Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Spec Explorer

A Model-Based Testing tool from Microsoft

Seminararbeit

Manuel Naujoks

Betreut durch

Prof. Dr. Thomas Fuchß

Bischweier, den 10. Dezember 2011

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Bischweier, den 10. Dezember 2011
Manuel Naujoks

Inhaltsverzeichnis

1 Einführung

In dieser Seminararbeit soll das Programm *Spec Explorer* vorgestellt werden. Dabei handelt es sich um eine Anwendung die modellbasiertes Testen unterstützt. *Spec Explorer* ist seit 2010 ein Visual Studio Power Tool [DuW10] und integriert sich als Erweiterung in die Entwicklungsumgebung von Microsoft ab Version 2010 Professional. Über die Visual Studio Gallery kann das Tool installiert werden [Mic10].

In diesem Einführungs-Kapitel sollen die Grundlagen vermittelt werden, die für ein Verständnis des *Spec Explorer* Add-Ins erforderlich sind. Dazu wird in Abschnitt 1.1 zunächst der Begriff des *Testen* erläutert, bevor in Abschnitt 1.2 auf das wissenschaftliche Konzept hinter dem Tool eingegangen wird. *Spec Explorer* implementiert eine Weiterentwicklung einer als *Abstract State Machine* bekannten Notation von zustandsbehafteten Systemen. In Abschnitt 1.3 wird anschließend beschrieben, wie das Konzept der abstrakten Zustandsmaschine Einfluss auf die Entwicklung der Visual Studio Erweiterung hatte. Dabei werden mehrere Technologien kurz vorgestellt, die als Zwischenschritte auf dem Weg von *Abstract State Machines* (ASM) bis Spec Exploror gesehen werden können.

1.1 Testen

Traditionelles Testen von Software wird oft manuell vorgenommen. Das bedeutet das ein Mensch das Programm bedient und sämtliche Szenarien und Testfälle ausprobiert und mit einem erwarteten Ergebnis vergleicht. Da Software im Laufe der Zeit immer komplexer geworden ist und die realisierten Szenarien pro Anwendung zugenommen haben, wurden Tests zunehmend automatisiert. Testfälle müssen so nur noch einmal spezifiziert werden und können dann automatisiert durchgeführt werden. Ein Softwaresystem kann auf verschiedenen Ebenen getestet werden. Bisher wurde der Begriff des Testen verallgemeinert verwendet. Die drei bekanntesten Test-Ebenen werden im folgenden kurz vorgestellt, wie sie auch in [Com11] beschrieben sind.

1. Eine Ebene für Tests stellt die komplette Anwendung dar, in der Szenarien wie von einem Nutzer ausgeführt getestet werden können.

2. Eine weitere Ebene ist eine logisch zusammengehörende Einheit von Programmcode (Unit). Hier werden die einzelnen Bausteine des komplexen Systems isoliert voneinander getestet.

3. Eine dritte Ebene fast mehrere isolierte Bausteine zusammen und testet deren Zusammenarbeit als Szenario.

Testfälle der ersten Ebene werden meistens von den Benutzern des Systems definiert, um die allgemeine Funktionsfähigkeit festzustellen. Tests der zweiten Ebene werden meistens nur von Entwicklern definiert um die Korrektheit des Programmcode sicherzustellen. Tests der dritten Ebene werden meistens von Entwicklern, Projektleitern und Benutzern definiert.

Nachdem unterschiedliche Arten des Testen kurz vorgestellt wurden bleibt bei jeder Art eine Schwierigkeit. Testfälle müssen definiert werden und zwar so, dass sie die Anwendung möglichst ausführlich getestet wird. Bei zunehmender Komplexität der Anwendung und zunehmender Agilität der Anwendungsentwicklung stellt dies eine Herausforderung für Entwickler dar, da diese nicht nur den Programmcode, sondern auch den Testcode schreiben müssen. *Spec Explorer* ist ein Werkzeug das Entwickler bei der Testfallerzeugung und Lösung dieses Problems unterstützen kann.
Was als wissenschaftliche Disziplin [Gur93] begann, findet mit Tools wie *Spec Explorer* Einzug in die Softwareentwicklung. Die zu testende Anwendung wird zunächst in einer abstrakten Form modelliert. Aus diesem Modell lassen sich anschließend Testfälle erzeugen, die auf den Programmcode der Anwendung angewendet werden können und die Anwendung so anhand des Modells validieren.

1.2 Abstract State Machines

Ein Computerprogramm ist ein Folge von Anweisungen die sequentiell ausgeführt werden. Eine solche Anweisung entspricht einer Zustandsänderung des Programms. Die Menge aller Zustandsänderungen repräsentiert die Funktionsweise des Systems. Yuri Gurevich hat 1999 [Gur99] gezeigt, dass ein solches System in einer von ihm erdachten Struktur namens *Abstract State Machine* abgebildet werden kann. Demnach kann jeder Algorithmus, egal wie abstrakt, so modelliert werden. Eine *Abstract State Machine*, 1993 unter der Bezeichnung *Evolving Algebra* vorgestellt [Gur93], ist ein Zustandsautomat, der Zustände nicht als explizite Objekte betrachtet. Laut Gurevich sind Zustände ambient im System vorhanden und entstehen implizit durch Veränderung von veränderlichen Eigenschaften des Systems. Auch die Zustände in einem Computerprogramm sind abstrakt und werden durch die Anweisungen des Programms verändert. Die gleichen Zustandsänderungen können durch von Gurevich als *update*-Anweisungen bezeichnete Ausdrücke vorgenommen werden. Gurevich's *Abstract State Machine* kann damit wie in Abbildung 1.1 als eine Menge von *update*-Anweisungen dargestellt werden, die jeweils von einer *guard*-Annweisung begleitet werden. Die Kombination von *guard*- und *update*-Annweisung kann auch als Regel bezeichnet werden. Die *guard*-Anweisung vergleicht den Zustand des Systems mit einem Erwartungswert und führt bei einer Übereinstimmung die zugehörige *update*-Anweisung aus. Die *Abstract State Machine* läuft die Folge immer wieder durch und stellt somit das Programm der Zustandsmaschine dar. Endzustände, Abbruchbedingungen und andere Notationen können ebenfalls definiert werden.

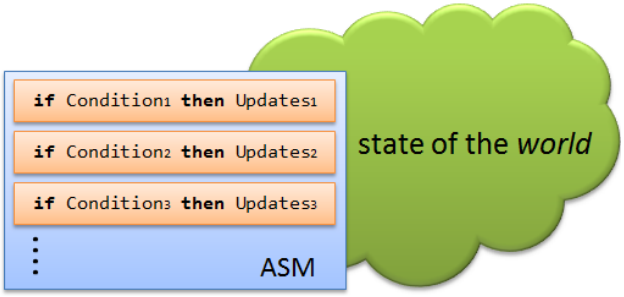

Abbildung 1.1: Schematische Darstellung einer *Abstract State Machine*

Das Konzept der *Abstract State Machine* wird auch durch Egon Börger in Form eines Tutorials [Bör] zusammengefasst. Ihm zufolge entspricht der aktuelle Zustand im System der aktuellen konkreten Ausprägung aller Variablen in diesem System. Die Menge aller Zustände ist somit durch die Wertebereiche aller Variablen im System definiert. Zusammen mit Robert Stärk hat Egon Börger 2003 seine Abhandlungen in dem sogenannten *AsmBook* zusammengefasst [EB03].

1.3 Von ASMs bis Spec Explorer

Nachdem der Begriff des Testen in Abschnitt 1.1 eingeführt und das Konzept von abstrakten Zuständen in Kapitel 1.2 erläutert wurde, beschäftigt sich dieses Kapitel mit dem Werdegang der Visual Studio Erweiterung *Spec Explorer*. Wolfgang Grieskamp beschreibt in seinem Blogeintrag [Gri09] die Entwicklung von der theoretischen Idee der *Abstract State Machine* bis zu einem allgemeinen Programm das von Entwicklern genutzt werden kann.

1.3.1 AsmL

Nachdem Gurevich 1999 die theoretische Grundlage gelegt [Gur99] hat, wurde die erste Implementierung einer Beschreibungssprache für *Abstract State Machines* von einem Team in Microsoft vorgenommen. Diese Sprache wurde unter der Bezeichnung *AsmL* (Abstract State Machine Language) bekannt. Laut Grieskamp [Gri09] waren neben ihm selbst auch Yuri Gurevich, Margus Veanes, Wolfram Schulte, Lev Nachmanson, Colin Campbell und Nikolai Tillmann an der Entwicklung der Sprache beteiligt. Das Projekt wurde nie vollständig veröffentlicht und war eher ein Forschungsprojekt. *AsmL* wurde nicht zum Testen, wie in Abschnitt 1.1 vorgestellt, entwickelt, sondern diente dem formalen Beweisen der Korrektheit von Softwaresystemen. Auf der Codeplex-Seite von *AsmL* [Mic03] kann der Quellcode eines Binärcompilers heruntergeladen werden.

1.3.2 AsmL-T

Ein erster Ansatz *AsmL* zum Testen von Softwaresystemen zu verwenden, wurde mit einer Erweiterung der Sprache unter dem Namen *AsmL-T* von Wolfgang Grieskamp und Nikolai Tillmann entwickelt [Gri09]. Eine wesentliche Ergänzung war die Funktion sämtliche Zustände des mit einer *Abstract State Machine* beschriebenen Softwaresystems explorativ zu ermitteln. Grieskamp nennt dieses Feature *state space exploration*. Damit war das erste modellbasierte Test-Werkzeug entwickelt.
AsmL-T wurde laut Grieskamp [Gri09] verwendet, um die *WCF* (Windows Communication Foundation) Implementierung zu testen. Angeblich wurde das Testen mithilfe von *AsmL-T* beliebter und bekannter als die *AsmL*-Sprache selbst.

1.3.3 Spec#

Aufgrund des abnehmenden Interesse an *AsmL* wurde laut Grieskamp [Gri09] eine neue Implementierung benötigt. Aus diesem Grund entstand *Spec#* als leichtgewichtige *AsmL*-Implementierung. *Spec#* nutzt die Syntax von C# und nutzte *Code Contracts*, wie sie von Matthew Podwysocki in seinem Blogpost [Pod08] beschrieben werden. Obwohl C#-Syntax verwendet wird, wird für *Spec#* ein spezieller Compiler benötigt. Wahrscheinlich hat sich auch *Spec#* aus diesum Grund

bis heute noch nicht durchgesetzt. Ob *Spec#* die *AsmL*-Implementierung in *AsmL-T* ersetzte, geht aus Grieskamps Blogpost [Gri09] leider nicht hervor. 2004 wurde *AsmL-T* dann Microsoft-intern unter dem Namen *Spec Explorer 2004* veröffentlicht.

1.3.4 PEX

Laut Grieskamp [Gri09] wurde das Team, das sich bei Microsoft mit der Entwicklung von *Spec Explorer 2004* beschäftigt, 2005 getrennt. Während sich ein kleiner Teil des ursprünglichen Teams weiterhin mit der Entwicklung von modellbasierten Test-Werkzeugen beschäftigt, wurde ein neues Team gegründet. Dieses neue Team entwickelte ein Programm, das zu testenden Programmcode analysiert und Eingabewerte ermittelt, die zu einer möglichst hohen Testabdeckung des zu testenden Programmcodes führen. Dieses Programm wurde inzwischen unter dem Namen *PEX* veröffentlicht.

2 Modellbasiertes Testen

Spec Explorer ist ein modellbasiertes Test-Werkzeug. Dieses Kapitel beschäftigt sich damit, was modellbasiertes Testen überhaupt bedeutet. Nico Kicillof beschreibt MBT (Model Based Testing) in seinem Blogppost [Kic09] als leichtgewichtige und formale Methode um ein Softwaresystem zu validieren. MBT ist formal, weil es auf einer formalen, also Maschinen lesbaren Spezifikation, also dem Modell der Software arbeitet, die getestet werden soll. Kicillof nennt eine solche Software *System Under Test*. Weiterhin ist MBT leichtgewichtig, da es im Gegensatz zu anderen formalen Methoden keinen mathematischen Beweis anstrebt um zu zeigen das die Implementierung der Spezifikation unter allen möglichen Umständen entspricht. MBT erzeugt aus einem Modell systematisch eine Menge von Testfällen, die auch *test suite* genannt wird. Diese Testfälle, wenn sie auf die Implementierung des Systems angewendet werden, bieten ausreichende Überzeugung, dass sich das *System Under Test* so verhält, wie sein Modell es definiert hat. Laut Kicillof [Kic09] ist der Unterschied zwischen leichtgewichtigen und schwergewichtigen formalen Methoden der Unterschied zwischen ausreichender Überzeugung und vollständiger Sicherheit. Allerdings sei der Preis für absolute Sicherheit sehr hoch, da schwergewichtige Methoden benötigt werden, die nur sehr schwierig angewendet werden können. Der leichtgewichtige MBT-Ansatz dagegen, würde laut Kicillof, viel besser skalieren und wurde bereits erfolgreich in großen Projekten, auch innerhalb Microsoft, angewendet.

2.1 Anwendung

Nachdem die theoretischen Grundlagen des modellbasierten Testen erwähnt wurden, beschäftigt sich dieser Abschnitt mit der Anwendung der Methode. Neben vielen Texten im Internet gibt es ein Buch, das besonders hervorzuheben ist. Margus Veanes, Colin Campbell und Wolfram Schulte haben das *Spec Explorer* Team bei Microsoft verlassen und zusammen mit Jonathan Jacky ein Buch über modellbasiertes Testen mit C# veröffentlicht [JJ08]. Zu diesem Buch haben besagte vier Personen eine open-source Implementierung eines modellbasierten Test-Werkzeugs namens *NModel* released [Mic08], welches auf codeplex kostenlos heruntergeladen werden kann. Inwiefern *NModel* die gleichen Funktionen wie *Spec Explorer* besitzt, konnte im Rahmen dieser Seminararbeit leider nicht näher untersucht werden.

2.2 Schwierigkeiten

Obwohl die Methode des modellbasierten Testen laut Nico Kicillof [Kic09] wesentlich einfacher und besser anwendbar als andere Verifikationsverfahren sind, bestehen auch hier einige Schwierigkeiten. Wolfgang Grieskamp hat sich in seinem Buch [Gri06] mit diesen Problemen auseinander gesetzt. In den folgenden Unterabschnitten werden die fünf Probleme näher beschrieben, die die effektive Nutzung der Methode mithilfe eines Werkzeugs besonders schwierig, beziehungsweise umständlich machen.

2.2.1 State Explosion

Wolfgang Grieskamp beschreibt in seinem Buch [Gri06] und in seinem Blogpost [Gri09] das Problem, dass in einer Anwendung sehr viele Zustände existieren, als *State Explosion*. Die Durchführung einer *State Space Exploration*, also das Ermitteln aller Zustände eines Systems, wird insofern erheblich erschwert, da signifikante Zustände nicht in der Fülle von unwichtigen Zuständen erkannt werden können. Weiterhin besteht die Möglichkeit einer unendlich großen Zustandsmenge im Falle der Verwendung einer listenartigen Datenstruktur. Eine Operation, die auf einer Liste ausgeführt werden kann, hat für jede Konfiguration der Liste einen Zustand als Vorbedingung. Wenn eine Liste theoretisch unendlich viele Elemente enthalten kann, existieren unendlich viele Vorbedingungen. Ähnlich ist es wenn Objekte dynamisch instantiiert werden. Für jedes neue Objekt existiert eine neue Vorbedingung für jede globale Operation. Dieses Phänomen beschreibt Grieskamp als *State Explosion*, da die Menge der Zustände sprichwörtlich explodiert, also unkontrollierbar groß wird.

2.2.2 Spezifikation

Ein weiteres von Wolfgang Grieskamp beschriebenes [Gri06] Problem besteht in der Art und Weise wie ein Modell spezifiziert wird. In *AsmL-T* wurde ein Modell in *AsmL* spezifiziert und erforderte somit die Kenntnis der Sprache. Das Erlernen einer neuen Sprache ist in der Industrie aufgrund von Zeitnot meist nicht möglich. Da ein Modell eine Realität besonders einfach und abstrahiert repräsentieren soll, sollte dieses Modell auch ohne großen Aufwand erstellt werden können. Die Schwierigkeit der Modellerzeugung sollte in der Formalisierung der Anforderungen liegen und nicht in der technischen Übersetzung der Anforderung in eine spezielle Sprache. Je einfacher eine solche Sprache nutzbar ist, desto unwahrscheinlicher ist die Einführung von Fehlern bei der Formalisierung der Anforderungen.

2.2.3 Zustände und Szenarien

In Unterabschnitt 2.2.1 wurde die Menge an ermittelten Zuständen bereits erwähnt. Da Zustände in Unterschiedlichen Kontexten auftreten können, ist eine Abstraktion von diesen Zuständen erforderlich, um die Funktion der Implementierung den Zuständen zuordnen zu können. Wolfgang Grieskamp beschreibt eine Lösung dieses Problem als *Szenario-Orientierung* [Gri06]. Ein Szenario ist ein Ablauf von verschiedenen Funktionen des Systems. Das System erreicht darin mehrere Zustände in einer bestimmten Reihenfolge. Ein Beispiel für ein Szenario wäre folgendes:

1. Benutzer meldet sich am System an.
2. Benutzer führt eine Aktion aus.
3. Benutzer melded sich am System wieder ab.

Szenarien können daher auch *User Stories* genannt werden. Laut Grieskamp [Gri09] unterstützen viele MBT-Werkzeuge nur das Testen der der Zustände der Implementierung, nicht aber das Testen von Szenarien. Er sieht eine Schwierigkeit in der Kombination des Zustandsraums (*state space*), der durch die *State Space Exploration* generiert wird, mit den aktionsorientierten Szenariobeschreibungen. Während Testfälle, wie sie in Abschnitt 1.1 als Tests der dritten

Ebene beschrieben wurden, Szenarien testen, sollte ein MBT-Ansatz diese Szenarien ebenfalls testen können um eine vergleichbare Testgranularität gewährleisten zu können.

2.2.4 IDE Unterstützung

Wolfgang Grieskamp identifiziert in seinem Blogpost [Gri09] die mangelnde Integration in Entwicklungswerkzeuge als weiteres Problem. Viele formale Verifikationsverfahren nutzen spezielle Programme, die eine hohe Einarbeitungszeit erfordern, die Entwickler dagegen in der Regel nicht aufbringen können. Wenn die Methode des modellbasierten Testen an Beliebtheit und Bekanntheit zunehmen soll, muss eine Umgebung genutzt werden, in der sich die Entwickler auskennen. Es kann allgemein als unrealistisch angesehen werden, dass ein Entwickler eine neue Umgebung erlernen wird, wenn er nur eine indirekte Produktivitätssteigerung beim Testen erreichen kann. Im Gegensatz dazu ist eine Testumgebung, mit der der Entwickler bereits vertraut ist, die ideale Voraussetzung. Die zu testende Implementierung, das Modell und die Verifikation sollte laut Grieskamp über eine einheitliche Benutzerschnittstelle erreichbar sein. Da die IDE (Integrated Development Environment) bereits für die Implementierung des Systems genutzt wird, liegt eine Integration des MBT-Werkzeugs in dieses Programm nahe.

2.2.5 Testfallerzeugung

Wenn MBT-Werkzeuge eine Implementierung anhand eines Modells verifizieren, dann sollte der Prozess der Verifikation ebenfalls von einem Test Runner durchgeführt werden können, der auch *normale* Tests durchführen kann. Wolfgang Grieskamp sieht in der fehlenden Unterstützung für Programme, die Testfälle automatisiert ausführen können, sogenannten *Test Runners*, ein weiteres Problem [Gri09] des MBT-Ansatzes. Die Verifikation anhand eines Modells sollte ohne hohe Laufzeitkosten durchgeführt werden können, also ohne jedes mal eine *State Space Exploration* durchführen zu müssen. Besonders vor dem Hintergrund von Continuous Integration ist eine automatische Verifikation als Teil eines automatisierten Buildvorgangs durchaus denkbar. Der Schritt des Testen der Implementierung sollte laut Grieskamp so allgemein wie möglich durchgeführt werden können und zwar auf eine Weise, die dem traditionellen Testen mit *Unit Tests* sehr ähnlich ist. Eine Erzeugung der Testfälle für ein bekanntes Testing Framework würde das modellbasierte Testen, zumindest was den Schritt der Verifikation angeht, erheblich erleichtern.

3 Spec Explorer 2010

Die aktuelle Version des Tools ist *Spec Explorer 2010*. Ein erster Prototyp des Werkzeugs, der dem heutigem Funktionsumfang annähernd entspricht, wurde bereits 2006 veröffentlicht. Nachdem in dem vorigen Kapitel 2 die Idee des modellbasierten Testen beschrieben und auf deren Schwierigkeiten in der Anwendung eingegangen wurde, wird dieses Kapitel die Visual Studio Erweiterung *Spec Explorer* vorstellen. Dabei wird zunächst gezeigt, wie *Spec Explorer* die in Abschnitt 2.2 vorgestellten Schwierigkeiten versucht zu lösen. Die Punkte, die im Folgenden als Lösungen beschrieben werden, sind aus der Evaluierung des Werkzeugs hervorgegangen.

3.1 Die Lösung

In diesem Abschnitt wird beschrieben, wie *Spec Explorer* die in Abschnitt 2.2 beschriebenen Schwierigkeiten bei der Verwendung eines MBT-Werkzeugs adressiert.

3.1.1 State Explosion

Das in Unterabschnitt 2.2.1 beschriebene Problem, dass der Zustandsraum sehr groß wird löst *Spec Explorer* durch Abbruchgrenzen. So ist es möglich den Zustandsraum nur für eine definierte Anzahl an Interation zu explorieren. Weiterhin lassen sich auch Abbruchbedingungen als sogenannte *Bounds* definieren, die Zustände ab einer definierten Entfernung vom Startzustand ignorieren. Was Konfigurationsmöglichkeiten betrifft nutzt *Spec Explorer* sogenannte *configs* und *machines*, welche in einer Konfigurationsdatei definiert werden. Diese Konfiguration wird über eine Scriptsprache namens *CORD* realisiert.
Über *configs* können Einstellungen wie beispielsweise Abbruchbedingungen für die Exploration definiert werden. Weiterhin können auch die Methoden des Modells, die für die Exploration von Bedeutung sind, mit einem Wertebereich für Parameter versehen werden um spezielle Parameterkombinationen zu erzeugen. Diese Methoden werden von *Spec Explorer* genutzt um das Modell zu explorieren und somit alle Zustände des Modells zu erreichen. Dabei werden die Parameterkombinationen berücksichtigt.
Über *machines* kann die Benutzbarkeit des Modells definiert werden. In Abschnitt 4.1 wird der Exploration Manager vorgestellt. In diesem Manager, der die Interaktion des Benutzers mit dem Tool ermöglicht, werden alle *machines* aufgelistet. Eine *machine* könnte zum Beispiel die Exploration des Modells repräsentieren, oder den Zustandsraum mit Szenarien, die ebenfalls als *machine* definiert werden können, verbinden. Auch die Testfallerzeugung wird mit einer *machine* beschrieben.

3.1.2 Spezifikation

Spec Explorer löst das Problem des in Unterabschnitt 2.2.2 als Modellspezifikation beschrieben Problem auf eine sehr einfache Art. Die formale Spezifikation des Modells, das die Implementierung repräsentieren soll, kann mithilfe der .NET Sprache C# vorgenommen werden.

Dadurch muss der Entwickler keine neue Syntax lernen und kann sich auf die Übersetzung von Anforderungen in ein C#-Modell konzentrieren. Unterstützung für andere .NET Sprachen als Modellsprache konnte im Rahmen der Evaluierung nicht festgestellt werden.

3.1.3 Zustände und Szenarien

In Unterabschnitt 2.2.3 wurde das Problem beschrieben, dass Funktionen, wie Abschnitt 1.1 vorgestellt, als Szenarien getestet werden. Erst durch ein erfolgreich getestetes Szenario kann davon ausgegangen werden, dass ein Feature einer Anwendung funktioniert. Ein Szenario ist, wie bereits erwähnt, eine Abfolge von Benutzer-Aktionen: Einloggen, Bestellen, Ausloggen wäre ein Beispiel. *Spec Explorer* erlaubt es, Szenarien als *machines* zu definieren. Dazu wird eine *CORD*-Syntax verwendet, die sehr stark an reguläre Ausdrücke erinnert, um Aktionen, die im Modell als solche gekennzeichnet werden, in definierter Reihenfolge aufzurufen. Welche Möglichkeiten diese Ausdrucksweise bietet, konnte in dieser Seminararbeit leider nicht weiter analysiert werden.
Wird die *machine* eines Szenarios über den Exploration Manager aufgerufen, wird das Modell anhand des Szenarios ebenfalls exploriert und als Zustandsdiagramm dargestellt. Dabei nimmt das Modell pro Aktion einen Zustandswechsel vor, sodass mehrere Zustände erreicht werden. In einer weiteren *machine* können die Zustände, die durch ein Szenario erreicht werden, aus der Zustandsraum-Exploration herausgeschnitten werden. *Spec Explorer* spricht hier von *slicing*. Das Ergebnis dieser parallelen Ausführung der Explorations-Maschine und der Szenario-Maschine kann ebenfalls als Zustandsdiagramm dargestellt werden. Hier werden dann alle Zustände angezeigt, die durch Aufrufe der Aktionen des Modells mit konkreten Werten im Rahmen eines Szenarios erreicht werden. Es entsteht also ein Zustandsautomat, der ein Szenario, also die Features einer User Story, repräsentiert. Aus dieser kombinierten *machine* werden später auch die Testfälle für dieses Feature erzeugt.

3.1.4 IDE Unterstützung

In Unterabschnitt 2.2.4 wurde mangelnde Unterstützung für Entwicklungsumgebungen als Problem klassifiziert. *Spec Explorer* ist eine Visual Studio 2010 Erweiterung und damit in die IDE von Microsoft integriert. Die Integration wird in Kapitel 4 näher beschrieben. Unterstützung für andere Entwicklungsumgebungen ist nicht vorhanden. Da Erweiterungen für Visual Studio 2010 erst ab der Professional Edition installiert werden können, ist *Spec Explorer* leider nicht für die Visual Studio Express Editionen verfügbar.

3.1.5 Testfallerzeugung

Das Funktionen der Anwendung Szenario-orientiert getestet werden, wurde bereits in Abschnitt 1.1 erwähnt. Wie Szenarien mit *Spec Explorer* im Rahmen der Zustandsraum-Exploration verwendet werden können, wurde in Unterabschnitt 3.1.3 beschrieben. *Spec Explorer* nutzt das standardmäßig mit Visual Studio 2010 Professional ausgelieferte Test Framework (*Microsoft.VisualStudio.TestTools.UnitTesting*), sowie dessen Integration *MS Test*. Damit laufen die von *Spec Explorer* erzeugten Testfälle im Test Runner von Visual Studio. Unterstützung für andere Test Frameworks konnte leider nicht festgestellt werden.
Testfälle werden durch eine *machine* erzeugt, die den Zustandsautomaten eines Szenarios in einzelne lineare Abläufe von Aktionen aufteilt. Aus einem Szenario werden so mehrere isolierte

Abläufe, die in dem Automaten parallel existieren. Die erzeugten Tests sind überladen und erhalten neben den Aufrufen der Aktionen der Implementierung, die durch das Modell festgelegt wurden, eine Menge an Kommentaren, die den Testfall unleserlich machen. Da die erzeugten Tests jederzeit neu aus der entsprechenden *machine* erzeugt werden können, ist eine manuelle Änderung nicht vorgesehen.
Bei der Testfallerzeugung kommt zum ersten mal die Implementierung zum Tragen. Während die Zustände und Szenarien über dem Modell definiert werden, nutzen die erzeugten Tests die Implementierung. Die Tests rufen die über Aktionen auf dem Modell definierten Methoden der Implementierung auf, und vergleichen anschließend den Zustand der Implementierung mit dem Zustand den das Modell in gleichem Kontext hatte. Der Test schließt erfolgreich ab, wenn die Implementierung der gleichen Zustand besitzt wie das Modell. Der Test schlägt fehl, wenn die Zustände nicht identisch sind. Also wird die in Kapitel 2 beschriebene Verifikation der Implementierung anhand der durch das Modell spezifizierten Zustände über die erzeugten Testfälle vorgenommen. Die Verifikation könnte dank der erzeugten Testfälle auch ausgelagert werden und beispielsweise automatisch durchgeführt werden, sobald ein neuer Build der Implementierung entsteht. Die Testfälle müssen erst neu erzeugt werden, wenn sich die Anforderungen und damit das Modell der Software ändern.

3.2 Umgebung

Abbildung 3.1 illustriert den Ablauf des Testens mit *Spec Explorer*. Dabei werden unter anderem die drei Bestandteile der Umgebung dargestellt (die Implementierung, das Modell und die erzeugten Testfälle) und in wie diese zusammenhängen.

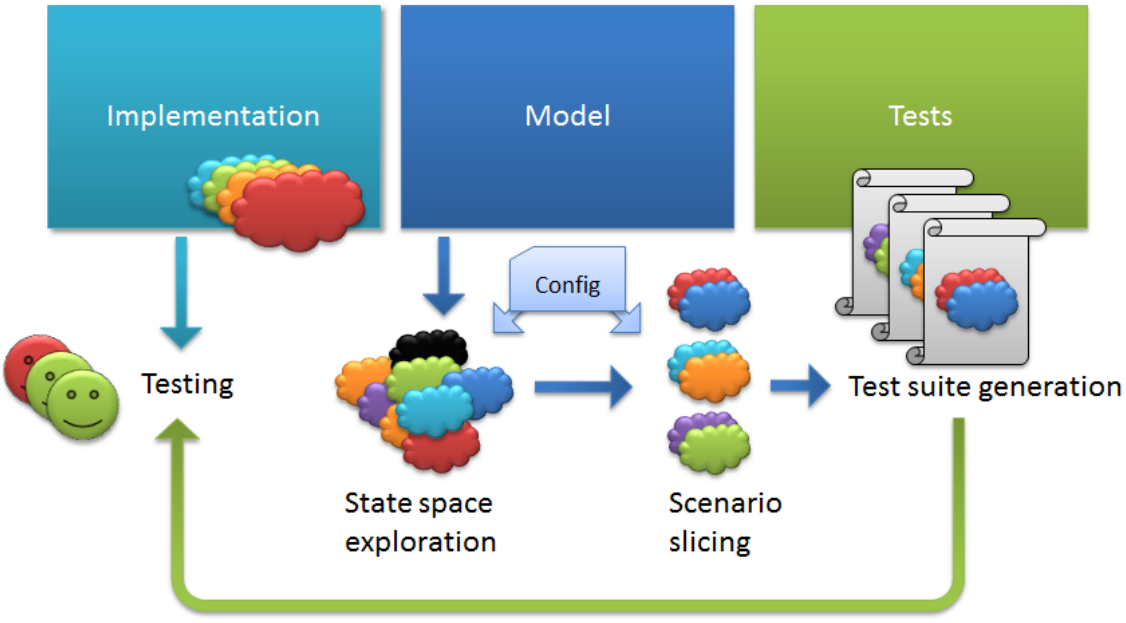

Abbildung 3.1: Schematische Darstellung des Arbeitsablaufs bei der Verwendung von *Spec Explorer*

In der semantischen Darstellung wird davon ausgegangen, dass bereits eine Implementierung besteht, die anhand eines ebenfalls bestehenden Modells verifiziert wird. Alternativ könnte auch nur ein Modell existieren und die Implementierung würde erst nach der Testfallerzeugung im Rahmen eines *Test First*-Ansatzes entstehen. Ebenfalls kann eine bestehende Implementierung auch nachträglich getestet werden, indem das Modell nachträglich entsteht. Eine gute Spezifikation entsteht dann, wenn Modell und Implementierung möglichst unabhängig voneinander entstehen.
In diesem Abschnitt werden die Zusammenhänge von Implementierung, Modell und Konfiguration des Modells beschrieben.

3.2.1 Implementierung

Der erste Teil einer *Spec Explorer* Umgebung besteht aus der Implementierung. In Abbildung 3.2 ist links die Umgebung zu sehen, die durch *File/New Project/Spec Explorer Project* erstellt wird. Die Visual Studio Solution enthält drei Projekte. Das erste Projekt ist das Modell-Projekt und enthält neben der Klasse *AccumulatorModelProgram* die Konfiguration *Config.cord*. Das zweite Projekt ist das Implementierung-Projekt und enthält nur die Klasse *Accumulator*. Das letzte Projekt ist ein *MS Test*-Projekt, dass die erzeugten Testfälle enthält. Da das Test-Projekt aus dem Modell-Projekt entsteht, wird dieses Projekt an dieser Stelle nicht weiter behandelt.

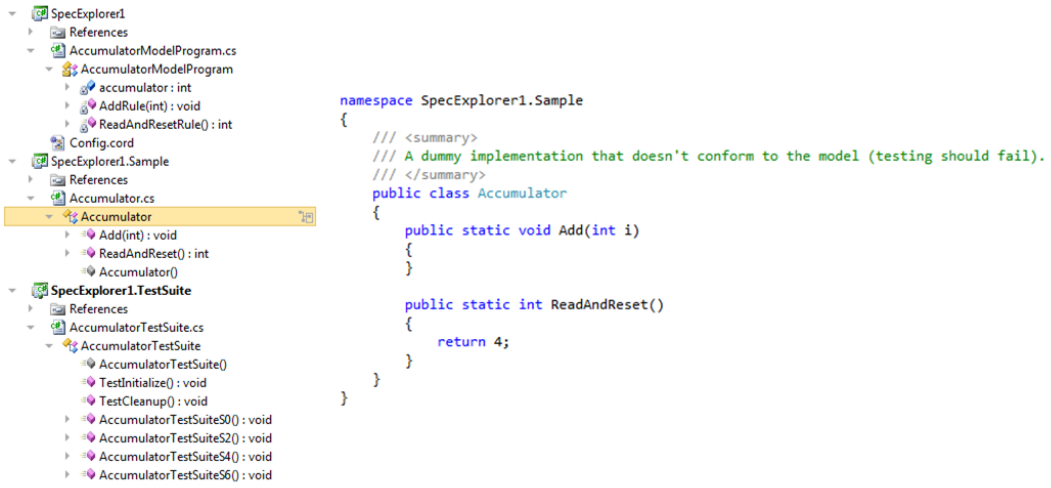

Abbildung 3.2: Fehlerhafte Implementierung

Die erwähnte Projektstruktur ist typisch für ein *Spec Explorer* Projekt, auch wenn die im Beispiel erwähnten Klassen dem Sample entsprechen, das *Spec Explorer* automatisch erzeugt, wenn ein Projekt erstellt wird. In einer produktiven Umgebung würden die erstellten Klassen entfernt und durch eigene ersetzt werden. Ein *Spec Explorer* Projekt kann auch nachträglich aus einem bereits existierenden Projekt erzeugt werden, indem ein Modell- und Test-Projekt angelegt werden. Dieser Fall wird im Folgenden nicht weiter betrachtet.
Die in Abbildung 3.2 dargestellte Klasse *Accumulator* ist Teil der Implementierung. Die dargestellte Implementierung ist fehlerhaft, da immer der Wert *4* zurück gegeben wird. Auch wenn dieses Beispiel trivial erscheint, werden die Eigenschaften von *Spec Explorer* sowie der entspre-

chende Arbeitsablauf an diesem Beispiel gut deutlich. Die Absicht der Implementierung ist in diesem Fall Ganzzahlen aufzuaddieren und zurückzugeben. Die aufsummierten Zahlen sollen dabei wieder auf 0 gesetzt werden. Dieses Verhalten soll durch ein Modell verifiziert werden können.

3.2.2 Modell

Das zweite Projekt in der *Spec Explorer* Solution enthält das Modell. *Spec Explorer* unterstützt zwei Modell-Typen. Zum einen das statische Modell und zum anderen das nicht-statische Modell. Das statische Modell ist eine statische Klasse, dessen Aktionen ebenfalls nur statisch sind. Dadurch braucht sich *Spec Explorer* bei der Exploration des Zustandsraumes nicht um Instanzen kümmern. Das nicht-statische Modell verwendet Methoden von Objekten als Aktionen während der Exploration. Dadurch können unterschiedliche Zustände entstehen, je nachdem auf welchem Modell-Objekt eine Aktion ausgeführt wird. Aus Demonstrationszwecken wird im Folgenden der statische Ansatz verwendet. Das bedeutet nicht, dass das statische Modell keine Objekte verwenden kann, sondern lediglich die Aktionen müssen statisch bleiben. Dieser Ansatz hat also nur ein einziges Objekt-Kontext und das ist die Klasse selbst.

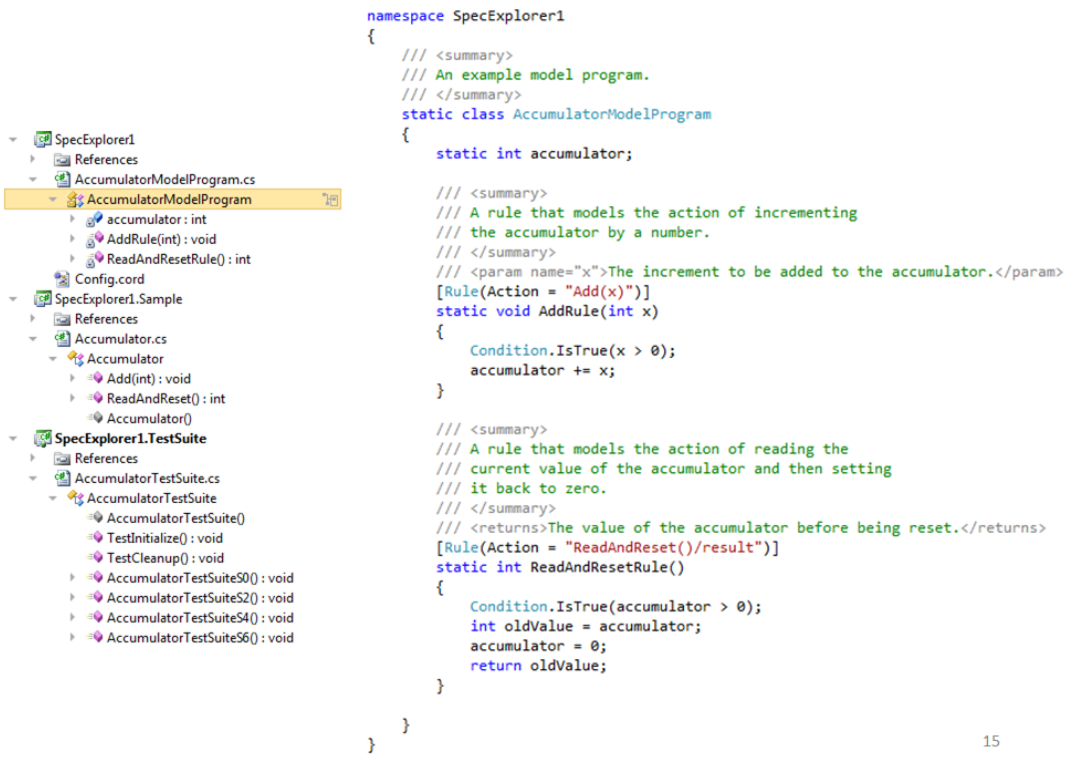

Abbildung 3.3: Das Modell der Implementierung

In Abbildung 3.3 ist der Quellcode des Modells des Aufsummierers dargestellt. Methoden, die als Aktionen unterschiedliche Zustände während der Exploration erzeugen, werden als Regel definiert, indem ein *Rule*-Attribut zur Dekoration der Methode verwendet wird. In diesem Attribut wird auch die Methode der Implementierung spezifiziert, die der Zustandsänderung

entspricht. Dies ist notwendig, da die erzeugten Testfälle genau diese Methode nutzen sollen, um die erwartete Zustandsänderung der Implementierung herbeizuführen und anschließend zu verifizieren. Die Methodensignaturen müssen kompatibel sein.
Die Regeln des Modells entsprechen einer einfachen Realisierung der Implementierung. Das Modell einer Datenbank könnte beispielsweise eine Hashtable als Datenstruktur verwenden. Weiterhin können *Code Contracts*, die bereits in Abschnitt 1.3.3 über *Spec#* erwähnt wurden, verwendet werden. Im Beispiel enthält jede Regel des Modells eine Pre-Condition, wodurch sichergestellt wird, dass das Modell nur positive Zahlen, die größer als Null sind, aufaddiert. Dadurch wird die Zustandsraum-Exploration eingeschränkt um dem Problem der Zustandsraum-Explosion aus Unterabschnitt 2.2.1 entgegengewirkt. Aufrufe dieser Aktion erzeugen nur neue Zustände, wenn die Vorbedingungen gelten.

3.2.3 Konfiguration

Neben dem Modell enthält das zweite Projekt einer *Spec Explorer* Solution auch eine Konfiguration. Diese Konfiguration besteht aus den in Unterabschnitt 3.1.1 bereits vorgestellten *configs* und *machines*. Abbildung 3.4 zeigt die beiden *configs* der Konfiguration des Summen-Beispiels. Die *Main-config* nimmt über *switches* allgemeingültige Einstellungen der Explorationen der *machines* vor. Unter anderem wird in dem Beispiel der Pfad zu dem Test-Projekt festgelegt, in dem die Testfälle erzeugt werden sollen. Eine maximale Aktions-Tiefe für Aufrufe der Modellregeln wird ebenfalls festgelegt. Die wichtigste Einstellung ist die Definition der Regeln der Modells als Aktionen (*actions*).

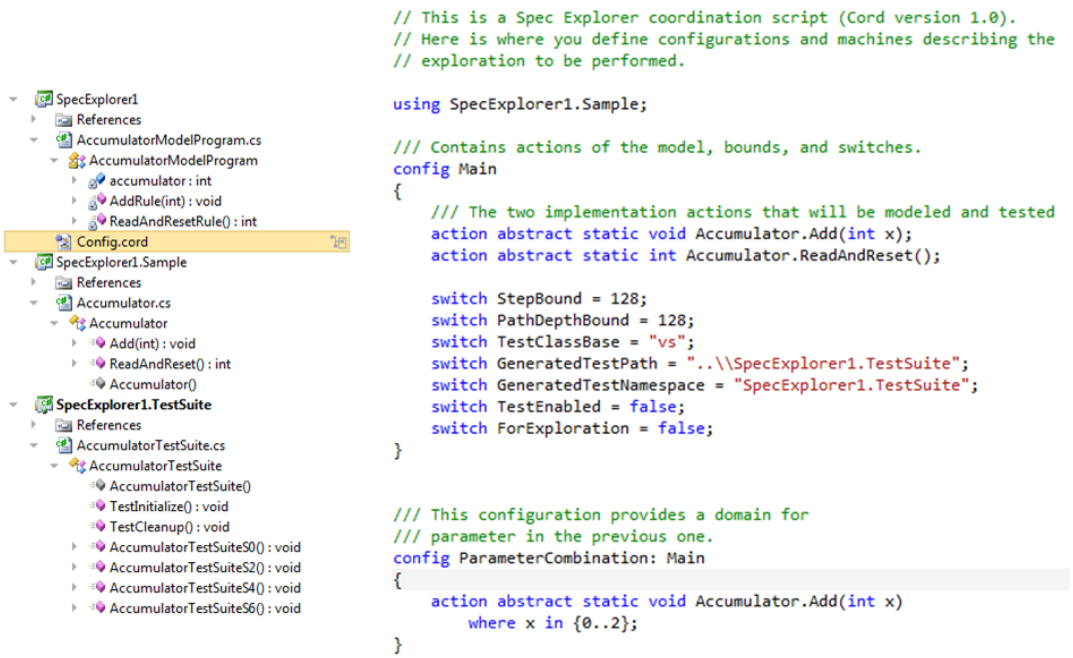

Abbildung 3.4: Die Konfiguration der Modellexploration

Die *ParameterCombination-config* stellt eine Parameterkombination dar, wie sie bereits in Unterabschnitt 3.1.1 beschrieben wurde. Diese Kombination ist eine Erweiterung für die *Main-*

config, was durch die Notation der Vererbung ausgedrückt wird. Die Aktion *Accumulator.Add(int x)* wird im Beispiel eingeschränkt, sodass der Wert für x aus dem Intervall *[0..2]* gewählt wird. Dies ist eine weitere Möglichkeit die Zustandsraum-Explosion aus Unterabschnitt 2.2.1 in den Griff zu bekommen.
Die Konfiguration besteht ebenfalls aus *machines*. Abbildung 3.5 zeigt die Definition von vier Maschinen. Die *AccumulatorModelProgram-machine* nutzt die *Main-config* und erlaubt die Exploration. Der komplette Zustandsraum wird unter Verwendung der in *ParameterCombination-config* definierten Parameterkombination exploriert. Die *DoubleAddScenario-machine* definiert ein Szenario, wie es in Unterabschnitt 3.1.3 vorgestellt wurde. Diese Maschine erlaubt ebenfalls die Exploration, allerdings nicht die des Zustandsraums des Modells, sondern die des Zustandsraums des Szenarios, ohne Werte des Modells. Die Syntax der Definition der Regel-Aufrufe ((*Add*, *Add*, *ReadAndReset*) beliebig oft und mit beliebigen Werten) ähnelt sichtlich der Syntax von regulären Ausdrücken.

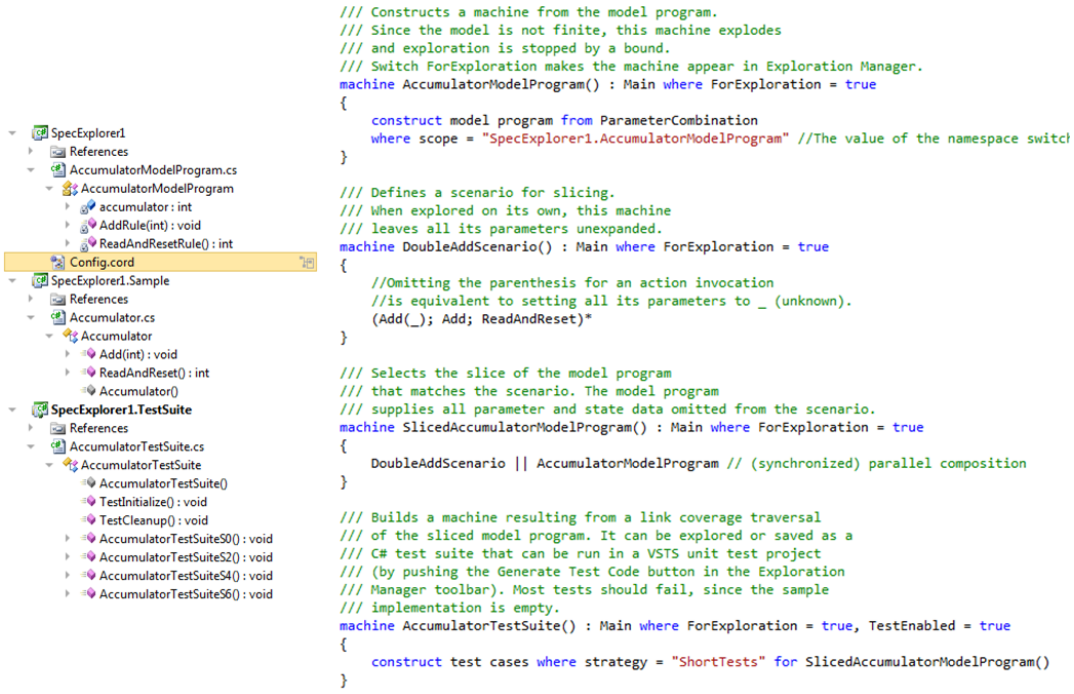

Abbildung 3.5: Die Konfiguration der Modell-Maschinen

Die *SlicedAccumulatorModelProgram-machine* lässt sich ebenfalls explorieren und verbindet den Zustandsraum mit dem Szenario. Die Notation der Verbindung über den ||-Operator lässt beide Maschinen parallel ausführen. Das Ergebnis ist also die Exploration des Zustandsraums des Szenarios, unter Berücksichtigung der Zustände des Modells. Es werden also alle möglichen Zustände gefunden, die das Modell, im Rahmen des Szenarios, haben kann. Die *AccumulatorTestSuite-machine* ist ebenfalls explorierbar und kann Testfälle erzeugen. Dafür nutzt diese Maschine das Explorationsergebnis der *SlicedAccumulatorModelProgram-machine*. Wenn diese Maschine exploriert wird, werden die zusammenhängenden Zustandsübergänge der Exploration des Szenarios aufgesplittet, sodass konkrete Durchläufe des Szenarios mit speziellen Werten entstehen.

Aus der Exploration werden also wieder getrennte Aufruf-Ketten von Aktionen bestimmt, aus denen anschließend kurze Testfälle erzeugt werden können.

3.3 Exploration

Eine in der Konfiguration definierte *machine* kann exploriert werden. Abbildung 3.6 zeigt die grafische Zustandsraum-Exploration des Modells, die von *Spec Explorer* erstellt wurde. Der Zustandsautomat wird von *Spec Explorer* automatisch gezeichnet und erlaubt panning und zooming, sowie die Auswahl von Zuständen und Zustandsübergängen. Dabei werden für die von *Spec Explorer* automatisch nummerierten Zustände weitere Eigenschaften, wie beispielsweise der aktuelle Wert der Modell-Veränderlichen, angezeigt.

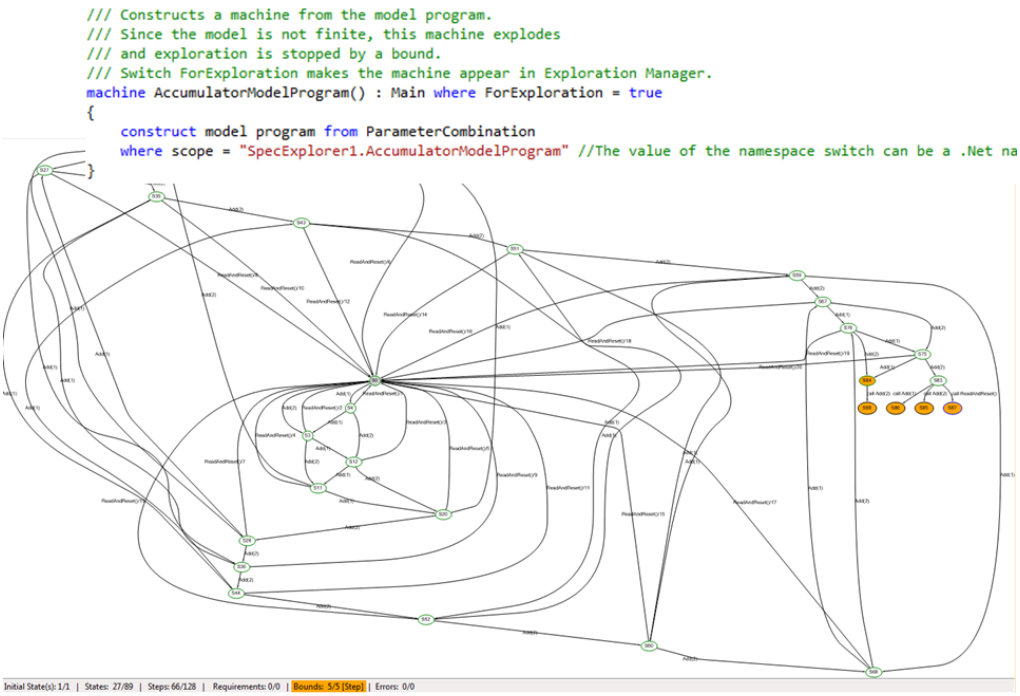

Abbildung 3.6: Darstellung der Exploration des Modells als Zustandsautomat

Aus Abbildung 3.6 geht ebenfalls hervor, dass die Exploration nicht unendlich ist. Die gelb eingefärbten Zustände markieren die Abbruchgrenze des Zustandsraums. Bei diesen gelben Zuständen ist die Grenze von 5 Bounds erreicht, sodass die Exploration abbricht. Bounds lassen sich über *configs* in der Konfiguration einstellen.

3.4 Szenarien

Szenarien wurden bereits in Unterabschnitt 3.1.3 behandelt. Dort wurde erklärt, wie *Spec Explorer* Szenarien als *machines* definieren kann, die ebenfalls exploriert werden können. In Abbildung 3.7 ist die Exploration des (*Add*, *Add*, *ReadAndReset*)-Szenarios dargestellt. Der grafische Zustandsautomat, der von *Spec Explorer* erzeugt wurde, ist sehr anschaulich und entspricht dem

Verhalten, dass man auch von der Implementierung erwarten würde.

```
/// Defines a scenario for slicing.
/// When explored on its own, this machine
/// leaves all its parameters unexpanded.
machine DoubleAddScenario() : Main where ForExploration = true
{
    //Omitting the parenthesis for an action invocation
    //is equivalent to setting all its parameters to _ (unknown).
    (Add(_); Add; ReadAndReset)*
}
```

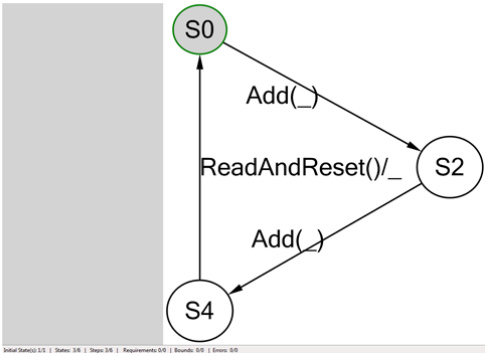

Abbildung 3.7: Darstellung eines Szenarios als Zustandsautomat

In Abbildung 3.8 ist die Exploration der *machine* zu sehen, die die Zustände des Szenarios aus dem kompletten Zustandsraum des Modells herausschneidet.

```
/// Selects the slice of the model program
/// that matches the scenario. The model program
/// supplies all parameter and state data omitted from the scenario.
machine SlicedAccumulatorModelProgram() : Main where ForExploration = true
{
    DoubleAddScenario || AccumulatorModelProgram // (synchronized) parallel composition
}
```

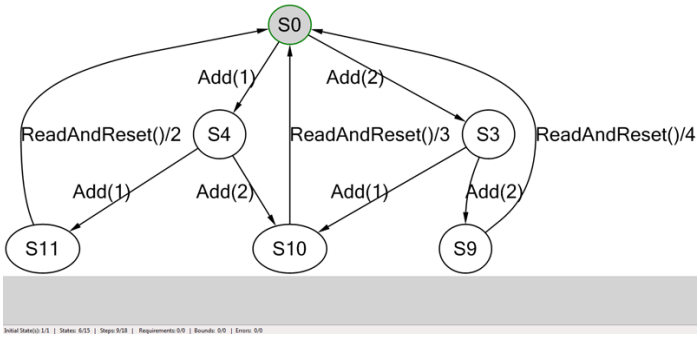

Abbildung 3.8: Darstellung eines Szenarios in Verbindung mit der Zustandsraum-Exploration

Das Ergebnis entspricht einem Zustandsautomaten, der das Szenario komplett abbildet. Alle Zustände, die das Modell in Bezug auf das Szenario haben kann, werden erreicht. Um das Szenario an der Implementierung zu testen, müssen jetzt lediglich die Aufrufe, die für die Zustandsübergänge von dem Startzustand *S0* über andere Zustände hinweg zurück zum Startzustand führen, als getrennte Aufruf-Ketten interpretiert werden.

3.5 Testfälle

Die getrennten Aufruf-Ketten, die für die Testfälle erforderlich sind, werden, wie am Ende von Unterabschnitt 3.2.3 beschrieben, über eine weitere *machine* definiert. Abbildung 3.9 zeigt die Exploration aller gültigen Pfade durch den Zustandsautomaten des Szenarios aus Abbildung 3.8. Die von der Maschine verwendete Strategie *ShortTests* ist für die Auftrennung der Pfade verantwortlich. Auf weitere Strategien, wie beispielsweise *LongTests*, wird in dieser Seminararbeit verzichtet.

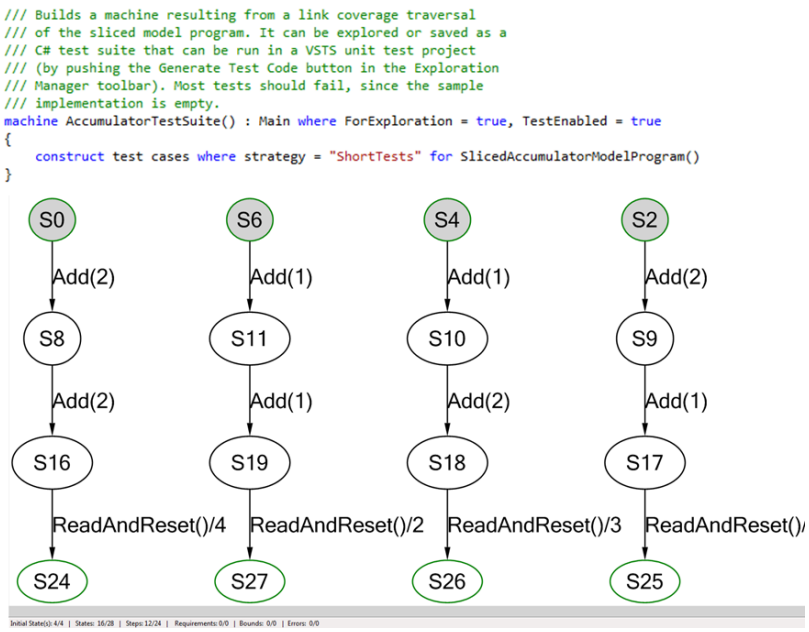

Abbildung 3.9: Darstellung aller testbaren Ausprägungen eines Szenarios

Diese Aufruf-Ketten können von *Spec Explorer* eins zu eins in Testfälle übersetzt werden. Der Exploration Manager unterstützt die Testerzeugung einer Maschine, die mit *TestEnabled = true* definiert ist. Auffällig ist hier die Syntax, die sich sehr an die menschliche Sprache anlehnt.
Spec Explorer erzeugt für jede Ablauf-Kette einen Testfall (*TestMethodAttribute*) in der Test-Klasse (*TestClassAttribute*) des Szenarios. Die erzeugten Tests haben eine Abhängigkeit von der *Spec Explorer* Runtime Testumgebung. Theoretisch könnten die Tests auch ohne diese Abhängigkeit funktionieren. Durch diese Referenz haben die Tests die Möglichkeit Timeouts zu setzen, sowie die Ausführung detailliert zu protokollieren. Abbildung 3.10 zeigt die Test-Klasse, während die Testfälle für die vier Aufruf-Ketten zugeklappt sind.

Abbildung 3.11 zeigt den erzeugten Testfall exemplarisch für die Aufruf-Kette (*Add(2)*, *Add(2)*, *ReadAndReset()/4*). Der Test ruft die Methoden der Implementierung auf, die mit den Regeln des Modells verknüpft wurden (wie Unterabschnitt 3.2.2 beschrieben) und vergleicht abschließend ob der Rückgabewert der *ReadAndReset*-Aktion dem erwarteten Wert *4* entspricht.

Die Tests sind mit den Testwerkzeugen in Visual Studio 2010 ab Professional ausführbar. Aufgrund der fehlerhaften Implementierung schlagen alle Tests fehl, deren Erwartungswert ungleich *4* ist.

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.239
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace SpecExplorer1.TestSuite {
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Reflection;
    using Microsoft.SpecExplorer.Runtime.Testing;
    using Microsoft.VisualStudio.TestTools.UnitTesting;


    [System.CodeDom.Compiler.GeneratedCodeAttribute("Spec Explorer", "3.5.3130.0")]
    [Microsoft.VisualStudio.TestTools.UnitTesting.TestClassAttribute()]
    public partial class AccumulatorTestSuite : VsTestClassBase {

        public AccumulatorTestSuite() {
            this.SetSwitch("ProceedControlTimeout", "100");
            this.SetSwitch("QuiescenceTimeout", "30000");
        }

        Test Initialization and Cleanup

        Test Starting in S0

        Test Starting in S2

        Test Starting in S4

        Test Starting in S6
    }
}
```

Abbildung 3.10: Codeausschnitt eines erzeugten Testfalls

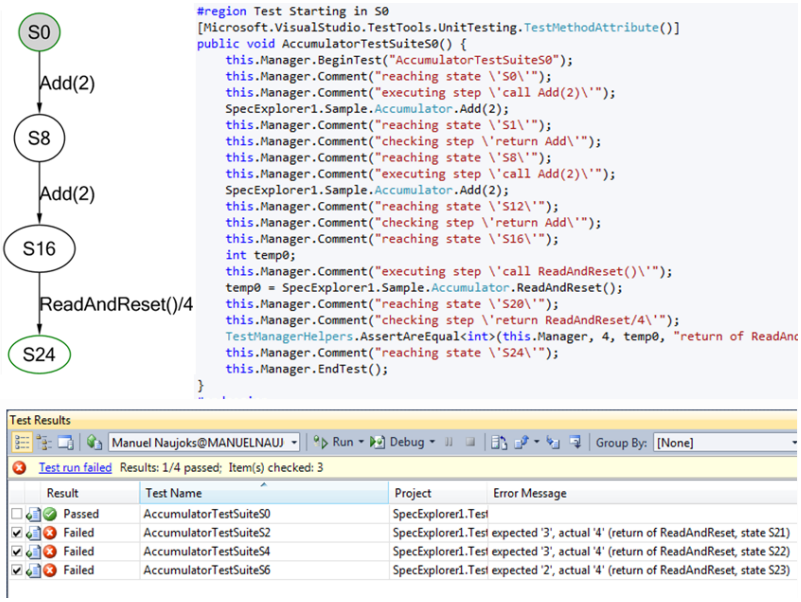

Abbildung 3.11: Codeausschnitt einer Test-Methode im erzeugten Testfall

Da die Testfälle selbst viel protokollieren, ist die Beschreibung eines fehlgeschlagenen Tests sehr aussagekräftig und hilfreich. Der Entwickler sieht sofort, inwiefern sich der erwartete Zustand vom tatsächlichen unterscheidet, und zwar direkt auf die Variablen der Implementierung bezogen. Damit ist die Testerzeugung aus dem Modell komplett von *Spec Explorer* übernommen worden. Auf diese Weise können hilfreiche Testfälle erzeugt werden, die noch nicht einmal programmiert werden mussten.
Abbildung 3.12 zeigt eine korrigierte Version der in Abbildung 3.2 gezeigten fehlerhaften Implementierung. Diese Implementierung entspricht der einfachsten Version einer Summen-Funktion und funktioniert, was durch das erfolgreiche Durchlaufen aller vier Testfälle impliziert wird.

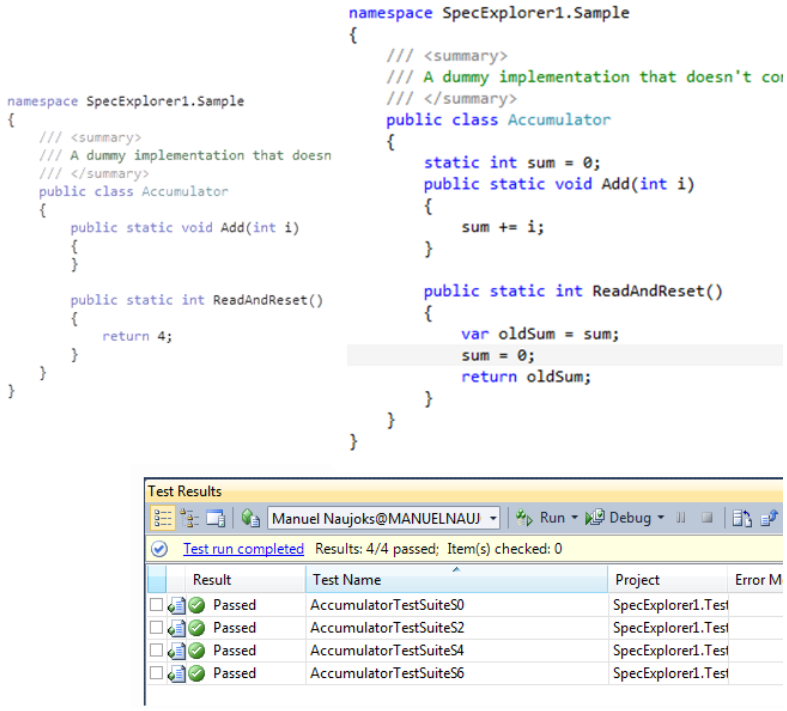

Abbildung 3.12: Korrigierte Implementierung

Damit endet die Evaluierung der Funktionsweise der Visual Stuido Erweiterung *Spec Explorer* 2010. Es wurde beschrieben, welche Projekt-Komponenten eine *Spec Explorer* Solution beinhaltet und wie diese zusammenhängen. Dabei wurde auf die Bedeutung der Konfiguration von Maschinen eingegangen und wie aus diesem Maschinen das Modell aus unterschiedlichsten Perspektiven im Rahmen von Szenarien exploriert werden kann. Weiterhin wurde gezeigt, wie Testfälle erzeugt werden können, die das zustandsorientierte Verhalten der Implementierung mit den Zuständen des Modells vergleichen. Von diesem Standpunkt aus, kann mit einer hohen Wahrscheinlichkeit von der Korrektheit der Implementierung ausgegangen werden. Im Gegensatz zu allen anderen Test-Methoden führt eine Veränderung der Spezifikation nur zu einer kleinen Änderung des Modells. Alle Testfälle können anschließend automatisch neu erzeugt werden.

4 Integration in Visual Studio

In diesem Kapitel wird auf die Integration der *Spec Explorer* Erweiterung in Visual Studio eingegangen. Die Intallation der Erweiterung fügt ein neues Projekt-Template unter *File/New Project* und ein neues Fenster, den Exploration Explorer, hinzu.

4.1 Exploration Manager

Der Exploration Manager ist ein Fenster, das sich in Visual Studio wie eine Toolbox beliebig andocken lässt. Es enthält eine Auflistung aller in der Konfiguration beschriebenen *machines*, die in Unterabschnitt 3.2.3 vorgestellt wurden. Jede dieser Maschinen kann über das Kontext-Menü exploriert werden, sofern die Maschine Exploration nicht unterdrückt. Aus einer *machine*, die *TestEnabled* = *true* definiert, können Testfälle erzeugt werden, die anschließend im Test-Projekt gespeichert oder einfach nur ausgeführt werden können. Abbildung 4.1 zeigt den Screenshot der Auflistung der Maschinen sowie das Kontext-Menü.

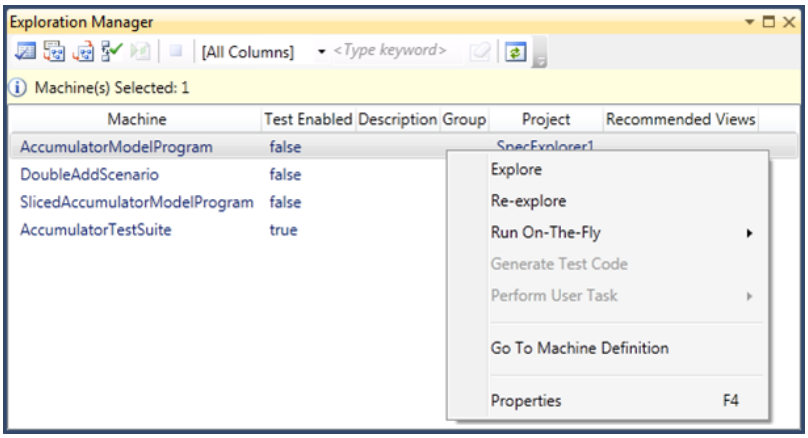

Abbildung 4.1: Screenshot des Exploration Manager Fensters in Visual Studio 2010

4.2 UML Extensions

Eine Erweiterung für *Spec Explorer* ist die Visual Studio Erweiterung *UML Extensions* [Mic11b]. Diese Extension erlaubt es aus einem explorierten Szenario ein UML-Sequenzdiagramm zu erzeugen. Aus diesem Sequenzdiagramm sind die konkreten Abläufe, die im Rahmen des Modells stattfinden, deutlich besser nachzuvollziehen und erlauben eine bessere Abbildung der Exploration auf den Quellcode des Modells. Leider kann im Rahmen dieser Seminararbeit nicht näher auf diese Erweiterung eingegangen werden.

5 Einsatz bei Microsoft

In diesem Kapitel wird auf die Verwendung des Tools innerhalb Microsofts eingegangen. Leider konnten keine Information über die Verwendung durch andere Einrichtungen gefunden werden, sodass die Microsoft für diesen Akzeptanzbericht gewählt wurde.
Wolfgang Grieskamp beschreib in seinem Blogpost [Gri09], dass das *Spec Explorer* Team seit 2007 in der Windows Server Division angesiedelt ist. Das ist dadurch zu erklären, dass diese Abteilung viele eigene und standardisierte Protokolle einsetzt, die effektiv und effizient getestet werden müssen. Im Rahmen der „Microsoft Open Specifications“ Bewegung [Mic11a] hat Microsoft begonnen sämtliche in Windows enthaltene Protokolle ausführlich schriftlich zu spezifizieren und zu testen. Dies hatte eine enge Zusammenarbeit mit dem *Spec Explorer* Team zur Folge, was zu dem Umzug des Team geführt hat. Mittlerweile gibt es neben dem Team in Redmond auch ein *Spec Explorer* Team in Peking, China, das sich mit der Entwicklung und dem Testen von *Spec Explorer* beschäftigt.
Wolfgang Grieskamp hat zusammen mit Nico Kicillof und weiteren *Spec Explorer* Experten eine Ausarbeitung geschrieben, die sich mit dem modellbasierten Qualitätssicherung der *Windows Protocol Documentation* auseinandersetzt [WG08].
In seinem Blogpost [Gri09] beschreibt Wolfgang Grieskamp die erfolgreiche Nutzung von Spec Explorer in einem Großprojekt mit einem Testaufwand von insgesamt 250 Mann-Jahren. Dabei geht nicht hervor, ob es sich bei diesem Projekt um die „Microsoft Open Specifications“ Bewegung handelt. Im Rahmen dieses Großprojekts wurden die beteiligten Teams vor die Wahl gestellt ein modellbasiertes Test-Werkzeug, höchst wahrscheinlich *Spec Explorer*, oder einen traditionellen Test-Ansatz zu nutzen. Laut Grieskamp haben sich 50% für das modellbasierte Tool und 50% für traditionelles Unit-Testing entschieden. Zusammenfassend bescheinigt Grieskamp den Teams, die das modellbasierte Test-Tool eingesetzt haben, eine Effizientsteigierung von durchschnittlich 42% nach einem Einsatz von 2,5 Jahren. Diese Steigerung wurde durch testunerfahrenes Personal erreicht und nicht, wie vielleicht vermutbar wäre, durch testerfahre Mitarbeiter. In welchem Zusammenhand diese Steigerung steht, erklärt Grieskamp leider nich. Es kann aber davon ausgegangen werden, dass die 42% im Vergleich zu den anderen Team, die das modellbasierte Tool nicht eingesetzt haben, verstanden werden können.

6 Fazit

Nachdem in den vorigen Kapiteln die Visual Studio Erweiterung *Spec Explorer*, sowie Grundlagen und Entwicklung beschrieben wurden, fasst dieses Kapitel die wichtigsten Gegenstände, die *Spec Explorer* besonders auszeichnen, zusammen.

Spec Explorer ist ein Test-Werkzeug, das es erlaubt Testfälle für eine Implementierung automatisch zu erzeugen. Diese Testfälle werden aus einem Modell, das die Implementierung repräsentiert, abgeleitet. In dem Modell werden die Anforderungen an die Software in C# spezifiziert. Dabei bleibt das Modell abstrakt und formal. *Spec Explorer* kann dieses Modell lesen und versucht alle Zustände, die das Modell haben kann, durch Ausführen von Regeln, die den Zustand des Modells verändern, zu erreichen. Es entsteht die sogenannte Zustandsraum-Exploration.
Auf diesem Zustandsraum lassen sich Zustände als Szenarien, die durch Aufruf-Ketten von Regeln definiert werden, zusammenfassen. Durch die Verbindung von Zustandsraum und Szenario lassen sich alle Zustände des Modells finden, die durch das Szenario entstehen können. Aus diesen Zuständen können die als Szenario definierten Aufruf-Ketten als Pfade auftrennen. Ein Pfad entspricht also einer mit konkreten Werten verbunden Aufruf-Kette von Regeln und erreicht bestimmte Zustände.
Aus einer solchen Ablauf-Kette kann ein Testfall erzeugt werden. Durch die Reihenfolge der Regel-Aufrufe können die entsprechenden Methoden des Modells aufgerufen werden. Der Zustand, den das Modell nach Ablauf der Aufruf-Kette hat, wird mit dem Zustand verglichen, den die Implementierung nach Aufruf der Methoden hat. Ein solcher Testfall wird von *Spec Explorer* für *MS Test* erzeugt und kann somit direkt durch Visual Studio ausgeführt werden.
Spec Explorer testet also ob die Implementierung die gleichen Zustände erreicht wie das Modell, wenn die gleichen Regeln oder Methoden zur Veränderung der Veränderlichen des Systems, beziehungsweise des Modells, ausgeführt werden. Diese Art von Testfall kann auch als *State-based* Testen bezeichnet werden. Das bedeutet auch, das *Spec Explorer* nicht die Struktur oder die Interaktion des Systems testet, sondern nur dessen Zustand.

Abschließend kann *Spec Explorer* als modellbasiertes Test-Werkzeug zusammengefasst werden, das die Zustände einer Implementierung anhand der Zustände einer Spezifikation verifizieren kann. Meiner Meinung nach ist das Tool sehr umfangreich und lässt sich gut nutzen. Das mitgelieferte Beispiel ist anschaulich und lädt zum Experimentieren ein. Manchmal sind die Möglichkeiten, beispielsweise Einstellungen für die Exploration vorzunehmen, nicht offensichtlich, sodass eine Web-Recherche erforderlich wird. Dort findet man leider noch nicht viel Material, was meines Erachtens daran liegt, dass das Tool nicht sehr bekannt ist. Die automatische Erzeugung von Testfällen ist sehr hilfreich, da man diese nicht programmieren muss. Wenn man kein Freund von *Test-Driven-Development* (TDD) ist, erhält man von *Spec Explorer* viel automatisierte Unterstützung.

Literaturverzeichnis

[Bör] Egon Börger. Abstract state machine tutorial. http://www.di.unipi.it/~boerger/ASMTutorialEtaps.html.

[Com11] Wikipedia Community. Unit testing. http://en.wikipedia.org/wiki/Unit_testing, 2011.

[DuW10] Rob DuWors. Spec explorer 2010 release 3.3 now available as a visual studio power tool! http://blogs.msdn.com/b/specexplorer/archive/2010/09/30/spec-explorer-2010-release-3-3-now-available-as-a-visual-studio-power-tool.aspx, 2010.

[EB03] Robert Stärk Egon Börger. *AsmBook*. Springer-Verlag, 2003.

[Gri06] Wolfgang Grieskamp. *Multi-paradigmatic Model-Based Testing*. Springer-Verlag, 2006.

[Gri09] Wolfgang Grieskamp. The spec explorer story. http://blogs.msdn.com/b/wrwg/archive/2009/10/28/the-spec-explorer-story.aspx, 2009.

[Gur93] Yuri Gurevich. Evolving algebras 1993: Lipari guide. http://research.microsoft.com/en-us/um/people/gurevich/opera/103.pdf, 1993.

[Gur99] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. ftp://www.eecs.umich.edu/groups/gasm/seqthesis.pdf, 1999.

[JJ08] Colin Campbell Wolfram Schulte Jonathan Jacky, Margus Veanes. *Model-based Software Testing and Analysis with Csharp*. Cambridge University Press, 2008.

[Kic09] Nico Kicillof. What is model-based testing? http://blogs.msdn.com/b/specexplorer/archive/2009/10/27/what-is-model-based-testing.aspx, 2009.

[Mic03] Microsoft. Asml (codeplex). http://asml.codeplex.com/, 2003.

[Mic08] Microsoft. Nmodel (codeplex). http://nmodel.codeplex.com/, 2008.

[Mic10] Microsoft. Spec explorer 2010 visual studio power tool (visual studio gallery). http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745/, 2010.

[Mic11a] Microsoft. Microsoft open specifications. http://www.microsoft.com/openspecifications/en/us/default.aspx, 2011.

[Mic11b] Microsoft. Uml extensions for spec explorer 2010 (visual studio gallery). http://visualstudiogallery.msdn.microsoft.com/ce73da2a-072f-44d0-ae18-600213b56520, 2011.

[Pod08] Matthew Podwysocki. Code contracts for .net 4.0 - specsharo comes alive. http://codebetter.com/matthewpodwysocki/2008/11/08/code-contracts-and-net-4-0-spec-comes-alive/, 2008.

[WG08] et al. Wolfgang Grieskamp, Nico Kicillof. Model-based quality assurance of windows protocol documentation. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=4539580&isnumber=4539517?tag=1, 2008.