit or delete any user-defined profile.

Apart from this list, there was a range of scenarios identified for the installer/capturing/command-line usage and the Netmon API, but they are not relevant to UI testing. The first two are installer scenarios which are not directly related to UI testing, but they form the knowledge that is used to start the UI tests with, as explained in the next section.

2.5.3 Testability

2.5.3.1 *Validity of a profile*

Since NPL allows inter-dependencies between protocols in different layers, it is not possible to specify a set of rules to decide if a set of parsers would compile if they are put together, without actually compiling them. In absence of such rules, it is not possible to model, truly creating a 'New' profile with parsers that are unseen, and decide if it is a valid profile or not. Even if compilation was achieved using the Netmon API, it presents a problem to verify if the user can correctly set this profile as active.

In order to cover the test requirements, we solved this problem by abstracting a level. When NetMon and parsers are newly installed, the installer puts a set of folders on the computer containing parsers relevant to a particular category, e.g. Common, Windows Stubs, Full Windows, Office, etc. The profiles installed by default are made by choosing a combination of

these folder paths. We selected another set of parsers by in addition to the default ones after talking to the parser team, and with the knowledge of how these different folders combine, the test cases produce new profiles which represent combination that are not installed by default or emulate existing profiles. Thus, the model only 'knows' that the presence or absence of particular folders in a list indicates a particular 'profile' without actual compilation.

2.5.3.2 *Verification of active profile*

An active profile would parse capture traffic according to the set of loaded parsers, and depending upon the parsing, different data would show up in the frame details view of the UI. It is not possible to find correctness of that data unless the data is actually parsed using the test cases, but that approach would use the parsing engine, which itself is indirectly under test through this test process. It is possible to use another medium (API, etc.) but the parsing engine is the same, thus such parsing would verify corresponding behavior between two ways to parse. While useful, this is outside the test scope of this study.

To solve this problem, we used a feature in Network monitor called Display Filter. It is possible to filter the displayed data according to some filter expression (e.g. TCP.Source) which throws out any frames that do not match the expression from the frame summary display. Thus, we identified filter expressions which are present in a particular profile but not others. If such a filter is applied, frames would be displayed only if the relevant parsers are loaded, as stub parsers do not parse the data but put everything under a trivial field. Such displayed frames may be counted, and also saved in a file to compare to existing baseline files.

The problem with this approach is that some profiles may be subsets. For example, the Microsoft Windows profile parser set which is installed by default, is the superset of all full parser sets that are installed. Thus any filter would be relevant to this profile. To solve this, we identified the distinct filters so that only a particular combination of filters would work for different profiles. We put them in a manner similar to the following:

Table 1: Specifying filters against profiles for testability

	Filter1	Filter2	Filter3	Filter4	Filter5	...
Profile1	X				X	
Profile2	X	X				
Profile3	X		X	X	X	
...	X	X	X		X	

Thus by running all the filters on a profile, it would be possible to distinctly identify each one. For the end cases, at a minimum (in a trivial profile) a filter specification of *Frame* would always work, and in a superset, all of them would work.

2.5.3.3 *UI Automation Framework*

The existing UI automation framework for Network Monitor uses a UI automation tool internally available at Microsoft. This tool accepts a sequence of component identifiers and actions to be performed on the components. This sequence is written into an XML-based file and the file is supplied as input to the tool.

Since this tool works through an input file before starting the test, the UI may not be actively engaged using this tool while testing. The test case must 'anticipate' the response from the UI after performing a sequence of actions, and take further actions according to the response. If a particular action cannot be taken (e.g. when the UI is an unexpected state), the test case would fail. It allows for only minimal conditional branching in the input sequence.

Thus, the 'adapter' of the Model-Based solution must create such a sequence to be used as input for the UI automation. Each C# method would specify such automation actions in accordance with exploration of actions in the model. When the test cases are executed, each test case produces an input file, which subsequently may be used for execution against the UI.

This is essentially a passive approach of test-case generation by traversing the finite state machine, as opposed to on-the-fly model-based testing.

Chapter 3: Design Approach of Test Generation and Execution

3.1 Overview

We started with identifying the different components of the system and information flow. We decided to follow these steps:

1. Design the model and machines.
2. Explore and visually analyze, validate the model and specification against each other.
3. Write the implementation code and generate test cases.
4. Write a test harness to get input files from test case execution and drive the UI automation.

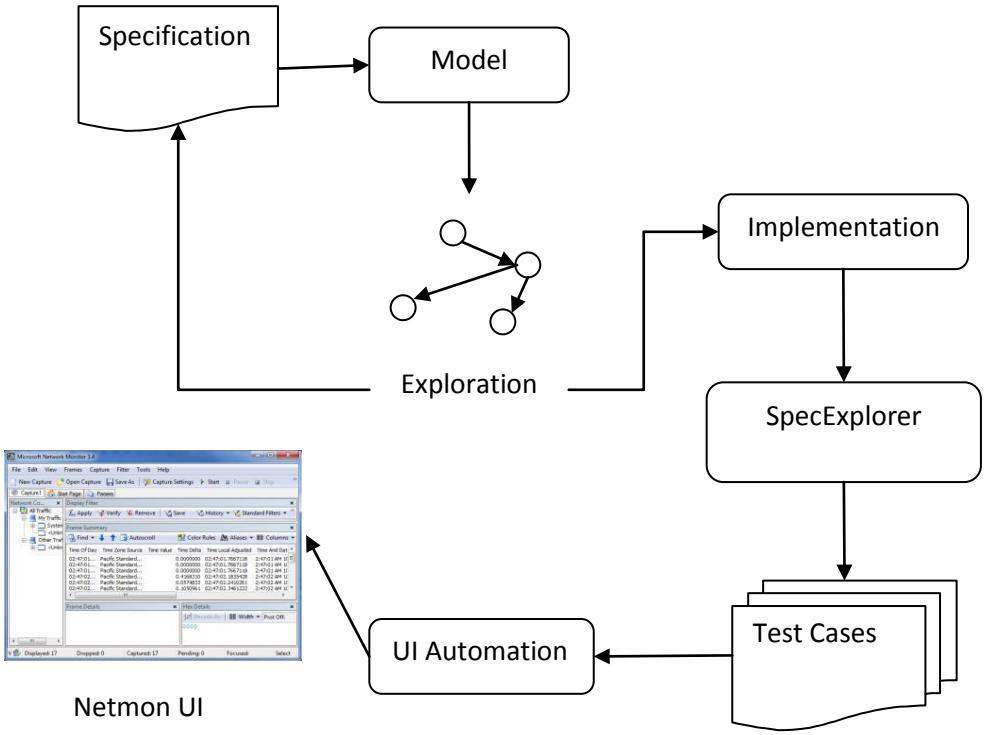

Figure 15: Design approach for the test process

The model design and exploration phase are briefly described in the next section.

3.2 Model Design

This chapter describes the approach followed to express an abstract model of the system in SpecExplorer. After studying the system, we abstracted out the essential actions and state, which are described below with the associated state machine. We started with the fundamental set of actions and state described in this section and then extended it to match the user interface. For the sake of a concise design description, only the basic state data and actions are discussed here. Trivial actions and complex decision logic is omitted.

SpecExplorer allows both Static- and Instance-based models to be explored. In static modeling, we have a set of actions any of which might be executed at a particular position in the exploration given the respective preconditions. In instance-based modeling, one may instantiate a group of actions as part of a class, and the actions from more than one object may simultaneously be candidates to be fired and thus make a sequence. We chose to write a static model, since multiple instances of the Parser Profile feature may not be run simultaneously.

The actions for the whole model may be grouped in two broad categories:

a. Actions related to the Base UI dialog.
b. Actions related to the Child UI dialog.

The last two sections combine the smaller models into one state machine, and apply model-slicing techniques to achieve a smaller number of test cases after exploration.

3.2.1 Base UI dialog

The base dialog provides the following core actions:

1. Set a profile from the list as active, and
2. Delete a profile from those displayed in the list.

The results for these actions may be achieved by a combination of *smaller* trivial actions, such as selecting a profile from the list and then pressing the set as active button, pressing the cancel button and opening the dialog again, etc. We start with declaring the abstract actions.

```
SetAProfileAsActive (Integer ProfileIndex);

DeleteAProfile (Integer ProfileIndex);
```

The state of the base dialog comprises of a list of profiles, and the currently active profile. We see in the profile details that a profile has a name, a description, and a list of folder paths which have the profiles parsers. Also associated with a profile is an attribute which says whether or not it is a user-defined (Custom) profile. Only custom profiles may be edited or deleted.

Since the name and description are of no consequence to the model, we can omit them when defining a profile's state. Thus we can describe the above mentioned state as:

```
Profile
{
      Boolean IsCustom;
      List<Path> PathList;
}
```

The list of folder paths is a pre-determined list here, which is installed by default with Netmon. As discussed in the testability section (2.5.3.1), it is not possible to determine the validity of a profile with random paths. Thus, with the knowledge of how the default parsers (grouped in paths) compile together, the tests would try to produce valid and invalid combinations to generate different profiles or errors to cover the test requirements.

Thus we may simply enumerate the paths:

```
Enumeration Paths
{
      SparserPath,
      WindowsStubsPath,
      WindowsFullPath,
      :
}
```

Using these definitions, we declare the state for the Base UI dialog as follows:

```
Struct BaseState
{
      List<Profile> Profiles;
      Integer ActiveProfileIndex;
}
```

For test purpose, we declare another action for the verification of the active profile:

```
VerifyActiveProfile ();
```

In order to keep the exploration small, we make sure that the verification step only happens immediately after setting a profile as active. With that requirement, this action could have been included into the SetAsActive step, but verification is much more complicated that simply setting a profile as active. Thus it is treated as an independent action for the ease of implementation, and also for flexibility of verification without having to necessarily set a profile as active. Also, it is important for this action to have a parameter, as explained in the next paragraph. To achieve the sequencing, we introduce another state variable:

```
Struct BaseState
{
      List<Profile> Profiles;
      Integer ActiveProfileIndex;
      Boolean IsVerificationActive;
}
```

The verification step does not have any significance for the model but is purely for implementation purpose. As discussed in the testability section (2.5.3.2), the implementation would need to know which filter combination should work in order to verify an active profile, we introduce an enumeration of filters here as well:

```
Enumeration Filters
{
      Filter1,
      Filter2,
      ...
      Error
}
```

In order to determine the particular filter value to use, we declare a helper method:

```
FindFilterForPaths (List<Paths>) returns Filters;
```

This method will accept a list of folder paths, and apply the knowledge of how they combine to find a particular filter combination suitable for a profile with such paths. Or it may return a special designated filter *Error*, if the combination of the folder paths cannot be compiled. Now we introduce a parameter in the Verify method:

```
VerifyActiveProfile (Filters Filter);
```

The purpose of this parameter is to pass this information to the implementation. When the implementation action is called with a particular filter set (or Error) it would either verify that the filter set is suitable for the active profile or anticipate an error message. Such a technique helps keep any decision-making in the model as opposed to the implementation, as long as they can agree on a common set of values for a parameter.

We initialize the base state in accordance with a fresh install of Netmon:

```
Profile Pure, Default, ...;

Pure.IsCustom     <-    false;
Pure.PathList     <-    { Paths.SParserPath };

Default.IsCustom  <-    false;
Default.PathList  <-    { Paths.SParserPath, Paths.WindowsStubsPath, ... }

...

BaseState _BaseState;

_BaseState.Profiles                <-    { Pure, Default, ...}
_BaseState.ActiveProfileIndex      <-    0;
_BaseState.IsVerificationActive    <-    false;
```

From the system specification, we know that any profile may be set as active, but only custom profiles may be deleted. Also, an active profile cannot be deleted.

In order to express that any ProfileIndex passed as parameter to SelectAProfile or DeleteAProfile should fall in the range of existing profiles in the list, we would restrict it to the values between 0 and the length of the Profiles list, excluding the latter.

Using these conditions and state definitions from above, we define the base dialog actions as follows:

```
SetAProfileAsActive (Integer ProfileIndex)
{
      Precondition ( _BaseState.IsVerificationActive = false);

      Precondition ( ProfileIndex >= 0
                        AND ProfileIndex < _BaseState.Profiles.Count );

      _BaseState.ActiveProfileIndex <-    ProfileIndex;
      _BaseState.VerificationFlag   <-    true;
}

DeleteAProfile (Integer ProfileIndex)
{
      Precondition ( _BaseState.IsVerificationActive = false );
      Precondition ( _BaseState.Profiles [ProfileIndex].IsCustom = true );
      Precondition ( _BaseState.ActiveProfileIndex != ProfileIndex );

      Precondition (ProfileIndex >= 0
                        AND ProfileIndex < _BaseState.Profiles.Count );

      _BaseState.Profiles.RemoveAtIndex(ProfileIndex);
}

FindFilterForPaths (List<Paths>) returns FilterSets
{
      ...   //Compilation logic
}

VerifyActiveProfile(FilterSets Filter)
{
      Precondition (_BaseState.IsVerificationActive = true );
      Precondition ( Filter =  FindFilterForPaths (
            _BaseState.Profiles[_BaseState.ActiveProfileIndex].PathList ));

      _BaseState.IsVerificationActive = false;
}
```

The precondition in the VerifyActiveProfile method ensures that the only value of Filter it may be called by is right one for the current profile, and hence the corresponding method in the implementation also gets called by that value.

At this point, if we assume the only two profiles declared in the base state, the state machine is drawn below.

Please note that the Delete action is not called here, since none of the declared profiles are custom and we have not provided actions to add custom profiles yet. Also, the parameter 'Filter' for the verify action would always have the same value for the same profile, which only matters for the implementation and does not increase the number of states.

The Accepting states are not identified in this machine, since from the UI perspective, for these actions, any state is an accepting one. In order to generate tests, accepting states may be defined, but using SpecExplorer's scenario-based approach for slicing, it is not required, as described in the last section.

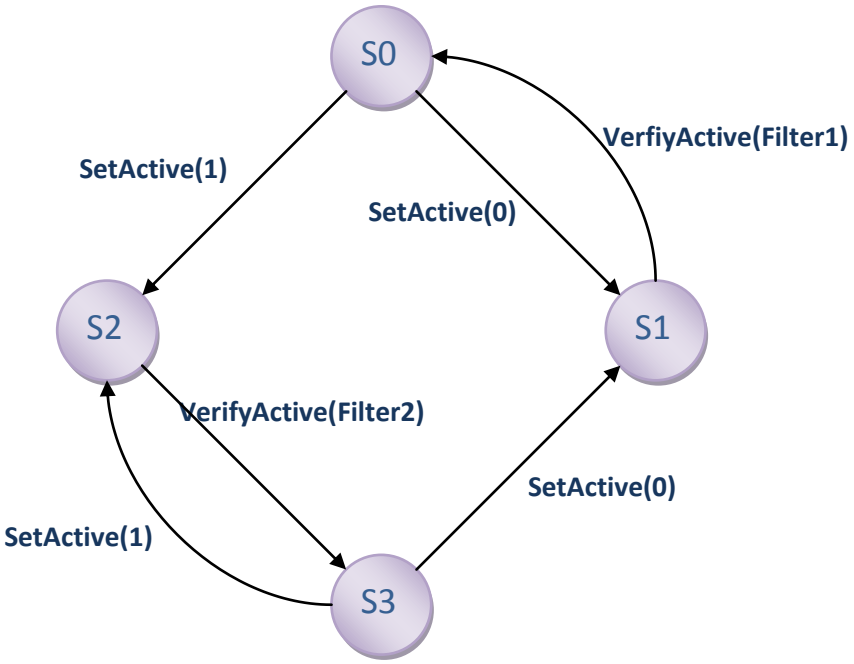

Figure 16: FSM for base dialog actions

If for the sake of watching the effect of the Delete action, we initialize the default profile with its Custom attribute assigned the value *true*, the state machine would change to:

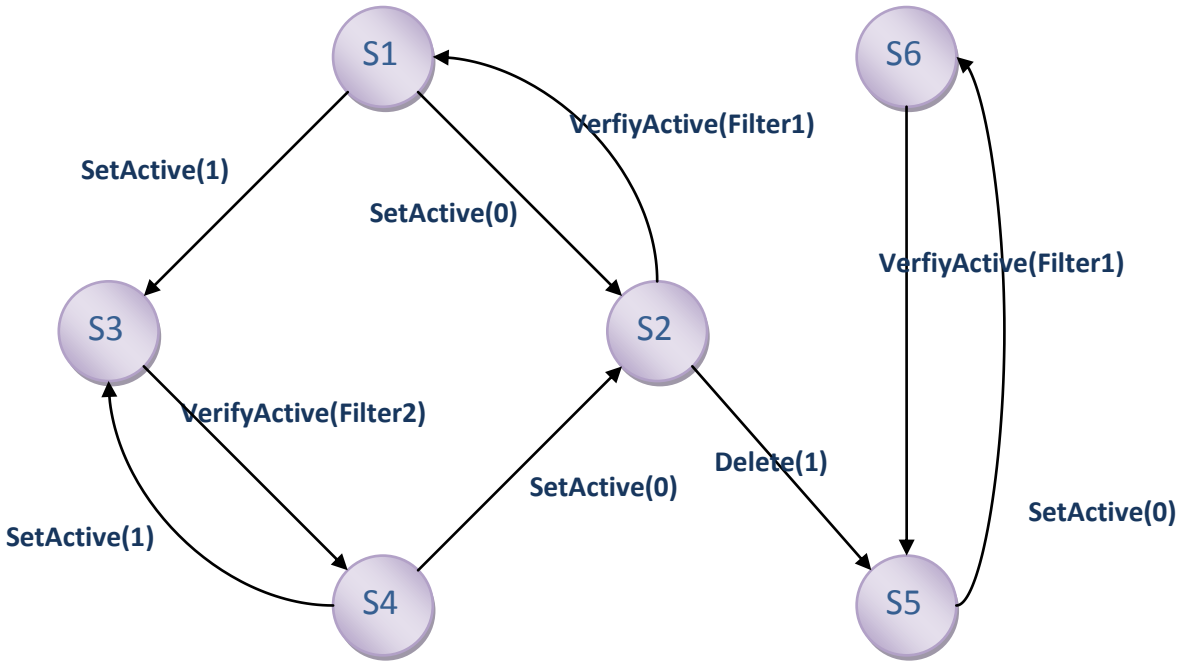

Figure 17: FSM for base dialog with a default profile made custom

3.2.2 Child UI dialog

The function of the child dialog is to provide properties of a single profile. A user may use this to:

1. Create a new blank profile,
2. Extend an existing profile to include more parsers, and
3. Edit a custom profile.

In addition to these, the user may also use other trivial functions, such as changing the sequence of folder paths in a profile, Open a folder path, view the paths in a profile, etc.

The basic state of this dialog comprises of

1. Mode: (New blank/Extend/Edit)
2. Name
3. Description
4. Folder paths in the current list.

This dialog might be launched from the base dialog for any of these modes. For the mode, a domain of two values would suffice, since the extend option would only be active if a non-custom profile is selected in the base dialog at the time of launch. Including that in the base state and combining these together is discussion in the next section.

The name and description are of no consequence to the model. The only condition is that the name field may not be empty if the OK button is clicked, so we include whether or not *something* is written in the space.

The basic actions a user might perform on this dialog:

1. Add/Edit/Delete a folder path in the list
2. Write/edit a name or description
3. Hit OK or Cancel.

We define the state for the Child dialog as follows:

```
Struct ChildState
{
      Boolean IsModeEdit,
      Boolean IsNameAndDescriptionWritten,
      List<Paths> PathsInList
}

ChildState _ChildState;
```

The initialization of _ChildState would be done by the base dialog actions. Now we define the Child dialog's actions:

```
SelectANewPath(Paths Path)
{
      _ChildState.PathsInList.Add(Path);
}

EditAPath(Integer PathIndex, Paths EditedPath)
{
      Precondition (PathIndex >= 0
                        AND PathIndex < _ChildState.PathsInList.Count );

      _ChildState.PathsInList[PathIndex]        <-    EditedPath;
}

DeleteAPath(Integer PathIndex)
{
      Precondition ( _ChildState.PathsInList.Length > 0 );
      Precondition (PathIndex >= 0
                        AND PathIndex < _ChildState.PathsInList.Count );

      _ChildState.PathsInList.RemoveAtIndex(PathIndex);
}

WriteNameAndDescription(String Name, String Description)
{
      _ChildState.IsNameAndDescriptionWritten   <- true;
}

HitOK()
{
      Precondition ( _ChildState.IsNameAndDescriptionWritten = true );
      Precondition ( _ChildState.PathsInList.Length > 0 );
}

HitCancel()
{

}
```

Even with the smallest domain, the above state machine would be too big to represent on this page. In order to produce meaningful test cases, we would need to slice it to be focused to a particular scenario from the test specification.

The parameter for WriteName would only be a symbolic value when SpecExplorer explores this machine, since a concrete initialization is not needed. A few starting states are shown below:

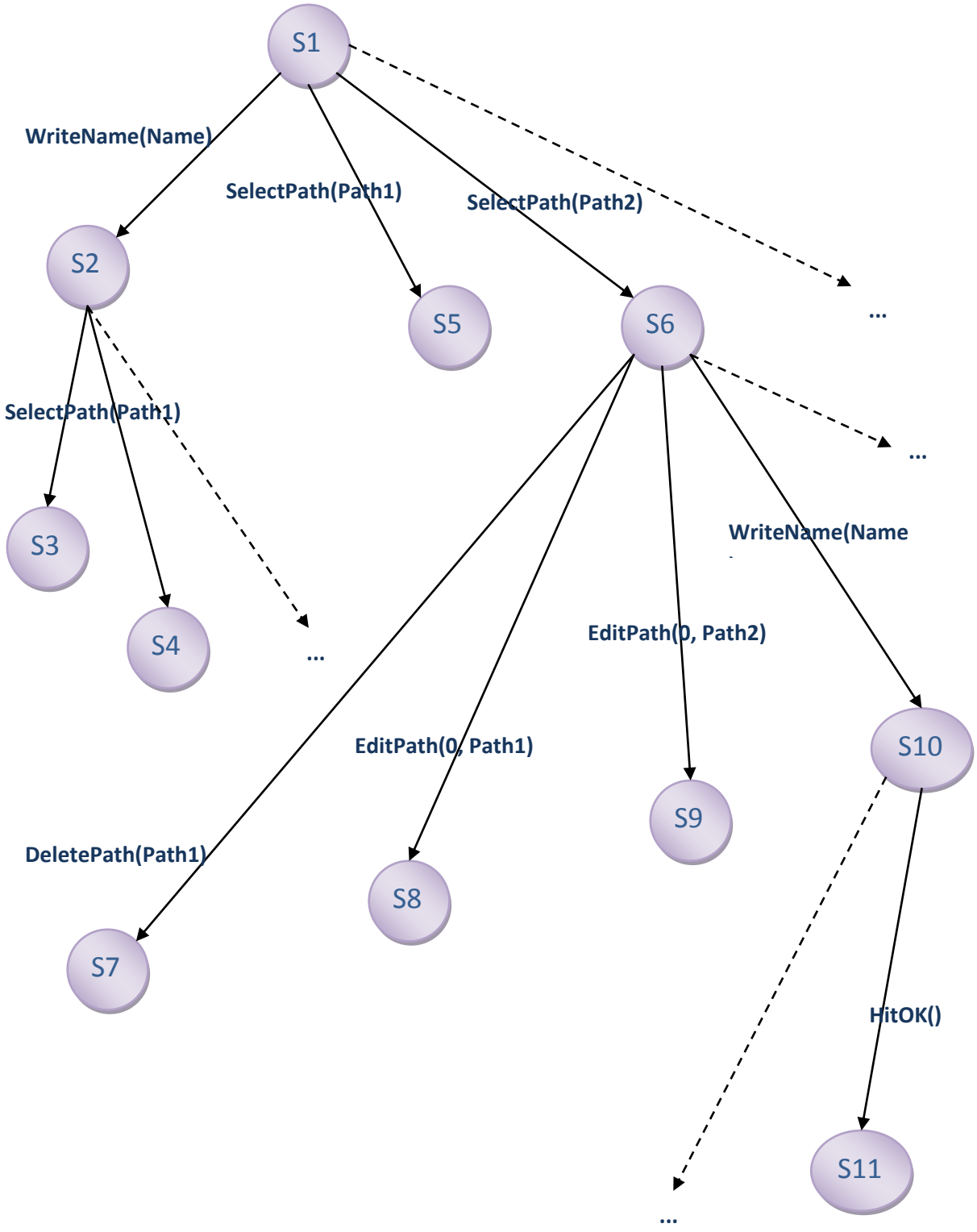

Figure 18: part of FSM for the child dialog

3.2.3 Combining the two

To combine these two together, first we include a flag which says which dialog is currently active, and initialize it with the base state:

```
Boolean IsChildWindowActive;
IsChildWindowActive = false;
```

For this flag, each of the actions in the Base and Child dialog now would have a precondition which checks if the corresponding window is active:

```
SetAProfileAsActive(Integer ProfileIndex)
{
      Precondition ( IsChildWindowActive = false );
      Precondition ( _BaseState.IsVerificationActive = false);
      ...
}

DeleteAProfile(Integer ProfileIndex)
{
      Precondition ( IsChildWindowActive = false );
      Precondition ( _BaseState.IsVerificationActive = false );
      ...
}

...

SelectANewPath(Paths Path)
{
      Precondition ( IsChildWindowActive = true );
      _ChildState.PathsInList.Add(Path);
}

...
```

This flag can now be used to fire the child dialog from the base dialog. We include new launch actions in the base dialog:

1. Create a new blank profile
2. Extend a profile
3. Edit a profile

We also add an EditProfileIndex to the base state, to keep track of a profile being edited.

```
Integer EditProfileIndex;
EditProfileIndex = -1;
```

```
CreateNewBlankProfile()
{
      Precondition ( IsChildWindowActive = false );

      _ChildState.IsModeEdit                    <-    false;
      _ChildState.IsNameAndDescriptionWritten   <-    false;
      _ChildState.PathsInList                   <-    Empty_list

      IsChildWindowActive                       <-    true;
}

ExtendAProfile(Integer ProfileIndex)
{
      Precondition ( IsChildWindowActive = false );

      _ChildState.IsModeEdit                    <-    false;
      _ChildState.IsNameAndDescriptionWritten   <-    true;
      _ChildState.PathsInList                   <-
            _BaseState.Profiles[ProfileIndex].PathList

      IsChildWindowActive                       <-    true;
}

EditAProfile(Integer ProfileIndex)
{
      Precondition ( IsChildWindowActive = false );
      Precondition ( _BaseState.Profiles[ProfileIndex].IsCustom = true );

      _ChildState.IsModeEdit                    <-    true;
      _ChildState.IsNameAndDescriptionWritten   <-    true;
      _ChildState.PathsInList                   <-
            _BaseState.Profiles[ProfileIndex].PathList

      EditProfileIndex                          <-    ProfileIndex
      IsChildWindowActive                       <-    true;
}
```

On the child dialog side, we modify the HitOK and the HitCancel buttons to reflect changes back to the base dialog:

```
HitOK()
{
	Precondition ( IsChildWindowActive = true );
	Precondition ( _ChildState.IsNameAndDescriptionWritten = true );
	Precondition ( _ChildState.PathsInList.Length > 0 );

	Profile NewProfile;
	NewProfile.IsCustom	<-	true;
	NewProfile.PathList	<-	_ChildState.PathsInList

	If ( _ChildState.IsModeEdit = true )
	{
		_BaseState.Profiles[EditProfileIndex]	<-	NewProfile;
	}
	Else
	{
		_BaseState.Profiles.Add ( NewProfile );
	}

	IsChildWindowActive	<-	true;
}

HitCancel()
{
	IsChildWindowActive	<-	true;
}
```

The machine for this model would have too many states, and exhaustive testing is not the purpose of this exercise. In order to produce a meaningful set of test cases, we slice the model by writing a sequence of actions and constraining some parameters (An underscore '_' indicates SpecExplorer to expand all values):

ModelSlice:

```
{       CreateNewBlankProfile ();

        SelectAPath ({Path1, Path2});

        WriteNameAndDescription (_, _);

        HitOK ();

        SetProfileAsActive (_);

        VerifyActiveProfile (_);        }
```

This scenario would limit the machine to taking the above steps:

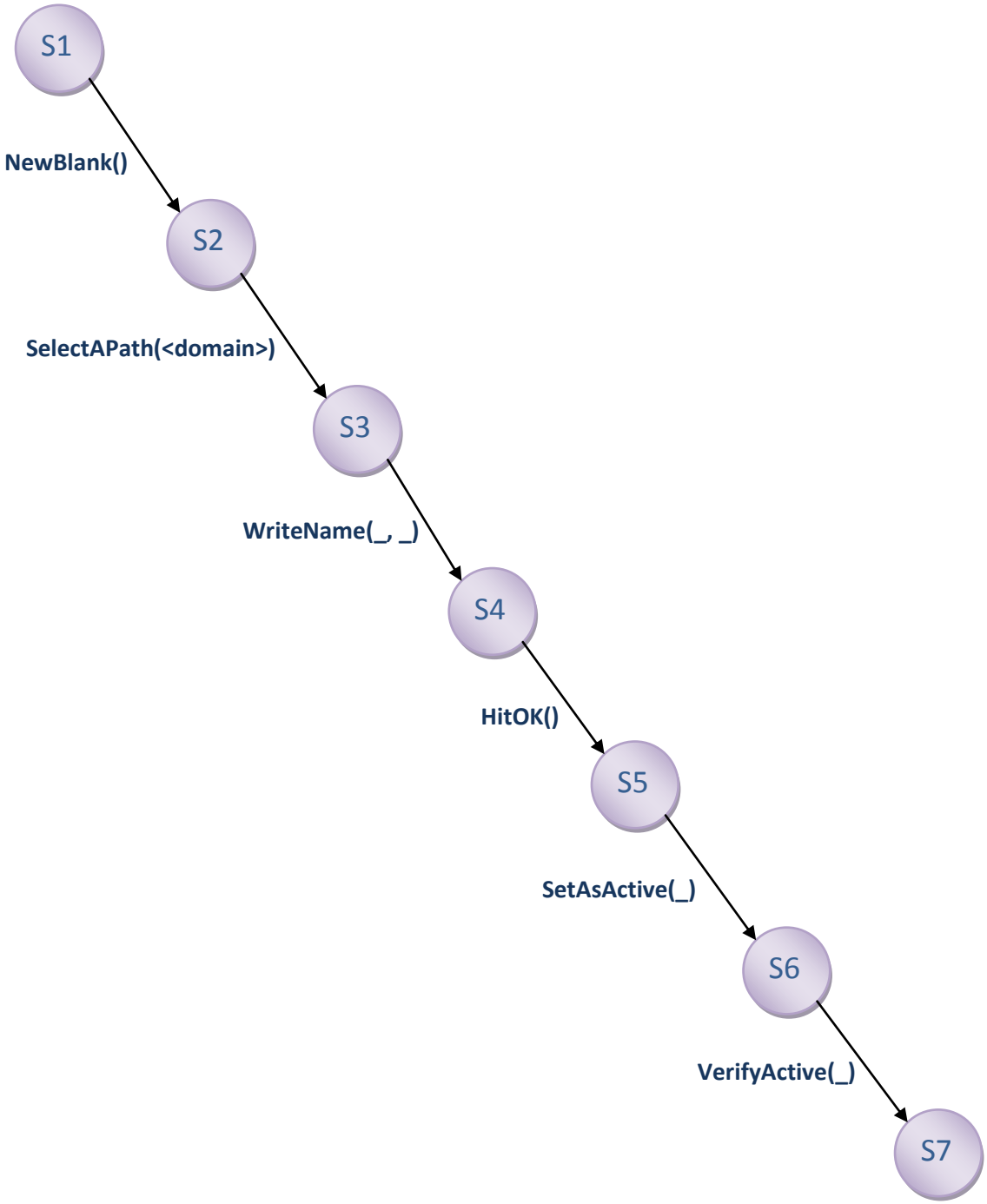

Figure 19: Scenario machine for creating and applying a new profile

When we merge this sliced machine with the full state machine, if we initialize with only a single profile, two Paths and concrete name and description, the following machine is obtained:

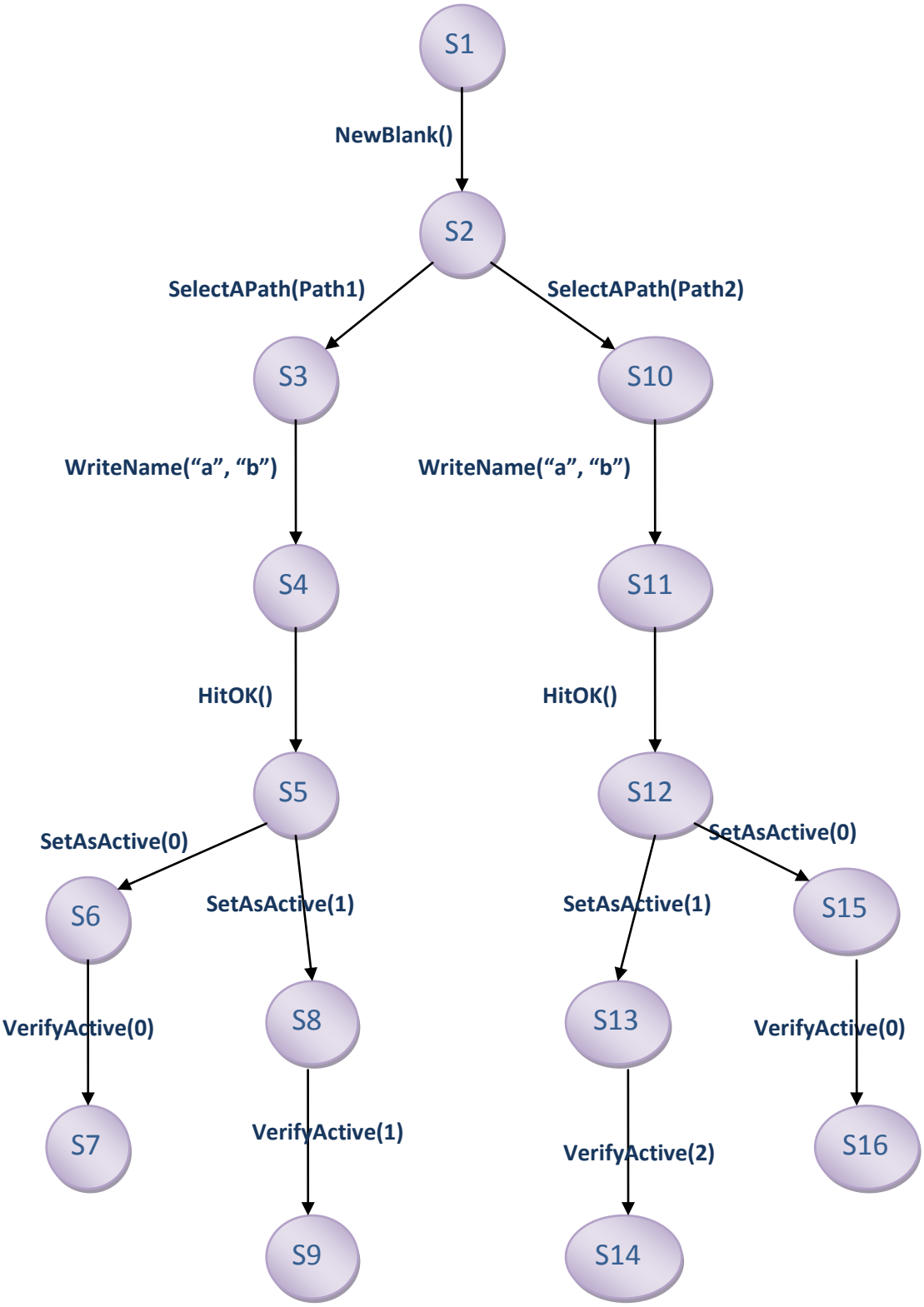

Figure 20: Limited composition of scenario with the full model

The values for the parameter in the verify method is the filter choice. While the model makes the decision which filter would be suitable, the exploration is not affected.

This machine may now be used to produce four distinct test cases, which walk the four paths from S1 to S7, S9, S14 and S16.

Starting with this model, we wrote additional actions and broke up the basic actions into trivial actions to correspond to actions in the User Interface (such as Open the settings window, click a particular button, or select an item from the list); and implementation code for the automation of such actions. Excerpts from the code are given in the next chapter.

Chapter 4: Implementation

This chapter produces code snippets from the implementation of the approach outlined in the previous chapter. The implementation consists of

- Model code
- Exploration of machines and scenarios
- Implementation (Adapter) code
- Test harness and execution

4.1 Model

The essential states and relevant actions are produced below:

State definitions:

Profile:

```
public struct Profile
{
    public bool Custom;
    public SequenceContainer<Paths> Paths;
}
```

Enumerated Paths:

```
public enum Paths
{
    ProfileDirectory,
    MySParserDirectory,
    MicrosoftParsers,
    Core,
    Common,
    Windows,
    CommonStubs,
    WindowsStubs
}
```

Filters:

```
public enum Filters
{
    NonCompilable,
    PureCapture,
    Default,
    Faster,
    MicrosoftWindows
    :
```

The base dialog state:

```
public struct BaseState
{
    public SequenceContainer<Profile> Profiles;
    public int ProfileSelected;
    public int ProfileActive;
    public bool ViewProfileWindowActive;
    public bool SetAsActiveCheckActive;
    public Filters CurrentFilterNumber;
}
```

The child dialog state:

```
public struct CreateNewState
{
    public bool Active;
    public bool Edit;
    public SequenceContainer<Paths> PathsInList;
    public bool NameAndDescriptionWritten;
    public int PathSelected;
    public bool ErrorDialogActive;
}
```

Declaring the state variables:

```
static bool Initiated = false;
static BaseState _BaseState;
static CreateNewState _CreateNewState;

static Profile PureCapture, Default, FasterParsing, ...
```

The implementation includes an action called initiate, which has to run before anything else runs. This uses helper methods which initialize all the state variables. For this purpose, the state variable Initiated is included.

```
[Action("Initiate()")]
static void Initiate()
{
    Contracts.Requires(!Initiated);

    initializeCreateNewState();
    initializeBaseState();

    Initiated = true;
}
```

Profile selection:

```
[Action("SelectAProfile(ProfileIndex)/result")]
static int SelectAProfile(int ProfileIndex)
{
    Contracts.Requires(Initiated);
    Contracts.Requires((ProfileIndex > -1));
    Contracts.Requires(ProfileIndex < _BaseState.Profiles.Count);
    Combination.Expand(ProfileIndex);

    Contracts.Requires(!_CreateNewState.Active);
    Contracts.Requires(!_BaseState.ViewProfileWindowActive);
    Contracts.Requires(!_BaseState.SetAsActiveCheckActive);
```

```
            _BaseState.ProfileSelected = ProfileIndex;
            return _BaseState.ProfileSelected;
        }
```

The return value in the above method is used only in order to display this value in the visualization. This is specified by '/result' in the action attribute.

Extending a profile:

```
        [Action("NewProfileFromSelected()")]
        static void NewProfileFromSelected()
        {
            Contracts.Requires(Initiated);
            Contracts.Requires(_BaseState.ProfileSelected != -1);
            Contracts.Requires(!_CreateNewState.Active);
            Contracts.Requires(!_BaseState.ViewProfileWindowActive);
            Contracts.Requires(!_BaseState.SetAsActiveCheckActive);

            initializeCreateNewState();

            _CreateNewState.Active = true;
            _CreateNewState.NameAndDescriptionWritten = true;
            _CreateNewState.PathsInList =
clonePathSequence(_BaseState.Profiles[_BaseState.ProfileSelected].Paths);
            _CreateNewState.PathsInList.Insert(1, Paths.MySParserDirectory);

        }
```

Method SetAsActive, if its state requirements are met, sets the current filter and triggers the CheckSetAsActive() method to fire. If the profile is successfully compiled, it changes the currently active profile.

```
        [Action("SetAsActive()")]
        static void SetAsActive()
        {
            ...

            Contracts.Requires(!_BaseState.SetAsActiveCheckActive);

            _BaseState.CurrentFilterNumber =
PathSequenceFilterNumber(_BaseState.Profiles[_BaseState.ProfileSelected].Path
s);

            if (!(_BaseState.CurrentFilterNumber == Filters.NonCompilable))
            {
                _BaseState.ProfileActive = _BaseState.ProfileSelected;
            }

            _BaseState.SetAsActiveCheckActive = true;
            ...
        }
```

When the user hits OK on the 'Create a blank profile' dialog, it checks if the name is written, and if there is at least one path in the list. If these conditions are not met, it activates an error dialog. Otherwise it performs the necessary assignments and de-activates the child dialog:

```
        [Action("HitCreateNewOK(Edit)")]
        static void HitCreateNewOK(bool Edit)
        {
            ...

            if ((!_CreateNewState.NameAndDescriptionWritten) ||
(_CreateNewState.PathsInList.Count == 0))
            {
                _CreateNewState.ErrorDialogActive = true;
            }
            else if (_CreateNewState.Edit)
            {
```

```
                Profile NewProfile = new Profile();
                NewProfile.Custom = true;       //Any non custom profile is
not editable
                NewProfile.Paths =
clonePathSequence(_CreateNewState.PathsInList);
                _BaseState.Profiles[_BaseState.ProfileSelected] = NewProfile;
                _CreateNewState.Active = false;
            }
            else if (!_CreateNewState.Edit)
            {
                Profile profile;
                profile.Paths =
clonePathSequence(_CreateNewState.PathsInList);
                profile.Custom = true;

                //Any new profile always shows up at the second place, and
since there are 4 read-only profiles, this should never be out-of-bounds
                _BaseState.ProfileSelected = 1;
                _BaseState.Profiles.Insert(_BaseState.ProfileSelected,
profile);
                _CreateNewState.Active = false;
            }

            ...
        }
```

4.2 Exploration of machines and scenarios

If the model is explored in its entirety, it produces a state explosion which easily hits a very high state bound. A portion of the exploration is produced here.

```
config Main
{
    bound steps = 1024;
    bound pathdepth = 1024;

    switch testclassbase = "vs";
```

```
    switch generatedtestpath = "..\\TestSuite";
    switch generatedtestnamespace = "SpecExplorerProject.TestSuite";

    action static void Implementation.Initiate();
    action static void Implementation.NewBlankProfile();
      …

}

machine Profiles() : Main
{
    construct model program from Main where namespace =
"SpecExplorerProject.Model"
}
```

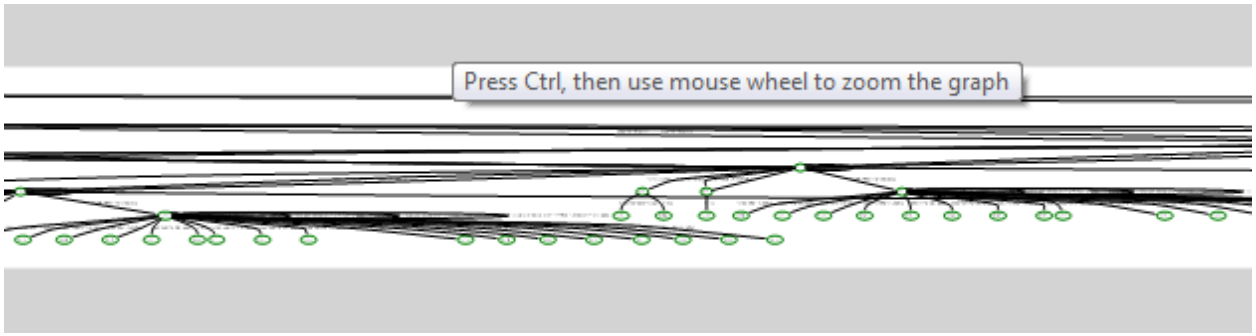

Figure 21: Full model exploration

The following sequence is written in order to satisfy a test scenario: 'A user should be able to set any default profile as active'.

```
machine ScenarioSetDefaultProfilesAsActive() : Main
{
      Initiate(); SelectAProfile(_); SetAsActive(); CheckSetAsActive(_);
      End();
}
```

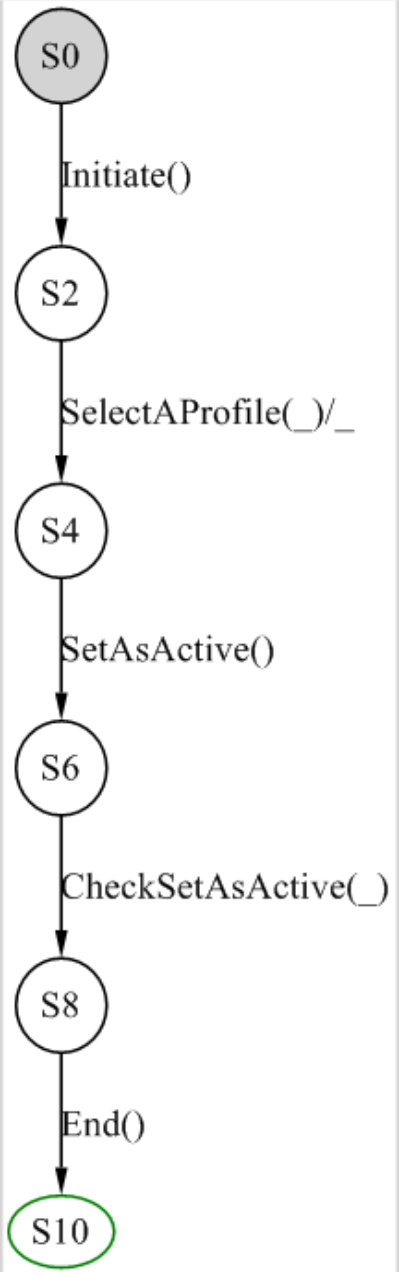

Figure 22: Scenario for applying a default profile

The above machine is intersected with the full model to produce a finite exploration:

```
machine SlicedSDPAA() : Main
{
      ScenarioSetDefaultProfilesAsActive() || Profiles
}
```

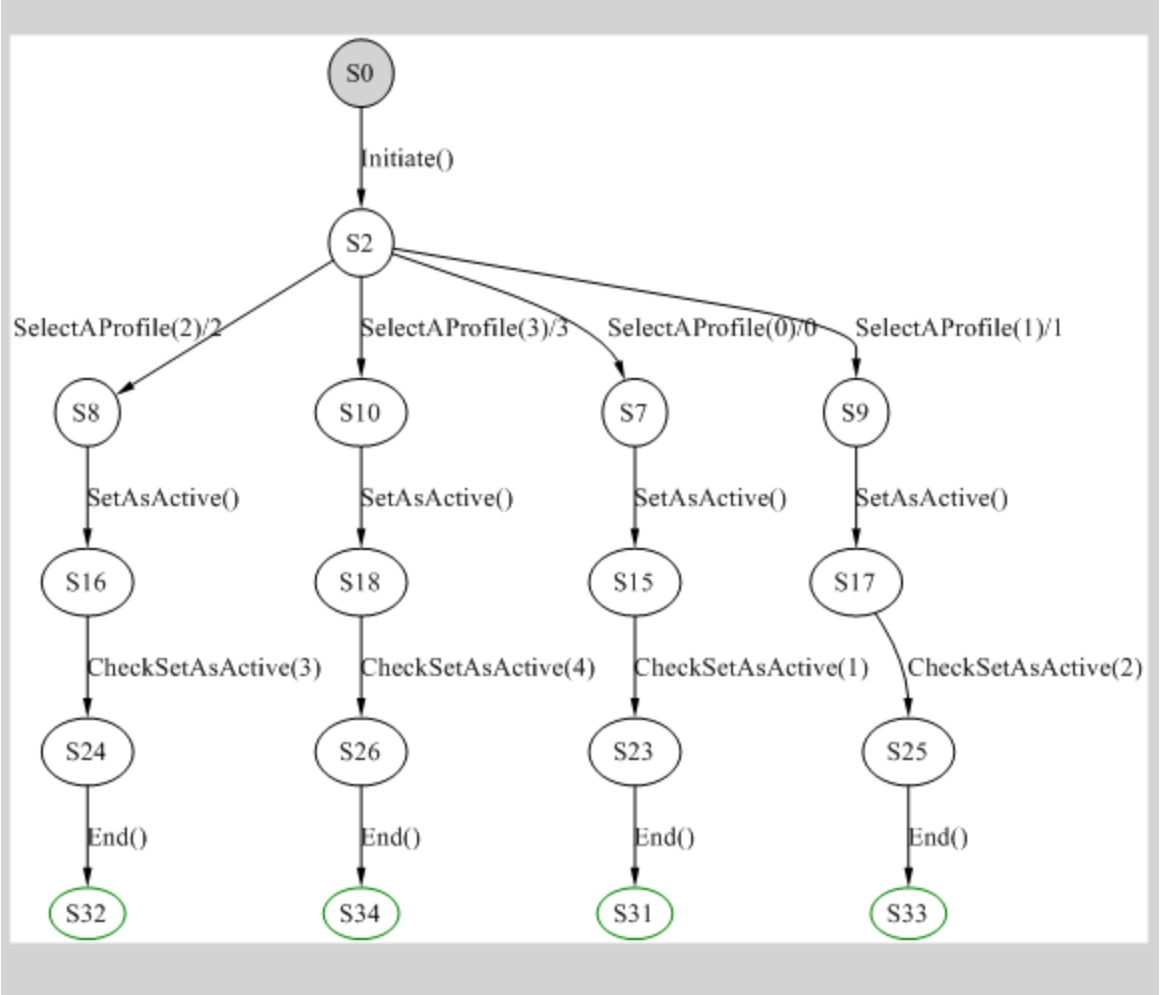

Figure 23: Composition of the scenario with the model

And finally, a machine is written to explore test cases from this model:

```
machine TestCasesSDPAA() : Main
{
      construct test cases for SlicedSDPAA
}
```

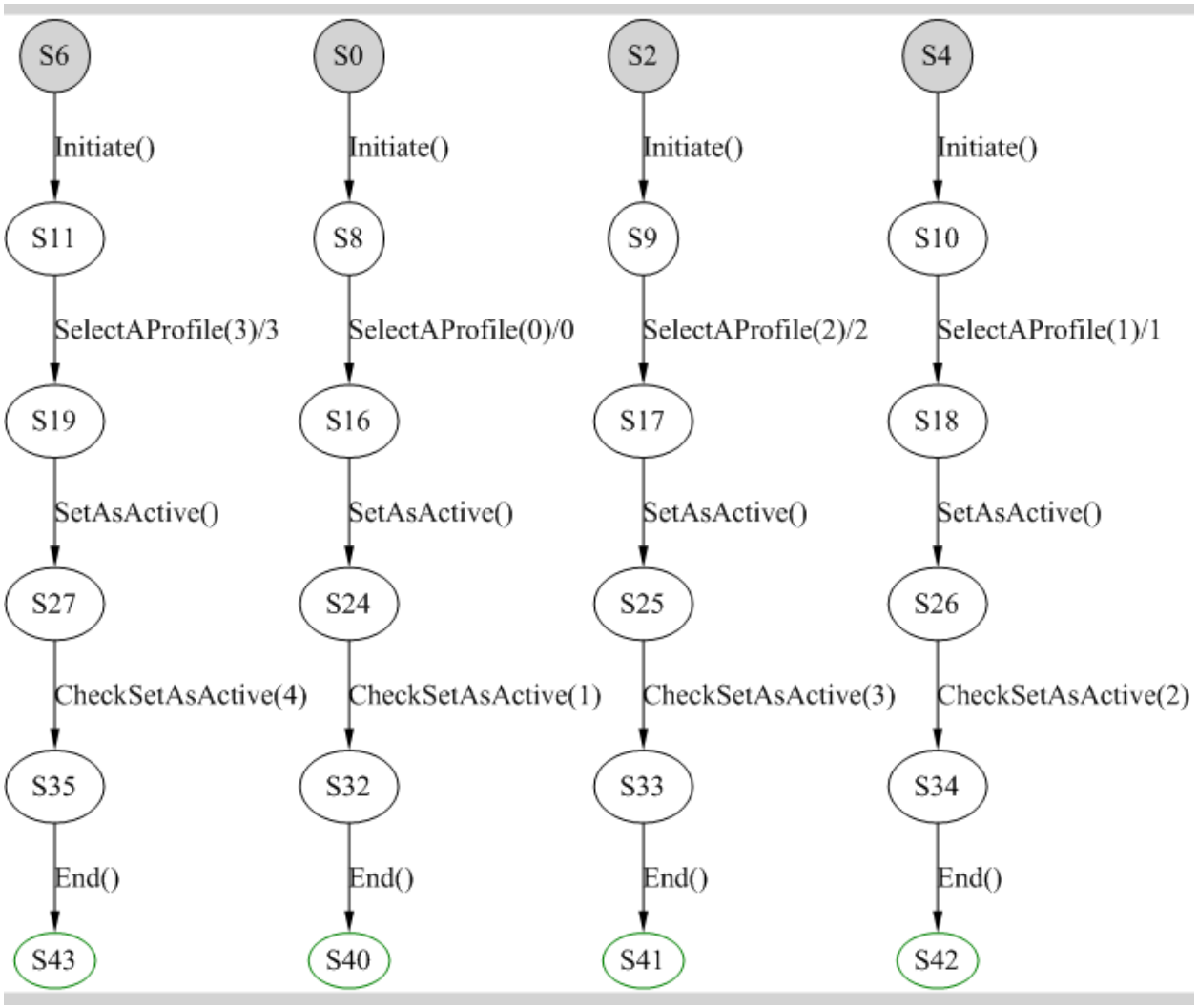

Figure 24: Generating test cases for the scenario

Another scenario is shown here as an example:

```
machine ScenarioEditAProfile() : Main
{
      Initiate();                   SelectAProfile({1, 2, 3});
      NewProfileFromSelected();     HitCreateNewOK(_);
      ViewAProfile();               SelectAPath(1, _);
      DeleteAPath(_);               SelectNewPath(_, _);
      SelectNewPath(1, _);          HitCreateNewOK(_);
      SetAsActive();                CheckSetAsActive(_);
      SelectAProfile(0);            SetAsActive();
      CheckSetAsActive(_);          SelectAProfile(1);
      DeleteAProfile();             End();                  }
```

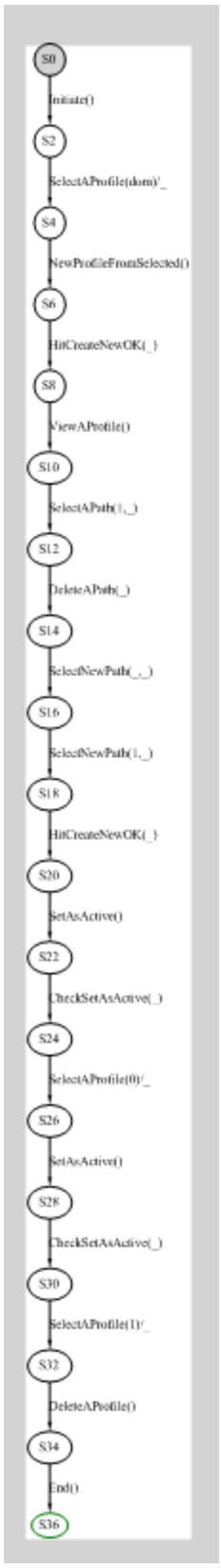

Figure 25: Scenario machine for editing a profile

A portion of exploration of the above machine intersected with the full model:

```
machine SlicedEAP() : Main
{
      ScenarioEditAProfile() || Profiles
}
```

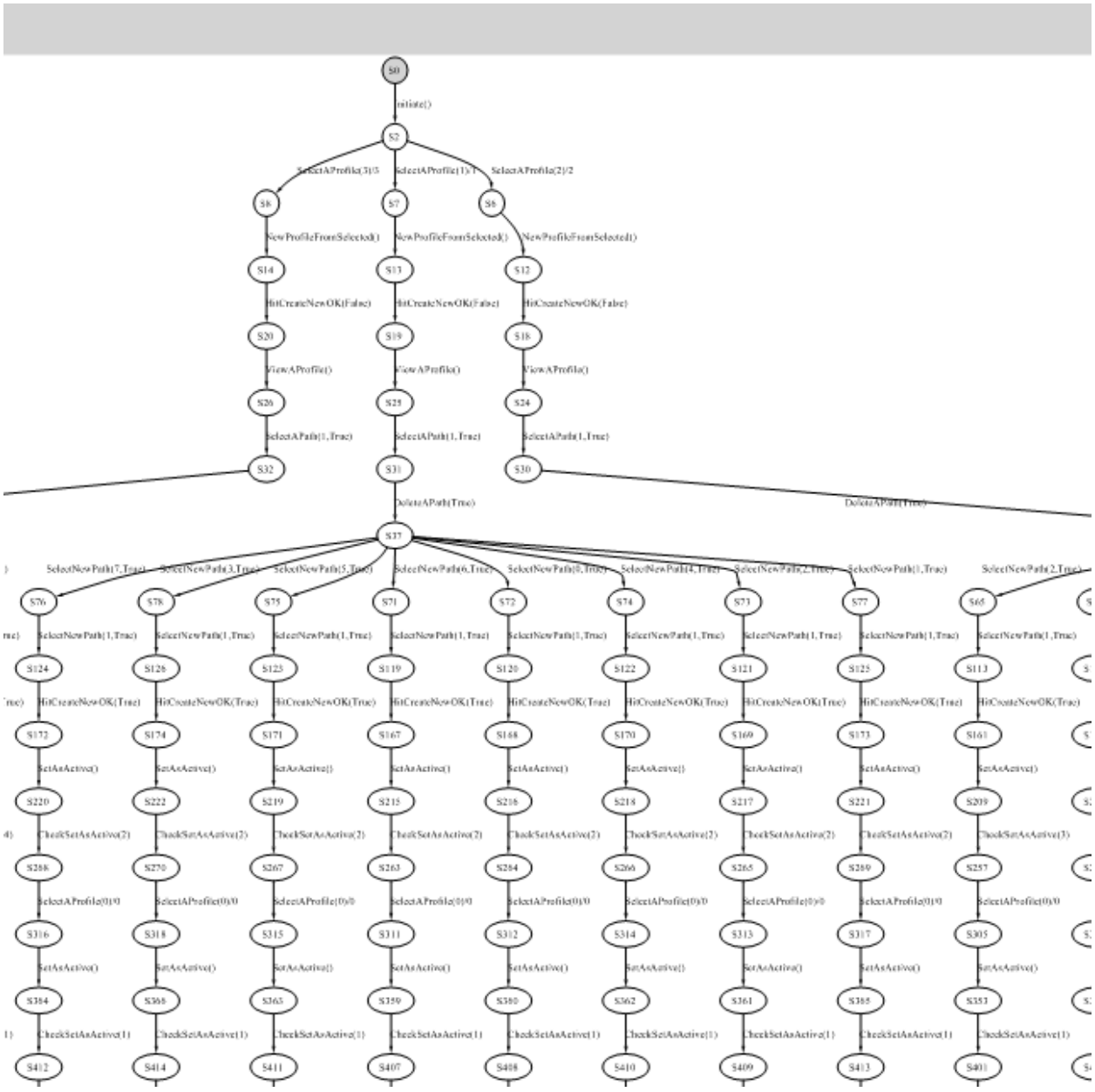

Figure 26: Part of exploration of edit a profile scenario

And some test cases

```
machine TestCasesEAP() : Main
{
      construct test cases for SlicedEAP
}
```

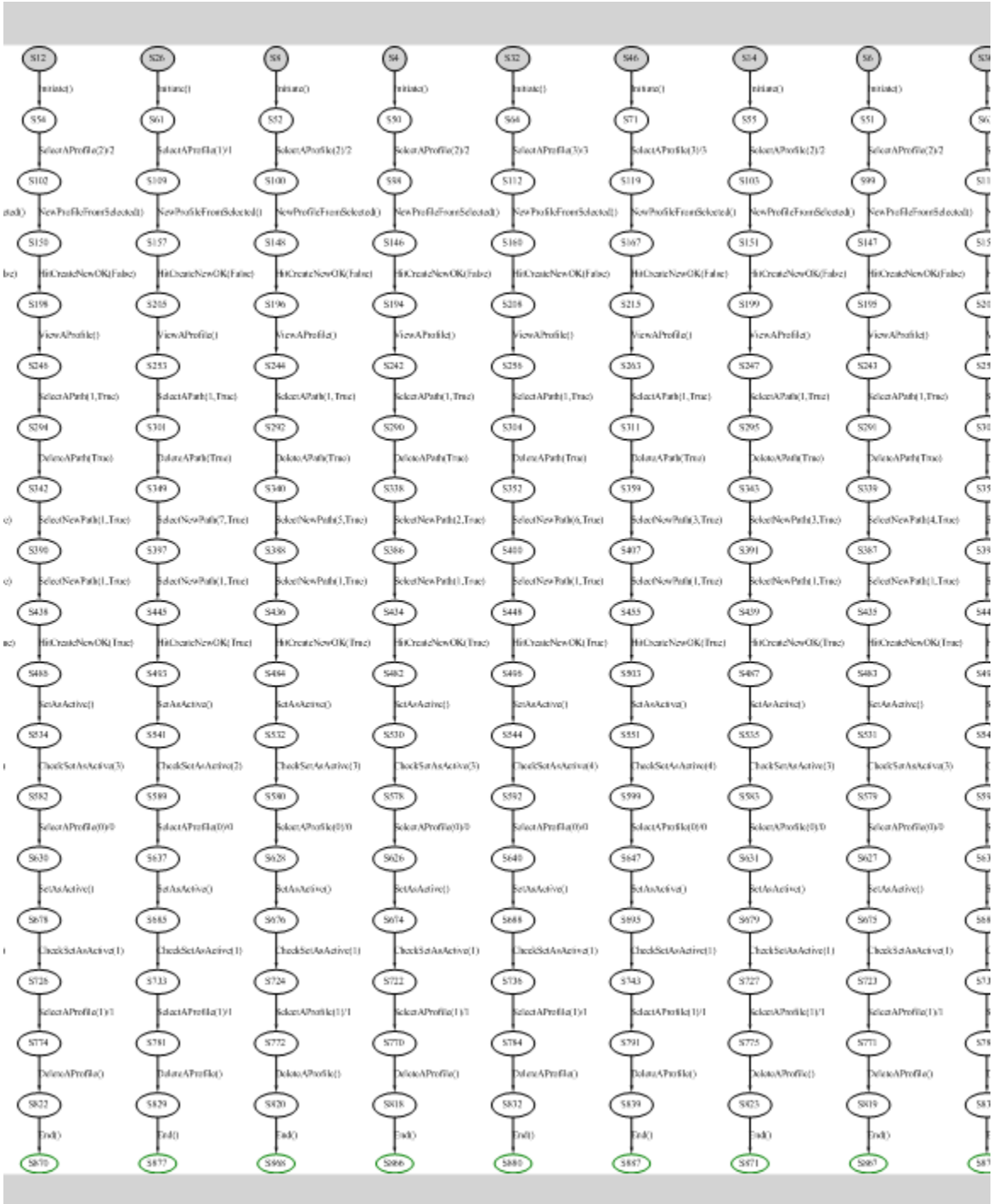

Figure 27: Part of test cases for edit a profile scenario

4.3 Implementation (Adapter) code

For the methods and states in the model, we wrote the corresponding implementation. For the state, the variables get concrete values from a configuration (e.g. Filter expressions, folder paths). The actions produce a set of UI automation steps in order to achieve the intended actions of the model. The parameters to the implementation methods get their values assigned from the model exploration, and the corrosponding concrete values are incorporated.

An Initiate() method initiates this process and an End() method combines all the steps together to form a sequnce and writes it to a file which may be used as input for the UI automation tool.

4.4 Test harness and execution

We wrote a separate test harness to execute the test cases produced from the implementation. Each test case, when executed, produces an input file, which is picked up and run against the User Interface. This harness also compares the resulting capture files that are saved with the existing baseline files and logs the results.

While it is possible to run the UI automation in the End() method mentioned in the section above, we found it easier to deploy the test case code as the previously written test cases worked in a similar manner. This test harness could be incorporated into the existing test framework and run with no special configuration.

Chapter 5: Results

5.1 Results

This project generated a total of 103 short test cases across six test scenarios, which covered all the UI test requirements. Test cases produced with the previous method would have been longer and much fewer, but the model generated cases provide much better coverage for each scenario by exercising many possible combinations.

Also, the test case generation with this model is much more flexible than the existing approach. Writing a new scenario with a different sequence of actions or parameter values is as easy as writing another cord machine. A particular corner case to verify a bug fix may also be generated in a similar way.

If a related feature is modeled in the future, the two models can be composed to create test cases for their interaction.

5.2 Limitations

While generating tests from a model certainly has a lot of advantages, we found the process inhibiting in two cases:

1. If the system is fairly simple (i.e. relatively small number of states); such as stress testing, and needs to be tested with a few input combinations, the cost of developing a model-based solution may not be effective (given the abstraction, design and the learning curve).
2. If the decision logic is so complicated that it cannot easily be abstracted, one may end up writing a model nearly as big and complicated as the real system.

5.3 Future Enhancements

This study implements a proof-of-concept for applying model-based testing to the Network Monitor user interface. A lot more can be gained by applying this technology to future features and past where coverage with existing test cases is not sufficient, especially where:

- Multiple simultaneous instances are used; SpecExplorer's instance-based modeling would provide a solution.

- Two or more components are acting together, i.e. where they share some actions and parts of the state of the whole system. SpecExplorer provides a good way to make composite models; each component may be modeled individually and composed together. Such combinations would provide much better coverage (especially where the components interact) and reduce the development time.

- Exploit the parameter combination techniques in SpecExplorer (e.g. Pairwise) to provide better coverage to parameter-intensive components with a small number of test cases.

- Exploit test strategies available with SpecExplorer to create more efficient test cases.

5.4 Conclusion

This study presented the usefulness and the cost/benefit analysis of using Model-Based testing approach against the Network Monitor user interface, and also gave us insight into using SpecExplorer effectively for other test areas in the test team.

Bibliography

Jacky, J., Veanes, M., Campbell, C., & Schulte, W. (2008). *Model-Based Software Testing and Analysis with C#.* Cambridge University Press.

Microsoft. (2010). *Cord Syntax Definition*. Retrieved 5 6, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ee691953(v=SpecExplorer.10).aspx

Microsoft. (2010). *Spec Explorer*. Retrieved 5 6, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ee620411(v=SpecExplorer.10).aspx

Nico Kicillof. (2009, 10 27). *SpecExplorer team blog*. Retrieved 11 2009, from MSDN Blogs: http://blogs.msdn.com/specexplorer

SpecExplorer. (n.d.). Retrieved October 28, 2009, from MSDN: http://msdn.microsoft.com/en-us/devlabs/ee692301

Utting, M., & Legeard, B. (2007). *Practical Model-Based Testing - A tools approach.* Morgan Kaufmann Publishers.