Universidade do Minho

Nuno Gil Correia Veloso da Veiga

Generating Automatically Test Cases Based on Models

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Prof. Dr. João Alexandre Saraiva

Abril 2012

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/108995/2008.

Acknowledgements

I would like to thank the following people for their help and support over the course of the completion of this thesis.
To João Saraiva, my supervisor, for his continued support and encouragement.
To Luís Anjos and Nuno Vieira, my co-workers, that helped in everything I needed during my research time in Primavera.
To my parents, Carlos and Isabel, that, even though in my life I may have not done everything the way they wanted and wished, were always there for my best interest, caring for me, my life and my future. I feel eternal gratitude for the two of you.
To my lovely sister, Catarina, for giving me company even when I didn't want it, for loving me unconditionally and for bringing a new cat into our household. Life with kittens is always better than life without kittens!
To my aunts, Nocas, Sissi and Patrícia, that I am sure have always believed in me.
To my grandfather, that age only made of him a more interesting, funny and reliable company. To my grandma, for all the love and food filled of love that fed me during all these years.
To my girlfriend Ana, for all the love, patience and understanding, and for staying by my side.
To all my friends, who make my life outside of research so enjoyable. Particularly, to my friends Ricardo, Renato, Miguel, Pedro Nuno, Bruno and Sebastião, whose company has been a constant for many years. To Zé and João Nuno, for the great time we have when we are together.

Abstracts

PRIMAVERA Business Software Solutions has dedicated a lot of time and effort in the last years in the development of a *framework* that allows a programmer to model an application and its respective services. This *framework* will then generate a big part of, not only the application source code, but also its database and the User Interface. The objective of this dissertation project is to add a new feature to this framework - the test automation component. By using such a component, the *framework* will be able to generate test cases to validate the applications requirements. Since trying all possible combinations would be unpractical, one of the main objectives of this project is to generate a smaller set of test cases only that can assure that the system is being properly tested.

Resumo

PRIMAVERA Business Software Solution dedicou muito tempo e esforço nos últimos anos no desenvolvimento de uma *framework* que permite a um programador modelar uma applicação e respectivos serviços. Esta *framework* irá então gerar grande parte do código fonte, assim como a base de dados e o User Interface. O objectivo deste projecto de dissertação é de adicionar uma nova função a esta *framework* - uma componente de automação de testes. Desta forma, a *framework* será capaz de gerar casos de teste para validar os requisitos da aplicação. Como gerar todas as possiveis combinações seria impraticável, um dos principais objetivos deste projetoé gerar um conjunto de casos de teste mais pequeno que possa garantir que o sistema está a ser a ser bem testado.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the background about PRIMAVERA and the motivation for the development of this project. It will also describe the structure of this document.

1.1 Overview

Due to the high competitive era we live in software industry any bug in the application has high impact in the image brand of a company. Studies show that finding and fixing defects during the early requirements phase has a much lower cost than if left undetected after production. Hence, it is crucial for organizations to adhere to structured testing processes and thereby deliver higher quality systems in less time and with fewer resources. Even though with manual testing we can find many defects in a software application, it is a very time consuming process which leads to the attempt to automate testing. Automated Software Testing (AST) is an important and active area of reasearch and it can be described as the application and implementation of software technology throughout the entire software testing life-cycle (STL) with the goal to improve STL efficiency and effectiveness. Many of the unsuccessful AST attempts and myths related to AST implementation, plus lack of sufficient AST knowledge, lead to the question "Why should I automate?".

Over the last few years, PRIMAVERA BSS has dedicated a big investment in the development of a *framework* that allows the development of their management *software* with state-of-the-art technologies and with an architecture service-oriented. One of the main objectives of this *framework* is the ability to generate a large amount of the *software* source code, allowing an improvement on the development process. The development of

this *framework* consisted in the integration of a set of Domain Specific Languages (DSL) that can be found in the Microsoft Visual Studio. These DSLs combined and interacting with each other made it possible to model a large part of an application, such as entities and their relations, the usability model, and set of operations over those entities.

With the objective to make their products better, PRIMAVERA wanted a module for their *framework* that could:

- Generate test cases in an automatic way
- Execute *User Interface* tests automatically
- Reuse the generated test cases for the new software versions

During my research time at PRIMAVERA, I was responsible for the development of the automatic test case generation module. I also provided help in fixing errors in the previously generated test scripts and added new features to those test cases. Since that extra work was outside the scope of the project it is not described in this document.

PRIMAVERA Business Software Solutions is a multinational company that develops and commercializes management solutions and platforms for business process integration in a global market, providing solutions for small, medium and large Organizations, as well as the public administration sector. About 40 thousand companies resort to PRIMAVERA BSS solutions everyday to optimize their business processes. PRIMAVERA BSS is present in many countries across the world, being market leader in many of them.

PRIMAVERA's commitment, since its establishment in 1993, has focused towards the development of avant-garde solutions that respond in advance to the future needs of companies. This effort has contributed much to its successful path, which is why PRIMAVERA is included in the 500 largest European companies with greatest growth potential, a ranking promoted by Growth Plus.

Above all, PRIMAVERA BSS is a brand that invests in the constant evolution of its competences and accomplishments, anticipating the needs and expectations of customers, companies, market and its surrounding universe. Motivated by a desire to exceed itself, in search of excellence in all activity sectors, PRIMAVERA BSS inspires itself in the future to innovate. Success is the result of the passion with which the company faces today's challenges.

1.2 Document Structure

The main body of this document is divided as follows.

- Chapter 2 contains a small studie about software testing techniques used in the current days, such as black-box, white-box and model-based testing.
- Chapter 3 introduces the Athena Framework, describing how it works aswell as its design principles and goals.
- Chapter 4 represents the main objective of this work, describing the steps for the development and the strategy used.
- Chapter 5 summarizes what has been achieved and what can be done in order to enhance the module.
- Appedix A contains the API for the developed source code.

Chapter 2

Automated Software Testing

2.1 Overview

The term automated software testing can have multiple meanings for members of the software development and testing community. To some the term may mean test driven development and/or unit testing; to others it may mean using a capture & record tool to automate testing. Or can even mean custom-developing test scripts using a scripting language such as Pearl, Python or Ruby. Generally, all tests that are currently run as part of a manual testing program - functional, performance, concurrency, stress, and more - can be automated. How is manual software testing different from AST? First of all, it enhances manual testing efforts by focusing on automating tests that manual testing can hardly accomplish. It doesn't replace the need for manual testers analytical skills, test strategy know- how, and understanding of testing techniques. This manual tester expertise serves as the blueprint for AST. It also can't be separated from the manual testing; instead both AST and manual testing are inter-winded and complement each other. AST refers to automation efforts across the entire STL, with a focus on automating the integration and system testing efforts. The overall objective of AST is to design, develop, and deliver an automated test and retest capability that increases testings efficiencies; if implemented successfully, it can result in a substantial reduction in the cost, time and recourses associated with traditional test and evaluation methods and processes for software-intensive systems.

2.2 Test Case Definition

We can find several definitions for what a test case is. As a matter of fact, it seems that each author has his own definition of what a test case is. Cem Kaner in his paper "What is a Good Test Case?" [Kan03] quotes definitions from several authors:

- IEEE Standard 610 (1990) defines a Test Case as:
  1. "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement"
  2. "(IEEE Std 829 - 1983) Documentation specifying inputs, predicted results , and a set of execution for a test item."
- For Ron Patton, test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software.
- Boris Beizer defines a test as "A Sequence of one or more subtests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial stat of the next. The word 'test' is used to include subtests, testes proper, and test suites".
- According to Bob Binder "A test specifies the pretest state of the implementation under testing (IUT) and its environment, the test inputs or conditions, and the expected result. The expected result specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, return values, and resultant state of the IUT and its environment, Test cases may also specify initial and resulting conditions for other object that constitute the IUT and its environment."
- For the author of the paper, Cem Kaner, a test case is a question that we ask to the program. It doesn't matter if the test fails or passes, what matter is information gain.

2.3 Techniques for Software Testing

According to [W.H87], testing can be classified as black box testing (specification-based) or white box testing (program-based). In the case of black box testing, the objective is to

reveal defects that are related to the external functionality, to the communication interfaces between modules, constrains like pre- and post-conditions and the behavior of the software itself. It has the name of "black box" because the software is handled like a black box from which we can't see what's inside, i.e., the content is unknown to us. This way, testers use the specification to obtain the test data without any concert about how the program that was implemented. On the other hand, program-based testing requires code handling and selection of test sets that exercise specific pieces of the code, not its specialization. The main objective is to identify faults in the program's internal structure.

2.4 Black Box Testing Techniques

2.4.1 Equivalence Partitioning

Somehow, this is the "mother of all testing" [Han08]. The objective in equivalence partitioning is to partition any set of possible values into sets that you think at are equivalent. For instance, we can have a field "Name". Being name a text input and assuming that according to the specification a valid name has a length between 5 and 20 and all characters have to be alphabetical, we can derive the following classes displayed on 2.1. The idea behind this is that all "Name" that have between 5 and 20 characters and are alphabetical should, in principle, be treated the same way. Following this, we can create tests that will cover all classes without the need to try out every possible value. An example on test cases that can be created for this case can be viewed on Table 2.2.

#	Class	Result
1	Name is alphabetical	Valid
2	Name is not alphabetical	Invalid
3	Name length between 5 and 20	Valid
4	Name length is less than 5	Invalid
5	Name length is greater than 20	Invalid

Table 2.1: Deriving 'Name' into Classes

#	Input	Result	Classes Covered
1	abc4	Fail	2
2	abc	Pass	4
3	Charles	Pass	1,3
4	Thequickbrownfoxjumpsoverthelazydog	Fail	5

Table 2.2: Possible set of Test Cases

2.4.2 Boundary Values

The boundary value technique is used along side with the equivalence partitioning. The objective of this technique is to test the border values of an equivalence partition since these boundaries are most of the times where errors that result in software faults occur. For example, if we have an input that will accept, according to the requirements, natural numbers that between [40..100] . The border values we will want to test are obviously 39,40,100,101.

2.4.3 Pairwise Testing

Pairwise testing has become a popular approach to software quality assurance because it often provides effective error detection at low cost [KKL08]. Pairwise is an algorithm that uses specially constructed test sets that guarantee every parameter value interact with each other at least one time. An example described in the next lines makes it easier to comprehend how pairwise works. Let's suppose our system is a Car comprised with the following components:

1. Transmission: Automatic, Manual
2. Engine: Electric, Gasoline
3. Horse Power : LowPower, MediumPower, HighPower
4. Airbag: Yes, No
5. Color: Black, Red, Yellow

If we sum this up will get a total of 2 x 2 x 3 x 2 x 3 = 72 possible combinations. Table 2.3 shows the 9 tests where each parameter interacts with each other at least once. The

effectiveness of pairwise testing is based on the observation that software faults often involve interaction between parameters. While some bugs can be easily detected using, for instance, boundary alues or using a "divide by zero" but some others can only be detected when we have multiple conditions at one time. According to [KKL08], pairwise testing could detect 70% for more than 90% of software faults for the application studied in [Kuh04].

Test	Transmission	Engine	Horse Power	Airbag	Color
1	Automatic	Gasoline	HighPower	Yes	Blac
2	Manuel	Electric	HighPower	No	Yellow
3	Automatic	Electric	LowPower	Yes	Red
4	Automatic	Gasoline	LowPower	No	Yellow
5	Manual	Electric	LowPower	No	Black
6	Manual	Gasoline	MediumPower	No	Red
7	Automatic	Electric	MediumPower	Yes	Yellow
8	Manual	Electric	HighPower	Yes	Red
9	Automatic	Electric	MediumPower	Yes	Black

Table 2.3: Pairwise Test Configuration

2.4.4 Dynamic Analysis

Dynamic analysis is the process of evaluating behavior of a system or component during execution. The most common analysis are the memory performance and CPU usage.

2.4.5 Error Guessing

As the name implies, error guessing is a technique where the tester simply based on past experience tries to guess errors in the system. There are some metrics that can be used as basis for error guessing like initializing data and repeat the process to check if the previous data was properly removed or try out some wrong type of data (e.g. negative number or inserting a string were a numeric value is expected).

2.5 White Box Techniques

First of all, we need to point out what Unit Testing is. Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use [DH07]. One of the main techniques for white-box testing is code coverage. Code coverage can be defined as metric used to measure the testing effort applied to the software application [BBA75]. The process to capture the metrics of Code Coverage involves Instrumentation of the program and execution of the tests. This way the code which has or hasn't been executed can be identified. We can see that unit Testing and code coverage are complementary to each other. Unit testing confirms the compliance of program performance with respect to the requirements, whereas code coverage reveals the areas left out of the testing [BBA75]. The method used to apply code coverage is done by creating a control-flow graph, where the nodes are statements of the program and the edges the control flows from one node to another. Code Coverage can have different types:

- Statement Coverage: its major benefit is that it is greatly able to isolate the portion of the code which will not be executed. This method tends to be expensive if we aim for 100% coverage.
- Branch Coverage: is defined as a metric for measurement of the outcomes of the decisions subjected to testing [BBA75].

2.6 Using a Framework to Automate Software Testing

A test automation framework is a set of concepts and tools that all together provide a basis for AST and simplifies its process. The use of a framework allows us to reduce cost with maintenance since it reuses scripts even if the test has any changes. This way the only file that suffers any changes is the test file case. The framework has function libraries , test data sources, object details and various reusable modules [Eli10].

1. defining the format to express if tests passed or failed.
2. creating a mechanism to connect to the application under test.
3. executing the tests.

4. reporting results.

The following sections will described the more common frameworks for testing automation, pointing out some of the benefits and disadvantages of their usage.

2.6.1 Data-Driven

Data-driven testing is a term used in the testing of computer software to describe testing done using a table of conditions directly as test inputs and verifiable outputs as well as the process where test environment settings and control are not hard-coded. In the simplest form the tester supplies the inputs from a row in the table and expects the outputs which occur in the same row. The table typically contains values which correspond to boundary or partition input spaces. [1]

In order to use Data-Driven we can use a tool that generates scripts by recording user actions. The "hard-coded" data is removed from the scripts and placed in external data files. These data files can be Excel spreadsheets, CVS or XML files, or even a database can serve as source for data. The generated scripts can be executed using this datasets from the data source. This approach can be used with unit (NUnit, JUnit, DUnit and MSTest tests) , functional (or GUI Testing) and load testing. Data-driven is especially useful in scenarios when an application needs to be quickly supplied a large amount of input values [Gup09]. Using Data-Driven provides us with the following benefits:

- We can have multiple datasets at one time
- Provides separation between the data and the scripts
- Increases test coverage by using lots of different data sets.
- Information like data inputs or outputs and expected results are stored in the form of a convenient managed text records
- As said before, having the possibility to store data in different data sources is an advantage.

And can have some concerns:

[1] http://en.wikipedia.org/wiki/Data-driven_testing

- Multiple scripts to maintain.
- Requires great expertise of the scripting language used by the automation tool.

2.6.2 Keyword-Driven

This framework requires the development of data tables and keywords, independent of the test automation tool used to execute them and the test script code that "drives" the software under testing and the data. In data-driven only the test data is included in the test data files, in Keyword-Driven the entire functionality of the application gets captured as step-by-step instructions for every test in a table. Once creating the test tables, a driver script (or a set of scripts) is written which executes each step executes the step based on the keyword contained the action field, performs error checking, and logs any relevant information [Gup09]. The main difference between data-driven and keyword-driven testing is that each line of data in a keyword script includes a keyword that tells the framework what to do with the test data on that line.

An example of a Keyword-driven testing table is displayed in Figure 2.1. The image is only displaying an extract of the file because the information that is needed can be seen in the first columns of each line.

	A	B	C	D	E	R
1	Enabled	TestType	TaskFriendlyName	TaskId	ListForEdition	ProductKeyTex
2	TRUE	Insert	Create Product	CreateEntityProduct	FALSE	P00001
3	TRUE	Compare	Update Product	UpdateEntityProduct	TRUE	P00001
4	TRUE	Update	Update Product	UpdateEntityProduct	TRUE	P00001
5	TRUE	Compare	Update Product	UpdateEntityProduct	TRUE	P00001
6	TRUE	Delete	Delete Product	DeleteEntityProduct	TRUE	P00001
7	TRUE	NotExist	Products	PresentQueryProducts	TRUE	P00001

Figure 2.1: Keyword-drive Table sample

This small example is executing the following steps:

1. Creates and checks if the entity is created.
2. Check if the edited data is the expected one.
3. Update the entity and checks if it went well.

2.7 Model-Based Testing

Model-Based Testing is a break-throught innovation in software testing because it completly automates the validation testing process [Utt05]. The main reasons for its success,according to [Pre05] are:

1. the need for quality assurance for increasingly complex systems
2. model-centric paradigms like UML with its connection to testing being more used
3. the arrival of the more test-oriented developtment methodologies

In a general term, Model-Based Testing is software testing in which test cases are generated in whole or in part from a model that describes some aspects of the system under test,such as the system's accepted input sequences, actions, conditions and output logic, or the flow of data through the application's modules and routines. [Puo] [EFW01]. The test cases generated from the model are executable and include an oracle component which assings a pass/fail verdict to each test [Utt05].

Model based testing helps to ensure a repeatable and scientific basis for product testing, gives good test coverage for the behavior of the products and allows tests to be directly linked to the system requirements [Utt05].

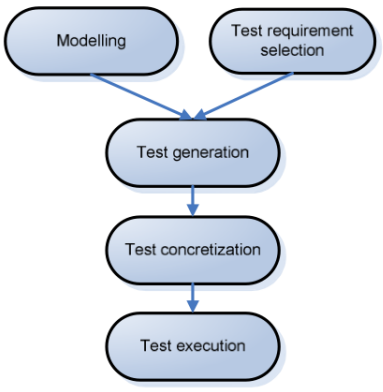

Figure 2.2: Model Based Testing

2.7.1 Model-Based Testing Phases

According to [Utt05], Model-Based Testing usually envolves four phases:

1. **building an abstract model of the system under test.** This is similar to the process of formally specifying the system, but the kind of specification/model needed for test generation may be a little different to that needed for other purposes, such as proving correctness, or clarifying requirements.

2. **validating the model (typically via animation).** This is done to detect gross errors in the model.With model-based testing, if some errors remain in the model, they are very likely to be detected when the generated tests are run against the system under test.

3. **generating abstract tests from the model.** This step is usually automatic, but the test engineer can control various parameters to determine which parts of the system are tested, how many tests are generated, which model coverage criteria are used etc.

4. **refining those abstract tests into concrete executable tests.** This is a classic refnement step, which adds concrete details missing from the abstract model. It is usually performed automatically, after the test engineer specifies a refinement function from the abstract values to some concrete values, and a concrete code template for each abstract operation.

After performing these steps, the real tests can be executed in order to detect in which tests the output is not the expected one. Model Based Testing provides useful feedback and error detection for the requirements and the model, as well as the system under test.

2.7.2 Advantages of Model-Based Testing

Using Model-Based Testing can result in the following benefits, regarding this project:

- Shorter schedules, lower cost, and better quality
- A model of user behavior
- Capability to automatically generate many non-repetitive and useful tests

- Test harness to automatically run generated tests
- Eases the updating of test suites for changed requirements
- Capability to assess software quality

2.7.3 Behavior Modeling Techniques

A variety of techniques/methods exist for expressing models of user/system behavior. These include, but are not limited to:

- **Decision Tables:** Tables used to represent sets of conditions and actions.
- **Finite State Machines:** Consists on a model with a finite number of states and transitions between those states, with their respective actions
- **Grammars:** describe the syntax of programming languages
- **Markov Chains (Markov process):** A stochastic process in which the probability that the process is in a given state at a certain time depends only on the value of the immediately preceding state
- **Statecharts:** Behavior diagrams specified as part of the Unified Modeling Language (UML).

Chapter 3

The Athena Framework

This chapter describes the Athena Framework. It points the main goals of its development and shows a small example on how to model an application using it.

The Athena project is a long-term project started by Primavera some years ago and it aims the development of a framework that can be used to model applications and generate automatically code for it, such as the User Interface, CRUD (Creat, Remove, Update, Delete) Operations and even the SQL scripts that create the Data Base. All these applications are under the concept of Software as a Service (SaaS). This means that the application and its associated data are hosted centrally, in this case in the Internet Cloud and will be accessed by used using a web browser over the internet. Recently, SaaS became a common delivery model for most business applications, such as accounting, enterprise resource planning (ERP), invoicing and human resource management (HRM).

3.1 Main Goals and Design Principles

The Athena project main objective is to provide a first version of the framework, usable outside the project team, to develop new solutions aswell as providing the features required to develop a full product.

The main design principles for this framework are the following:

- Design Patterns - the architecture implements known design patterns and uses best practices.

- Code reuse - the reutilization of third-party libraries to implement some parts of the architecture design is an important requirement since it can help the development productivity.
- Service Orientation - the Athena Framework follows a service-oriented design. This design principle guides the development of the framework interfaces (API), promoting independence between different component.
- Extensibility - all parts of the framework are designed with the goal of being easily extended and customized.
- Declarative Programming - most aspects of the framework behavior can be configured, customized and modified through XML-based configuration files. This approach allows that the change the behavior of the framework without having to recompile any software components.

3.2 Architecture

The Athena Framework is divided according to the logic layers shown on Figure 3.1

- Presentation Layer: this layer deals with the User Interface and User Experience.
- Service Layer: this layer includes all services that the client (presentation) uses to communicate with the server (lower layers).
- Business Logic Layer: this layer handles the business logic of the application, including the business processes that compose it.
- Data Layer: this layer is responsible for the data representation, persistence and extraction.

3.3 Athena Designers

The framework includes several designers, which one responsible to create a part of the application model.

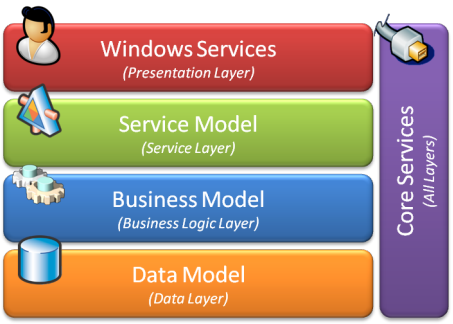

Figure 3.1: Athena Framework Architecture

- **Entities Designer**
  This designer will be used to model the data managed by the module: entities and their relations.

- **Services Designer**
  This designer will be used to model the behavior and the services provides by the module. It uses the entities model to obtain the entities available in each module.

- **User Interface Designer**
  This designer will be used to model the module's user interface. This model will contain the parts of the first version of the Entities Model that are concerned with UI.

- **Reporting Designer**
  This designer will be used to model the module's business intelligence, from the Query Builder data model to the BI OLAP cubes.

- **Common Features**
  These are the set of features that are common between all designers:

  - Serialization.
  - Import/Export.
  - Models cross-references.
  - Validation.
  - Code Generators.
  - Unit tests.
  - Design elements.

3.4 Modeling Applications in Athena

The applications developed using the Athena Framework are divided by modules that can be independent or dependent of each other. Those modules are then glued all together by what we call a 'Product'. This way, we can use developed modules for more than one solution. The figure 3.2 illustrates how the the products and modules can be related.

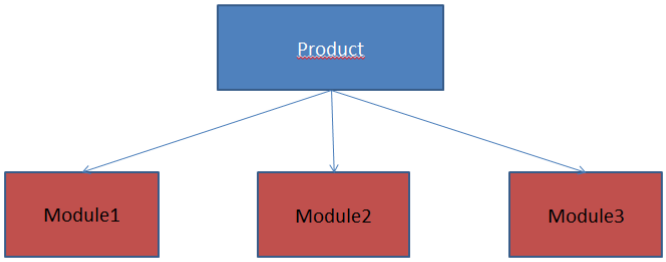

Figure 3.2: Athena Framework Module Development

In order to understand how we can develop a module using the Athena Framework, I made a brief description of the steps required to it:

- **Step 1 - Create Entities**

  Using the toolbox (Figure 3.3) and the Entities Designer we can create the Entities and their respective attributes as well as their relations. Figure 3.4 shows the graphical interface for creating entities.

  In order to create an Entity all we need is to drag and drop from the toolbox into the workspace. After that we can start to add attributes into that entity and the relations between entities using the "association" button from the toolbox. When an association is added, it is automatically created the foreign key attribute in the entity. Each attribute can have its own type. They following types are considered:

  - Text, ShortText, LongText, Memo for text type attributes. The length of the desired attribute determines which type to use.
  - Date, DateTime for dates.
  - Number, AutoNumber, Percentage, Decimal for numeric attributes.
  - Views for foreign keys.

- ValueList for enumerates.
- List for 1-n relations.

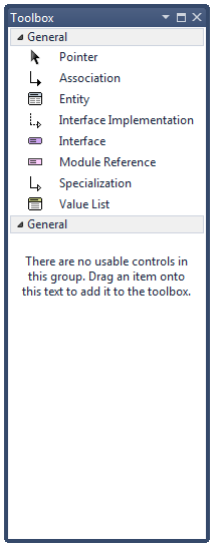

Figure 3.3: Toolbox

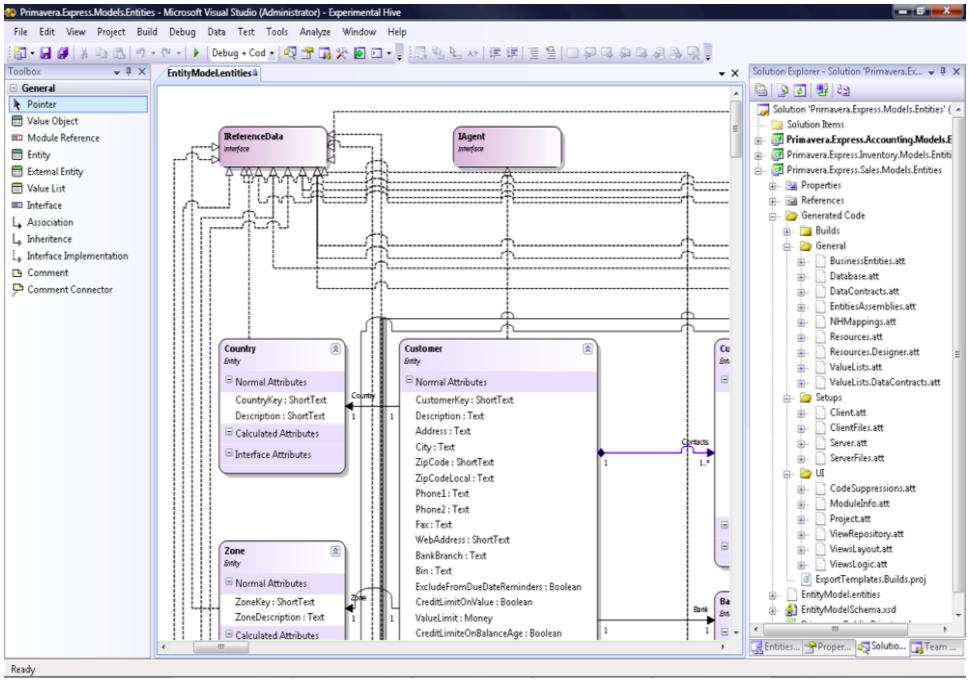

Figure 3.4: Entities Designer

- **Step 2 - Generate Services**

After we have created the Entities we can automatically generate all of the Insert, Update, Delete services for each Entity using the Services Designed. In case we don't want an entity to have one of more of those, for motives such as having an entity that is read only, we can deactivate them in Entities Designed. Figure 3.5 shows how the services interface looks like.

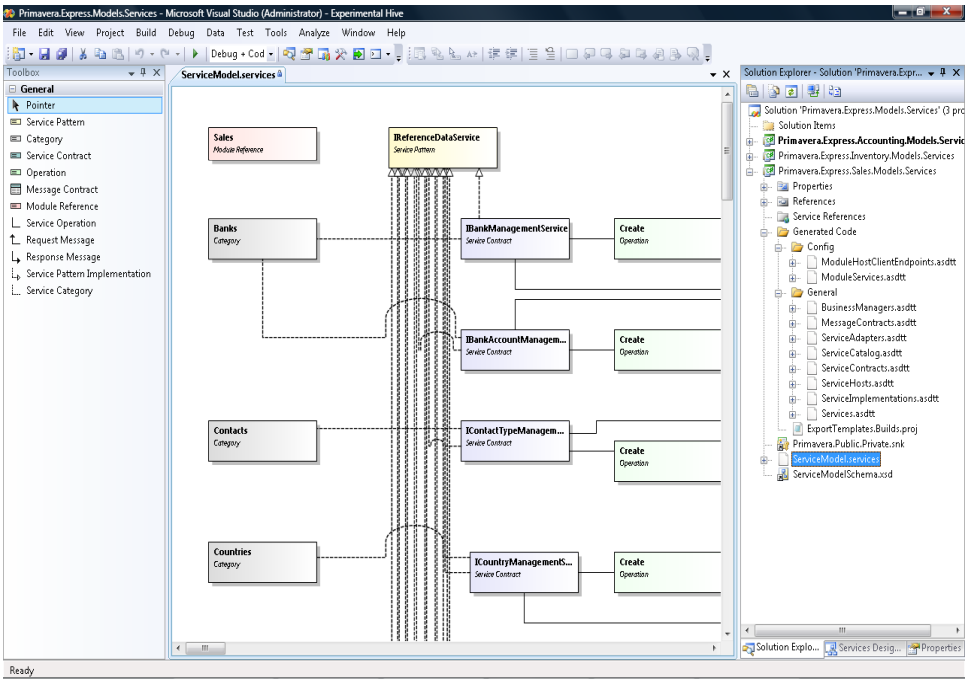

Figure 3.5: Services Designer

- **Step 3 - Transform All**

  Once those steps are done and we have no errors we can begin with the code generation using the Guidance Package. For this step all we need to do is press Transform All using the framework guidance package. When the *Transform All* ran and there were no errors, the test scripts that run using Visual Studio are also generated. Figure 3.6 shows the tests that are generating. For the ambit of this project only the ones highlighted, i.e., view tests are important.

After doing this, and if everything went well, we can compile and run the application. Note that only the logic layer, database and user interface is generated - all other custom code has to be added by the user so it can be a "real" application. There are also generated some elements for testing that will be described in the next section.

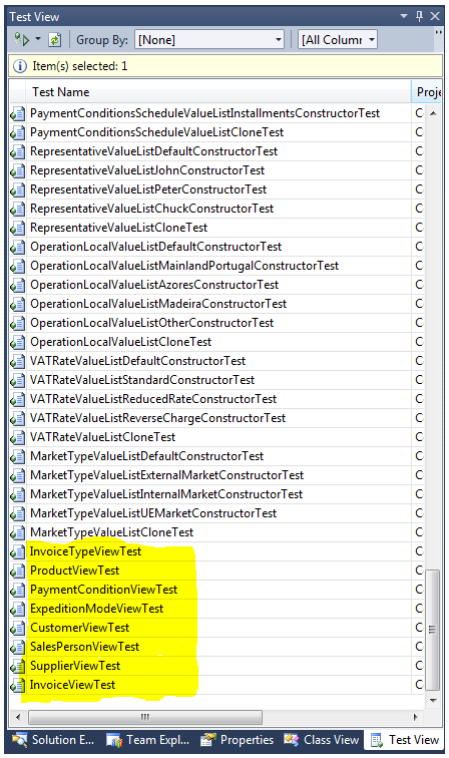

Figure 3.6: Generated Test Scripts

3.5 Developed Test Module

In order to develop the testing module, was mandatory to develop an application for testing. The Figure 3.7 shows the main module for the application, including the entitites and attributes, that was developed. By looking once again to Figure 3.7[1] we see that we have diferent elements in different colors. They representation is the following:

- Pink - represent the Modules used by the Module. Other modules can be imported to the module we are developing to import External Entities (see below).
- Green - are the External Entities, i.e., entities that are imported from other modules. In the case, are being imported External Entities from three other Modules.
- Grey - represents the Entities added to the Module we are developing.
- Brown - represents the ValueLists from the Entities.

[1] this document is better viewed when printed in color or in its electronic document

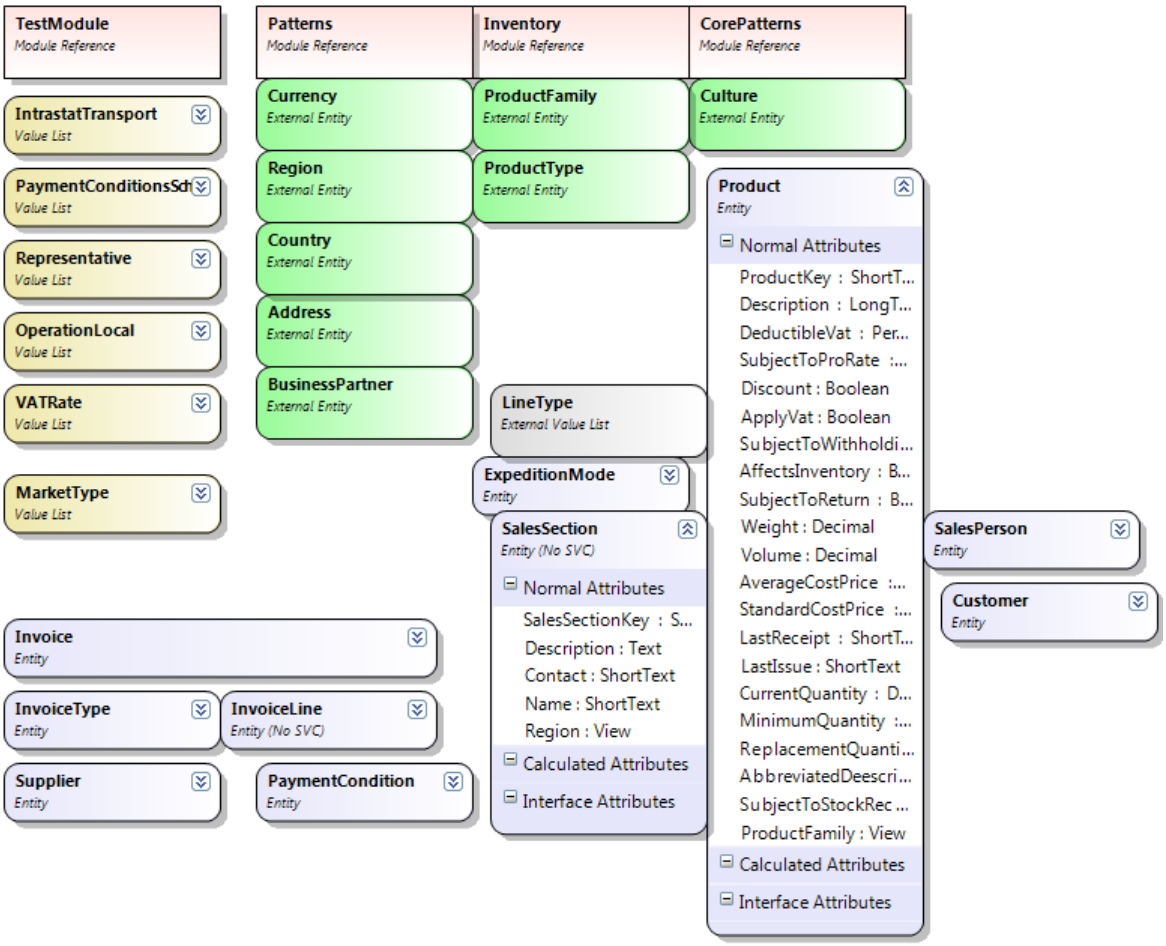

Figure 3.7: TestModule

Chapter 4

Automatic Test Case Generation Component

This chapter will consist on the steps and decisions taken in the development of the test automation component for the Athena Framework, aswell as the strategy used for the success of the project.

4.1 Overview

While modelling a module using the Athena Framework, we can create Entities and their respective Attributes and relations. Internally, they are just C# objects and, in case of the Attributes, there is a mapping between their Athena types and C# types. There is also a structure named Model which is a Graph, that contains all the model information. It is also important to mention that the framework uses T4 Text Templates to generate code. So, in this project was necessary to edit one of those template files in order to generate the Keyword-Driven files for testing automation.

The framework is divided into several DSLs, each one responsible for a specific part in the code generation. Since the goal is to test the UserInterface, it was on the UserInterface DSL that the developed classes were added. It was also necessary to edit one of those T4 Text Template files to use the developed methods.

4.2 Developed Classes

4.2.1 Test Rules

As the name suggests, a Test Rule is used to define rules about an attribute in order to generate data for testing. These rules are simple types of rules like "the attribute A has values between 13 and 30" or "the attribute B text has to have at least 10 characters".Test Rules are composed by the following fields:

- TestRule Types
  - **Minimum/Maximum Value:**
    A Minimum or Maximum rule can be defined to point out what's the minimum or maximum values that an attribute can have according to the requirements. If the attribute that has a rule of this type is numeric, it corresponds to its minimum or maximum value. If its text then corresponds to the minimum or maximum string length.
  - **Range**:
    This type is composed by both minimum and maximum value, and its used to define upper and lower range values for a the attribute. For example, if we have a text attribute that according to the requirements can only have a length between 10 and 20, a Range Rule can be defined for it.
  - **Multiple Ranges:**
    A Multiple Range Rule is defined when an attribute has valid values contained in more than one range. For example, if we have a numeric attribute which has valid values contained in [3,10] and [30,40] this type of rule can be used to define it. Note that this type of rules is not implemented for all attribute types, this is a feature related with future work.
  - **Allowed Values:**
    The Allowed Values rule type is used for attributes that can only have a set of values, p.e, a numeric attribute that can only have the values '1,3,7'.
- Rule Result

  Before pointing out the kinds of rule results it is important to provide a briefly explanation about Intervals and their classification. An interval is said to be left-

bounded or right-bounded if there is some real number that is, respectively, smaller than or larger than all its elements. An interval is said to be bounded if it is both left- and right-bounded; and is said to be unbounded otherwise. Intervals that are bounded at only one end are said to be half-bounded. The empty set is bounded, and the set of all reals is the only interval that is unbounded at both ends. Bounded intervals are also commonly known as finite intervals.[1] Give this, the Rule Result can be seen as a way to define whether or not if the interval is bounded (Pass) or unbounded (Fail). There is also an 'non-conclusive' Test Result which is used for Range or Multiple Range rules since they are dependent of the upper and lower border value.

Test Rules Creation

Test Rules can be added to the model by providing a XML file containing test rules. In case there is no XML file provided, a set of default test rules is automatically generated. The automatically generated rules will be RangeRules, according to Table 4.1.The same thing happens if the XML file lacks test rules for an attribute. The xml file containing the test rules needs to have the format shown on Figure 4.1

Athena Type	Values Range
ShortText	Length:[1,30]
LongText	Length:[1,100]
Text	Length:[1,50]
Memo	Length:[1,400]
Number	[-999999999,999999999]
AutoNumber	[0,999999999]
Decimal	[-999999999.0000,999999999.0000]
Percentage	[-999999999.0000,999999999.0000]

Table 4.1: Athena Framework Type Ranges

4.2.2 Test Data

After Test Rules are created either based on a XML file containing the Rules, it is time to generate the data itself for each attribute in the model, according to their test rules. The test data will contain the actual data values that will be used for testing.

[1] http://en.wikipedia.org/wiki/Interval_mathematics

```
<Entities>
      <Entity
        Name="InvoiceType">
            <Attribute
            Name="Attribute"
            Type="ShortText">
                  <Rule
                  Name="Range"
                  RuleType="Range"
                  Value="1,Pass|20,Fail">
            </Attribute>
            <Attribute
            Name="Description"
            Type="LongText">
                  <Rule
                  Name="Range"
                  RuleType="Range"
                  Value="1,Pass|100,Pass">
            </Attribute>
            (...)
      </Entity>
      (...)
</Entities>
```

Figure 4.1: XML file containing TestRules

The TestData classed is composed by:

- **object data**
  Contains the value of the data for testing. It is declared as a generic object since its type depends on the attribute type (Date, double, int, string , etc).
- **bool expectedResult**
  This member has the result of test of the associated data. If, for examples, the generated value falls under the 'Fail' of the TestRule it will be marked as *false*

Filling this structure requires the help from a developed class named Generate. The Generate class contains the following methods to properly generate data:

- **int RandomNumber(int min, int max)**
  Generates a number between *min* and *max*.
- **double RandomDouble(int min, int max)**
  Generates a decimal number between *min* and *max*.
- **string RandomString(int size)**
  Generates a string with the length equal to *size*.
- **DateTime RandomDate(DateTime a, DateTime b)**
  Generates a date between date *a* and *b*.

With the help of these methods, it is possible to generate values to test border values. By using the rules shown on Table 4.2 we will get the test data displayed on Table 4.3, in the case where the attribute used is a number. The same process is used for generating dates and text, the only things that change are the methods that are used.

#	Rule Type	Value/Result
1	Min	30/Fail
2	Max	40/Pass
3	Range	Min: 10/Pass
		Max: 25/Fail

Table 4.2: TestRule Examples

Rule Used	Value	ExpectedResult
1	29	Fail
	30	Fail
	31	Pass
	RandomInt(31,999999999)	Pass
2	39	Pass
	40	Pass
	41	Fail
	RandomInt(-999999999,40)	Pass
3	10	Pass
	9	Fail
	11	Pass
	25	Fail
	26	Fail
	24	Pass
	RandomInt(11,24)	Pass

Table 4.3: TestData Examples

4.2.3 Qict

Qict[2] is a Library that performs pairwise. As stated before, creating all the possible combinations would result, in some cases, in a very large of test cases. This happens because one of the objectives is to test border values for each attribute in an entity. By default, Qict gets a file with a proper syntax in order to perform pairwise. The syntax for the input is displayed in Figure 4.2. Parameters must follow some rules:

1. They have to be unique, i.e, there can not be more than one parameter with the same name.

2. Parameter name are follows by a colon character and delimited by commas.

Initialization

In the step, the structure that provides support for the algorithm is initialized. The members that are used are the following [McC09]:

[2] http://msdn.microsoft.com/en-us/magazine/ee819137.aspx

```
Param0: a, b
Param1: c, d, e, f
Param2: g, h, i
```

Figure 4.2: Qict input file example

- **int numberParameters**
- **int numberParameterValues** Holds the total number of values of the parameters. For the example shown on Figure 4.2 it has the value of 9.
- **int numberPairs** Holds the all possible pairs between parameters. For the example shown on Figure 4.2 it has the value of 26.
- **int poolSize** The poolSize variable stores the number of candidate test sets to generate for each test set. It was set by default to 20 and was the value used for this project.
- **int[][] legalValues** It is a jagged array where each cell in turn holds an array of int values. The legalValues array holds an in-memory representation of the input file, so cell 0 of legal values holds an array that in turn holds the values 0 (to represent parameter value "a") and 1 (to represent "b"). According to [McC09] working with string values is rather inefficient and that representing parameter values as integers yields significantly faster performance, reason why this strategy was used.
- **string[] parameterValues** It holds the actual parameter values and is used at the end of QICT to display results as strings rather than ints. In the example show on Figure 4.2, cell 0 holds "a", cell 1 holds "b" and so on through cell 8, which holds "i".
- **int[,] allPairsDisplay**

  The *allPairsDisplay* object is a two-dimensional array of ints. It is populated by all possible pairs. In this example, cell [0,0] holds 0 (for "a") and cell [0,1] holds 2 (for "c")—the first possible pair. Cell [1,0] holds 0 and cell [1,1] holds 3 to represent the second pair, (a,d). Table 4.4 clarifies how it works.

- **List<int[]> unusedPairs** The unusedPairs object is a generic List of int arrays. A List collection is used for *unusedPairs* rather than an array because each time a new test set is added to the test sets collection, he pairs generated are removed by the new

i	allPairsDisplay[i,0]	allPairsDisplay[i,1]	Matching Pair
0	0	2	(a, c)
1	0	3	(a, d)
2	0	4	(a, e)
3	0	5	(a, f)
4	1	2	(b, c)
5	1	3	(b, d)
6	1	4	(b, e)
7	1	5	(b, f)
8	0	6	(a, g)
9	0	7	(a, h)
10	0	8	(a, i)
11	1	6	(b, g)
12	1	7	(b, h)
13	1	8	(b, i)
14	2	6	(c, g)
15	2	7	(c, h)
16	2	8	(c, i)
17	3	6	(d, g)
18	3	7	(d, h)
19	3	8	(d, i)
20	4	6	(e, g)
21	4	7	(e, h)
22	4	8	(e, i)
23	5	6	(f, i)
24	5	7	(f, h)
25	5	8	(f, i)

Table 4.4: Mapping of *allPairsDisplay* to the actual pairs

test set from unusedPairs. Additionally, it serves as a convenient stopping condition that will occur when *unusedPairs.Count* reaches 0.

- **int[,] unusedPairsSearch** It is a square array with size numberParameterValues by *numberParameterValues*, where each cell holds a 1 if the corresponding pair has not been used, and a 0 if the corresponding pair has been used or is not a valid pair. For the example in Figure 4.2 it is:

0	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

So row one means pairs (0,0) and (0,1), that is, (a,a) and (a,b) are not valid. On the other hand, pairs (0,2), (0,3),etc, have not yet been used by a test set.

- **int[] parameterPositions** The *parameterPositions* array holds the location within a test set of a specified parameter value. After initialization, and for the same example as before, this array contains the values:

0	0	1	1	1	1	2	2	2

The *parameterPositions* index represents the parameter values(a, b, c, etc) and the cell value the respective parameters (Param0, Param1, Param2)

- **int[] unusedCounts**

The unusedCounts object is a one-dimensional array that holds the number of times a particular parameter value appears in the unusedPairs array. Initially unusedCounts holds:

7	7	5	5	5	5	6	6	6

The index represents a parameter value, and the corresponding cell value is the unused count. This means that we have 7 occurences of 0 (a) and 1 (b) in *unusedPairs*

- **List<int[]> testSets** The testSets object holds the pairwise test set results. It is initially empty but grows every time a new test set is generated. Each test set is represented by an int array.

With the key data structures in place, QICT reads the input file to determine values for *numberParameters* and *numberParameterValues*, and to populate the *legalValues* and *parameterValues arrays*. In order to avoid writing text files for Qict, the method that runs Qict was slightly changed so its input is a Collection of strings in which each string corresponds to a line as it would be in the file. During the transformation process, based on the attribute's test data, this Collection is stored in a member inside the entity and will be called by Qict to perform pairwise.

Once *legalValues* is populated, it is scanned to determine the number of pairs for the input:

```
for (int i = 0; i <= legalValues.Length - 2; ++i) {
  for (int j = i + 1; j <= legalValues.Length - 1; ++j) {
    numberPairs += (legalValues[i].Length * legalValues[j].Length);
  }
}
```

After initialization, the first row of *legalValues* holds {0,1} and the second row holds {2,3,4,5}. Notice that the pairs determined by these two rows are (0,2), (0,3), (0,4), (0,5), (1,2), (1,3), (1,4), and (1,5), and that in general the number of pairs determined by any two rows in *legalValues* is the product of the number of values in the two rows, which equals the row Length property of the rows. The next part of QICT code populates the *unusedPairs* List:

```
unusedPairs = new List<int[]>();
for (int i = 0; i <= legalValues.Length - 2; ++i) {
  for (int j = i + 1; j <= legalValues.Length - 1; ++j) {
    int[] firstRow = legalValues[i];
    int[] secondRow = legalValues[j];
    for (int x = 0; x < firstRow.Length; ++x) {
      for (int y = 0; y < secondRow.Length; ++y) {
        int[] aPair = new int[2];
        aPair[0] = firstRow[x];
        aPair[1] = secondRow[y];
        unusedPairs.Add(aPair);
      }
    }
  }
}
```

Here It is grabbed each pair of rows from *legalValues* using indexes *i* and *j*. Next, we walk through the values in each row pair using indexes *x* and *y*. Extensive use of multiple nested for loops like this is a hallmark of combinatorial code. After populating the *unusedPairs* List, It is used the same nested loop structure to populate the *allPairsDisplay* and *unusedPairsSearch* arrays. The initialization code next populates the parameterPositions array by iterating through *legalValues*:

```
parameterPositions = new int[numberParameterValues];
int k = 0;
for (int i = 0; i < legalValues.Length; ++i) {
  int[] curr = legalValues[i];
  for (int j = 0; j < curr.Length; ++j) {
    parameterPositions[k++] = i;
  }
}
```

The initialization code concludes by populating the unusedCounts array:

```
 unusedCounts = new int[numberParameterValues];
for (int i = 0; i < allPairsDisplay.GetLength(0); ++i) {
  ++unusedCounts[allPairsDisplay[i, 0]];
  ++unusedCounts[allPairsDisplay[i, 1]];
}
```

Here, as in many of the QICT routines, th author takes advantage of the fact that C# automatically initializes all cells in int arrays to 0.The main processing loop begins:

```
testSets = new List<int[]>();
while (unusedPairs.Count > 0) {
  int[][] candidateSets = new int[poolSize][];
  for (int candidate = 0; candidate < poolSize; ++candidate) {
    int[] testSet = new int[numberParameters];
  }
}
```

Because the number of candidate test sets is known to be *poolSize*, we can instantiate an array rather than use a dynamic-sized List object. Notice that the size of the *unusedPairs* collection controls the main processing loop exit. Now it's time to pick the "best" unused pair:

```
int bestWeight = 0;
int indexOfBestPair = 0;
for (int i = 0; i < unusedPairs.Count; ++i) {
  int[] curr = unusedPairs[i];
  int weight = unusedCounts[curr[0]] + unusedCounts[curr[1]];
  if (weight > bestWeight) {
    bestWeight = weight;
    indexOfBestPair = i;
  }
}
```

Here It is defined best to mean the unused pair that has the highest sum of unused individual parameter values. For example, if "a" appears one time in the current list of unused pairs, "b" appears two times, "c" three times and "d" four times, then pair (a,c) has

weight 1 + 3 = 4, and pair (b,d) has weight (b,d) 2 + 4 = 6, so pair (b,d) would be selected over (a,c).

Once the best unused pair has been determined, It is created a two-cell array to hold the pair values and determine the positions within a test set where each value belongs:

```
int[] best = new int[2];
unusedPairs[indexOfBestPair].CopyTo(best, 0);
int firstPos = parameterPositions[best[0]];
int secondPos = parameterPositions[best[1]];
```

At this point we have an empty test set and a pair of values to place in the test set, and we know the location within the test set where the values belong. The next step is to generate parameter values for the remaining positions in the test set. Now, rather than fill the test set positions in some fixed order (from low index to high), it turns out that it is much better to fill the test set in random order. First, It is generated an array that holds the parameter positions in sequential order:

```
int[] ordering = new int[numberParameters];
for (int i = 0; i < numberParameters; ++i)
  ordering[i] = i;
```

Next, It is rearranged the order by placing the known locations of the first two values from the best pair into the first two cells of the ordering array:

```
ordering[0] = firstPos;
ordering[firstPos] = 0;
int t = ordering[1];
ordering[1] = secondPos;
ordering[secondPos] = t;
```

And now we shuffle the remaining positions (from cell 2 and up) using the Knuth shuffle algorithm. This is why It was created a Random object at the beginning of the QICT code. The number of test sets produced by QICT is surprisingly sensitive to the value of the pseudo-random number generator seed value.

```
for (int i = 2; i < ordering.Length; i++) {
  int j = r.Next(i, ordering.Length);
  int temp = ordering[j];
  ordering[j] = ordering[i];
  ordering[i] = temp;
}
```

After shuffling, we place the two values from the best pair into the candidate test set:

```
testSet[firstPos] = best[0];
testSet[secondPos] = best[1];
```

Now we must determine the best parameter values to place in each of the empty test set positions. For each parameter position, It is tested each possible legal value at that position, by counting how many unused pairs in the test value, when combined with the other values already in the test set capture. Then It is selected the parameter value that captures the most unused pairs. The code to do this is the trickiest part of QICT and is listed below:

```
for (int i = 2; i < numberParameters; ++i) {
  int currPos = ordering[i];
  int[] possibleValues = legalValues[currPos];
  int currentCount = 0; int highestCount = 0; int bestJ = 0;
  for (int j = 0; j < possibleValues.Length; ++j) {
    currentCount = 0;
    for (int p = 0; p < i; ++p) {
      int[] candidatePair = new int[] { possibleValues[j], testSet[ordering[p]] };
      if (unusedPairsSearch[candidatePair[0], candidatePair[1]] == 1 ||
        unusedPairsSearch[candidatePair[1], candidatePair[0]] == 1)
        ++currentCount;
    }
    if (currentCount > highestCount) {
      highestCount = currentCount;
      bestJ = j;
    }
  }
  testSet[currPos] = possibleValues[bestJ];
}
```

The outermost loop abbove is a count of the total number of test set positions (given by *numberParameters*), less two (because two spots are used by the best pair). Inside that loop we fetch the position of the current spot to fill by looking into the ordering array I created earlier. The *currentCount* variable holds the number of unused pairs captured by the test parameter value. Notice that because we are filling test set positions in random order, the candidate pair of values can be out of order, so It needed to check two possibilities when we do a lookup into the *unusedPairsSearch* array. At the end of the this code, we will have a candidate test set that has values in every position that were selected using greedy algorithms. Now we simply add this candidate test set into the collection of candidates:

```
candidateSets[candidate] = testSet;
```

At this point we have n = poolSize candidate test sets and It is needed to select the best of these to add into the primary *testSet* result collection. The author states that he could assume that the first candidate test set captures the most unused pairs and could simply iterate through each candidate starting at position 0, but introducing some randomness produces better results. We pick a random spot within the candidates and assume it is the best candidate:

```
int indexOfBestCandidate = r.Next(candidateSets.Length);
int mostPairsCaptured =
  NumberPairsCaptured(candidateSets[indexOfBestCandidate],
    unusedPairsSearch);
```

Here It is used a little helper function named *NumberPairsCaptured()* to determine how many unused pairs are captured by a given test set. The helper function is:

```
static int NumberPairsCaptured(int[] ts, int[,] unusedPairsSearch)
{
  int ans = 0;
  for (int i = 0; i <= ts.Length - 2; ++i) {
    for (int j = i + 1; j <= ts.Length - 1; ++j) {
      if (unusedPairsSearch[ts[i], ts[j]] == 1)
        ++ans;
    }
  }
  return ans;
}
```

Now we walk through each candidate test set, keeping track of the location of the one that captures the most unused pairs:

```
 for (int i = 0; i < candidateSets.Length; ++i) {
  int pairsCaptured = NumberPairsCaptured(candidateSets[i],
    unusedPairsSearch);
  if (pairsCaptured > mostPairsCaptured) {
    mostPairsCaptured = pairsCaptured;
    indexOfBestCandidate = i;
  }
}
```

And now we copy the best candidate test set into the main result testSets List object:

```
 int[] bestTestSet = new int[numberParameters];
candidateSets[indexOfBestCandidate].CopyTo(bestTestSet, 0);
testSets.Add(bestTestSet);
```

At this point, we have generated and added a new test set, so we must update all the data structures that are affected, namely, the *unusedPairs* List (by removing all pairs that are generated by the new test set), the *unusedCounts* array (by decrementing the count for each parameter value in the new test set), and the *unusedPairsSearch* matrix (by flipping the values associated with each pair generated by the new test set from 1 to 0).

We continue generating candidates, selecting the best candidate, adding the best candidate to testSets and updating data structures operations. The processing will end when the

number of unused pairs reaches 0.

Then the final results are displayed:

```
 Console.WriteLine("Result testsets: ");
for (int i = 0; i < testSets.Count; ++i) {
  Console.Write(i.ToString().PadLeft(3) + ": ");
  int[] curr = testSets[i];
  for (int j = 0; j < numberParameters; ++j) {
    Console.Write(parameterValues[curr[j]] + " ");
  }
  Console.WriteLine("");
}
  Console.WriteLine("");
}
```

Running QICT qith the parameters shown on Figure 4.2 will result in the displayed on Table 4.5.

#	Result
0	a c g
1	b c h
2	a d i
3	b e g
4	a f h
5	b f i
6	b d g
7	a e h
8	a c i
9	a d h
10	a e i
11	a f g

Table 4.5: Pairwise Result

4.2.4 Attribute and Entity

Since the framework already had the Entity and Attribute classes, the strategy used was to extend those classes and add them members and properties needed for generate data.

Entity

- **Collection<string> qictData** containing the data to be used as input by Qict
- **Collection<string> qictGridData** same as above but for lists.

Attribute

- **Collection <TestData> testData** containing the Test Data set for the Attribute.
- **Collection <TestRules> testRules** containing the Test Rules set for the Attribute.

4.3 Process

For the sake of clarity, it was created a diagram shown on figure 4.3 with all the required steps for data generation. Also, the following simplified algorithm of the process can aid the understanding of the strategy can be seen on Algorithm 1. All these steps will be fully described in the following subsections.

```
model ⇐ readmodel
if testRules exist then
   rules ⇐ serializeTestRules(model)
end if
for all entity ∈ model do
   attributesList ⇐ getAttributes(entity)
   for all attribute ∈ entity do
      rules ⇐ read(testRules)
      numberOfRules ⇐ Count(rules)
      if numberOfRules = 0 then
         generate default test data for attribute
      end if
   end for
   generate TestData
   run pairwise
   create spreadsheet for entity
   generate test cases for entity
   insert test cases in spreadsheet
end for
```

Algorithm 1: Simplified Process Algorithm

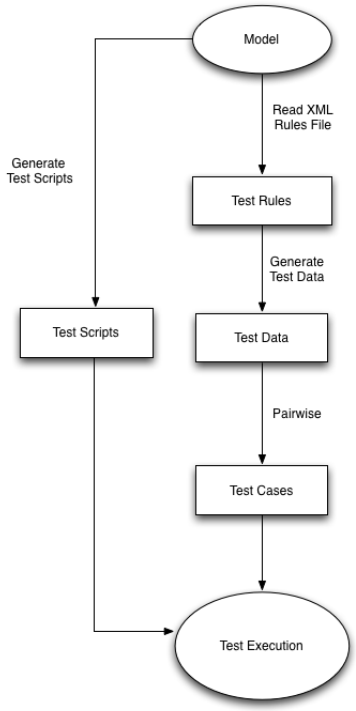

Figure 4.3: Process Diagram

4.3.1 Read Rules File

Using a XML file containing rules, with syntax like shown on Figure 4.1 and using the C# XML library made the reading file process much easier. The strategy used to deserialize rules is displayed on Figure 4.4

4.3.2 Generate Default TestData

By looking again at the Algorithm 1 we can see that when there is no TestRule associated to an attribute a set of TestData is generated, according to the attribute type. This process is shown on Figure 4.5

4.3.3 Generate Values

Now that the TestRules are created, data can be generated based on them. Since we have some special types - List, View, ValueList - they needed to be treated in a different way since their meaning on the User Interface is different from other types. So, they are treated

```
Input: model, filepath

XmlDocument doc = load(filepath)

XmlNodeList entitiesList = get xml elements by tag "Name"
for each (Xml node in entitiesList)
{
	Entity entity = get entity by node "Name" tag
	XmlNodeList attributesXmlList = get xml elements by tag "Attribute"

	for each (XmlNode attributeXml in atributesXmlList)
	{
		Attribute attribute = get Attribute by attributeXml "Name" tag

		XmlNodeList rulesXml leftarrow get Xml elements by tag "Rule"

		for each (XmlNode ruleXml in rulesXml)
		{
			RuleType type = get RuleType by ruleXml tag "Type"

			switch(type)
			{
				case Range:
					Parse RangeRule from ruleXml

				case AllowedValues
					Parse TestRule from ruleXml

				default:
					Parse TestRule from ruleXml
			}
		}
	}
}
```

Figure 4.4: Deserialize Test Rules from XML file

```
Input: model, attribute

switch(attribute type)
{
        case Boolean:
                add  "true"
                add  "false"
        case ValueList:
                get ValueList from model with same name as the attribute
                create TestData with that ValueList as value.
        case DomainType.Datetime:
          case DomainType.Date:
          case DomainType.Time:
                create TestData with minimum and maximum
                        possible dates and a random date
        case DomainType.Percentage:
        case DomainType.Decimal:
                create TestData like it was a RangeRule with min = -999999999.0000
                         and max = 999999999.0000
        case DomainType.ShortText:
                create TestData like it was a RangeRule with min length = 1
                        and max length = 20
        case DomainType.Text:
                create TestData like it was a RangeRule with min length = 1
                        and max length = 50
        case DomainType.Memo:
                create TestData like it was a RangeRule with min length = 1
                        and max length = 400
        case DomainType.LongText:
                create TestData like it was a RangeRule with min  length = 1
                        and max length = 100
        case DomainType.AutoNumber:
                create TestData like it was a RangeRule with min length = 0
                        and max length = 999999999
        case DomainType.Number:
        case DomainType.Money:
                create TestData like it was a RangeRule with min = -999999999
                        and max = 999999999
}
```

Figure 4.5: Generate Default Test Data

as follows:

- **List** - the only TestData this type of Attribute will have is the name of the sheet that it will be related with inside the spreadsheet. Those sheets for lists will be generated in a similar process as the entities spreadsheets.
- **ValueList** - the TestData for the ValueList will only be the values that that ValueList has.
- **View** - they are the last ones that are generated. Since they refer to Foreign Keys it is needed to generate their target entitues first.

```
Input: entity

for each (Attribute attribute in entity)
{
        switch(attribute type)
        {
                case ValueList:
                        for each item in ValueList add that value to the TestData
                case List:
                        add TestData with the name plural name of the target entity
                default:
                        GenerateGenericValues(attribute)
        }
}
```

Figure 4.6: Generate TestData

4.3.4 Pairwise Structure Creation

At this point, we have all the structure set to perform pairwise. Like said on section 4.2.3 the strategy used was to create an Array in which each line has a line according to the syntax used by Qict and store it in the QictData structure, inside an Entity. This method requires a string *value* that contais all the automation ids for the attributes in the entity.The process for this is shown on Figure 4.9. After this structure is created it is just needed to pass it as argument for Qict and pairwise will be performed.

```
Input: attribute

for each (TestRule testRule in attribute)
{
        switch(attribute type)
        {
                case Boolean:
                        GenerateBoolean(testRule)
                case AutoNumber:
                case Number:
                case Money:
                        GenerateNumber(testRule)
                case Decimal
                case Percentage:
                        GenrateDecimal(testRule)
                case Date:
                case DateTime:
                case Time:
                        GenerateDate(testRule)
                case Memo:
                case Text:
                case ShortText:
                case LongText:
                        GenerateText(testRule)
        }
}
```

Figure 4.7: Generate Values

```
Input: testRule

switch (testRule type)
{
        case Range:
                Add random Number/Text/Date with value between min and max
                Add min value
                Add min value + 1
                Add min value - 1
                Add max value
                Add max value - 1
                Add max value +1
        case MaxValue:
                Add max value
                Add max value + 1
                Add max value - 1
                Add min possible value according to attribute type
                Add min possible value -1 according to attribute type
                Add min possible value +1 according to attribute type
                Add random Numer/Text/Date with value
                        between "value" and mininum possible value according
                        to attribute type
        case MinValue:
                Add value
                Add value + 1
                Add value - 1
                Add max possible value according to attribute type
                Add max possible value -1 according to attribute type
                Add max possible value +1 according to attribute type
                Add random Number/Text/Date with value
                        between "value" and maximum possible value according
                        to attribute type
        case AllowedValue:
                Add values from the list
                Add random value that doesn't exist in list
}
```

Figure 4.8: Generate

```
input: atribute, value

string[] automationIdArray = Split(value, ',')

for each (automationId in automationIdArray)
{
        string s = empty string;
        Attribute attribute = GetAttributeByAutomationId(automationId)

        s = s + " : "

        for each (TestData testData in attribute.GetTestData)
        {
                s = s + testData.ToString
        }

        entity.QictData.Add(s)
}
```

Figure 4.9: Generate data for Qict

4.3.5 Create Spreadhseets

After performing Pairwise, the environment is set to create the test cases. First, since it is used a Keyword-Driven approach and all the test cases will be stored in a spreadsheet, those files have to be created. That spreadsheet contains the following fields:

- **Enabled**
  It can be *true* or *false*. It is used to mark the test for running or not.

- **TestType**
  It refers to the type of test that line will run. It can be Insert, Update, Delete, Compare, Exists, NotExists, OpenApplication, CloseApplication.

- **TaskFriendlyName**
  It is used to point the task name for the test. It can be, for example, 'Create Entity-Name'.

- **TaskId**
  It contains the automation id of the task.

- **ListForEdition**
  It has to be *true* in case of the Entity having a List attribute. Otherwise it is set to *false*.

- **FileName**
  Name of the application executable.

- **Arguments**
  URL of the application.

- **Name**
  Name of the window that is running the application

- **Type**
  Type of application. For the case of this project, it is Silverlight.

- **BrowerType**
  The brower used to run the application

- **Username**
  Username for login
- **Password**
  Password for login
- **NewPassword**
  Used in case it is the first time running the application and a new password has to be set.
- **AuthenticationFieldTextBlock** Can be *true* or *false*. Is used in case is the first time running and a new password has to be set.
- **WorkspaceItem**
  Name of the workspace that is used.
- **RememberCredentials**
  Used to mark or not the check box to remember the username and password.
- **RememberWorkspace**
  Used to mark or not the check box to remember the workspace name.

Other than this fields, the sheet will also have the automation ids of each attribute so it knows where to click or fill data. An example of a generated spreadsheet for Entity Supplier contained in the Module developed for testing, can be seen on Figure B.1.

The way to create these spreadsheets is to create and treat them like they are a Database using the Oledb C# library and inserting the Test Cases using SQL queries. So, it is needed to created SQL queries using the data from performing pairwise. This process is illustrated by Figure 4.10

4.4 Running Tests

This section describes an example on how to run a simple test using the generated scripts and the data. The module used to show this example is the one presented on Section 3.5

During the template transformation process the test scripts, that are generated will be run in Visual Studio, are created. As previously said, the ones that are important for this project

```
input: Entity entity
       Graph model
       string operation
       string task
       string url

string[] pairwiseData = Qict.RunQict(entity.QictData)

string tableName = entity.Name  + "TestData"
Collection<string> queries = new Collection()
string taskFriendlyName = task + "Entity" + entity.Name
task = task + entity.Name
bool listForEdition = entity.HasListAttribute

for( int i=1; i < pairwiseData.Length; i++)
{
       string s = new string

       s = "INSERT INTO ["+tableName +"$]"+
                     "VALUES ('FALSE', '" + operation + " ',' " + task +
                     " ',' " + taskFriendlyName + " ','"  +
                      'iexplore.exe' + " ', ' " + url +
                     " ', 'Athena Framework', 'Silverlight', 'admin', 'aa',
                     'False', 'False', 'False', 'False', + transform(pairwiseData)

       queries.Add(s)
}

if(listForEdition)
{
       Entity targetEntity = getTargetEntity(model,entity)

       for each ( string query in entity.GenerateGridInserts(targetEntity)
       {
              queries.Add(query)
       }
}
```

Figure 4.10: Queries Creation

are the View Tests. The goal of these tests is to test the User Interface of the application by inserting, creating, updating, comparing or deleting a record. Since this scripts required the capture of the automatin ids and controls from the brower in order to know the which fields to fill or which buttons to click, they require the help of a library named White Tools[3]. These tests will use the spreadsheets previously generated to run. It was also added to these scripts some methods from an internally developed library named *TestExecutions*. This library allows to store in a database the execution results from the tests aswell as some other important information like the time the test took to run and operative system version.

This example will be based on the first three lines of the spreadsheet shown in Figure B.1 on Appendix B. To run a test we simply need to right-cick on the test on the test viewer window of the Visual Studio and press *Run*. Figure 4.11 shows a small example of how this test will run.

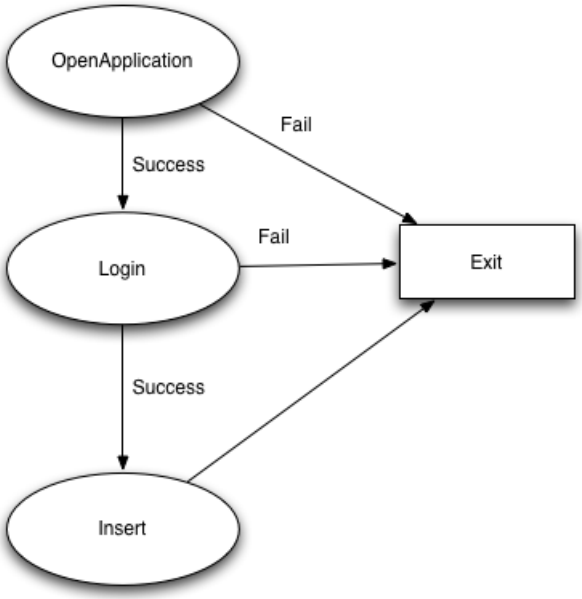

Figure 4.11: Schema for the example

For better understanding, the following shows the values of the fields and their respective values of the third line of the spreadsheet displayed on Figure B.1 which will be used for the example.

[3] http://white.codeplex.com/

4.4.1 Open Application

The first step for running a test will be to open the application. Looking once again at B.1 at the *TestType* field, on the first row is *OpenApplication*. The other fields relevant for openning the application, and their respective values are the following:

- **Filename** set as *iexplore.exe* since is the name of the executable for running the Internet Explorer
- **Arguments** set as *http://localhost:52244/Default.aspx* because it is the URL of the application for testing.

Other than that, the *Name* field is used to know which window on the operative system is used to run the test. In this case, he detects the Internet Explorer window with name *Athena Framework*.

4.4.2 Login

The login process is divided in two parts, the login into the application and the selection of the workspace. The relevant fields for this step are:

- Username
- Password
- RememberCredentials

Login

This step consists on filling the username and password fields in the application's user interface. Figure 4.12 displays how the cells and fields on the spreadsheet act on the user interface.

Select Workspace

The next step will be to select the workspace. Figure 4.13 displays, like before, how the cells and fields on the spreadsheet act on the user interface. The relevant fields for this step are:

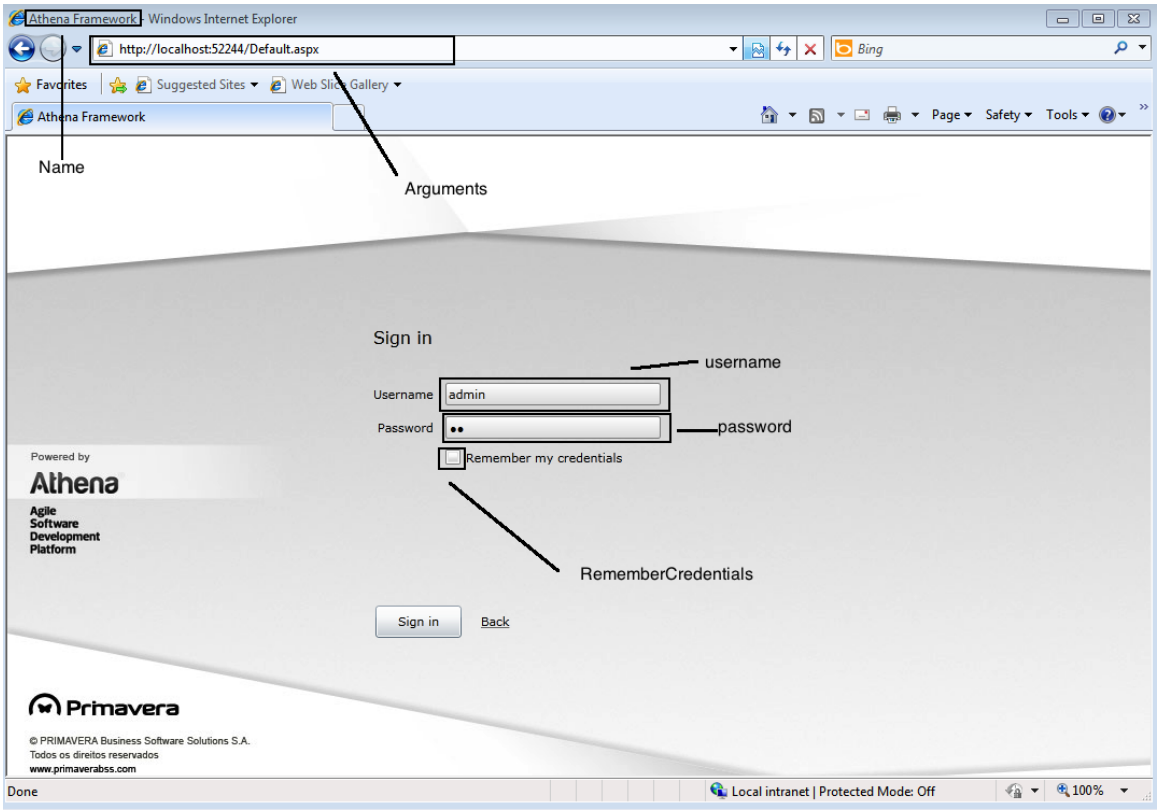

Figure 4.12: Login Screen

- Workspace
- RememberWorkspace

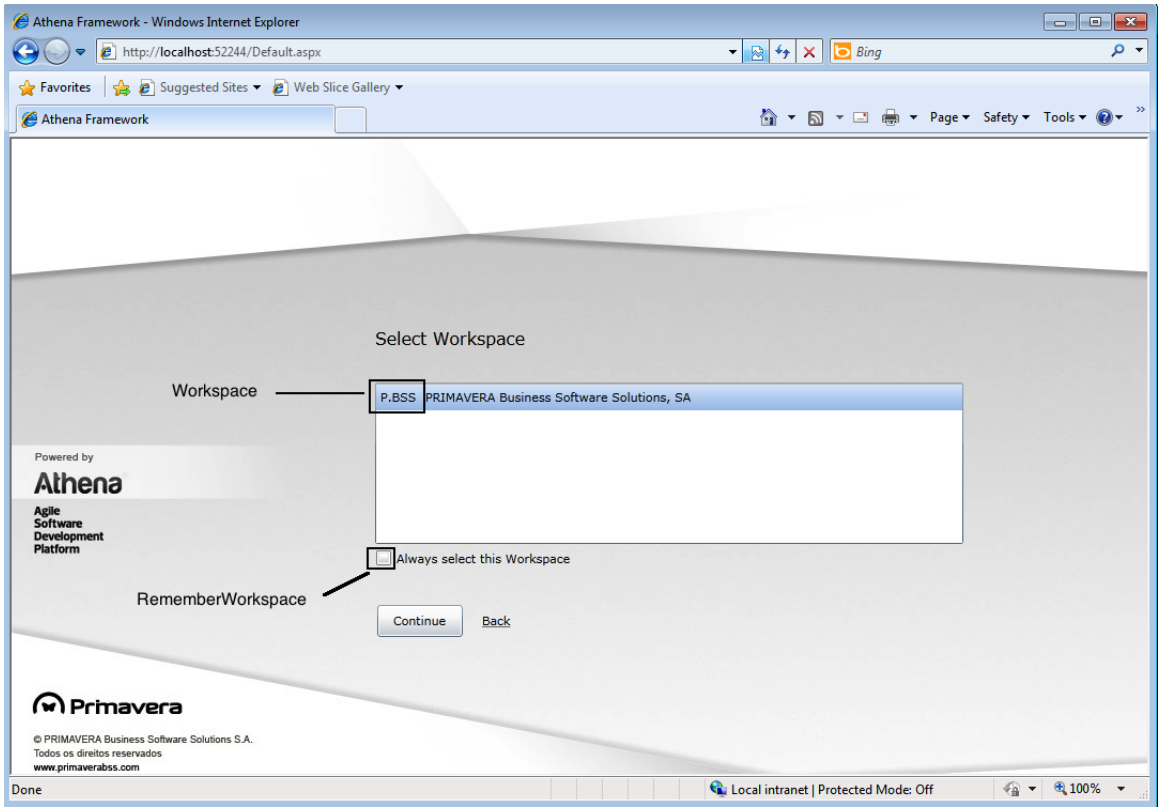

Figure 4.13: Selecting Workspace

4.4.3 Selecting Task

The selecting task step consists in reading the fields *TaskId* and *TaskFriendlyName*. The first one is just a search string while the latter holds the automation id of the operation in the user interface. Figure 4.14 illustrates this.

4.4.4 Insert Values

This step is where the values are inserted in the application. It requires the automation ids of the attributes, which are held by the field name, in this case, *SupplierKeyTextFieldId* , *NameTextFieldId* and *PriceHourNumberFieldId*. Figure 4.15 shows the user interface in

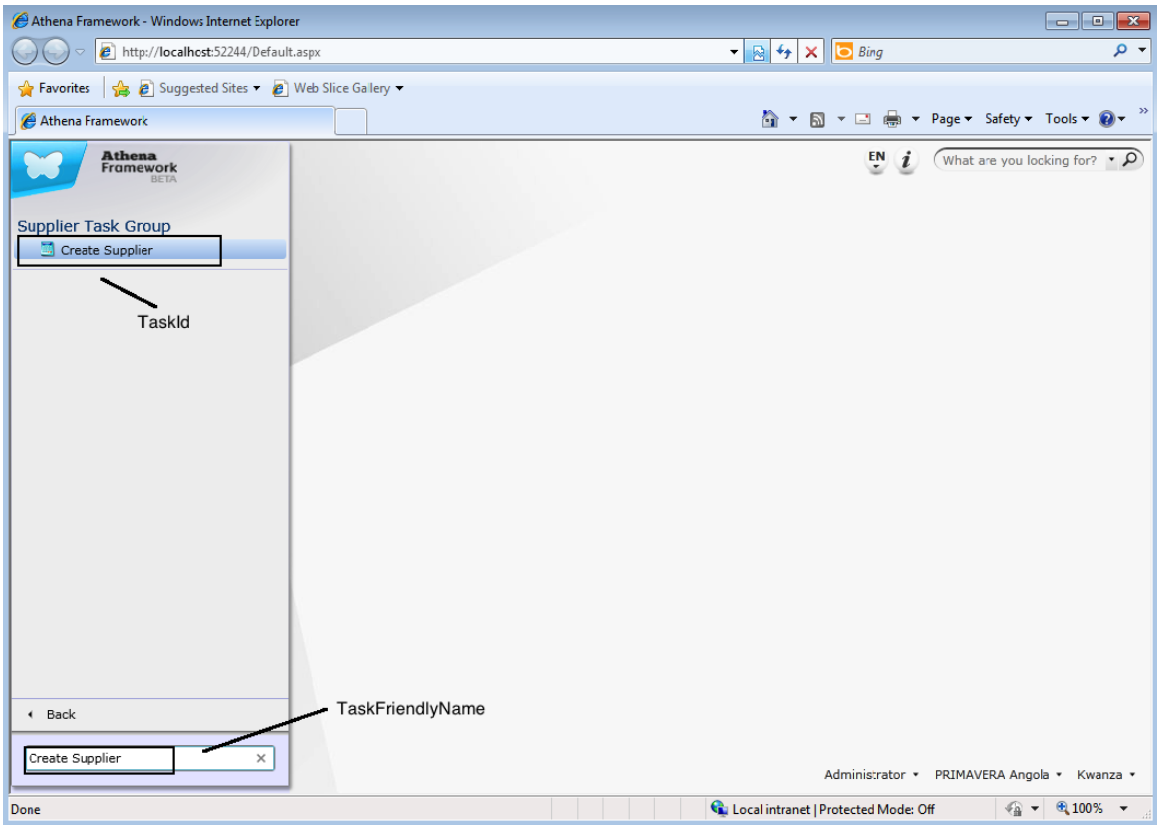

Figure 4.14: Selecting Task

this step. After filling the values the button *Create*, if everything went well, the values are inserted in the application's database. Otherwise, it will pop an error message wich is the trigger for the script to know that something went wrong.

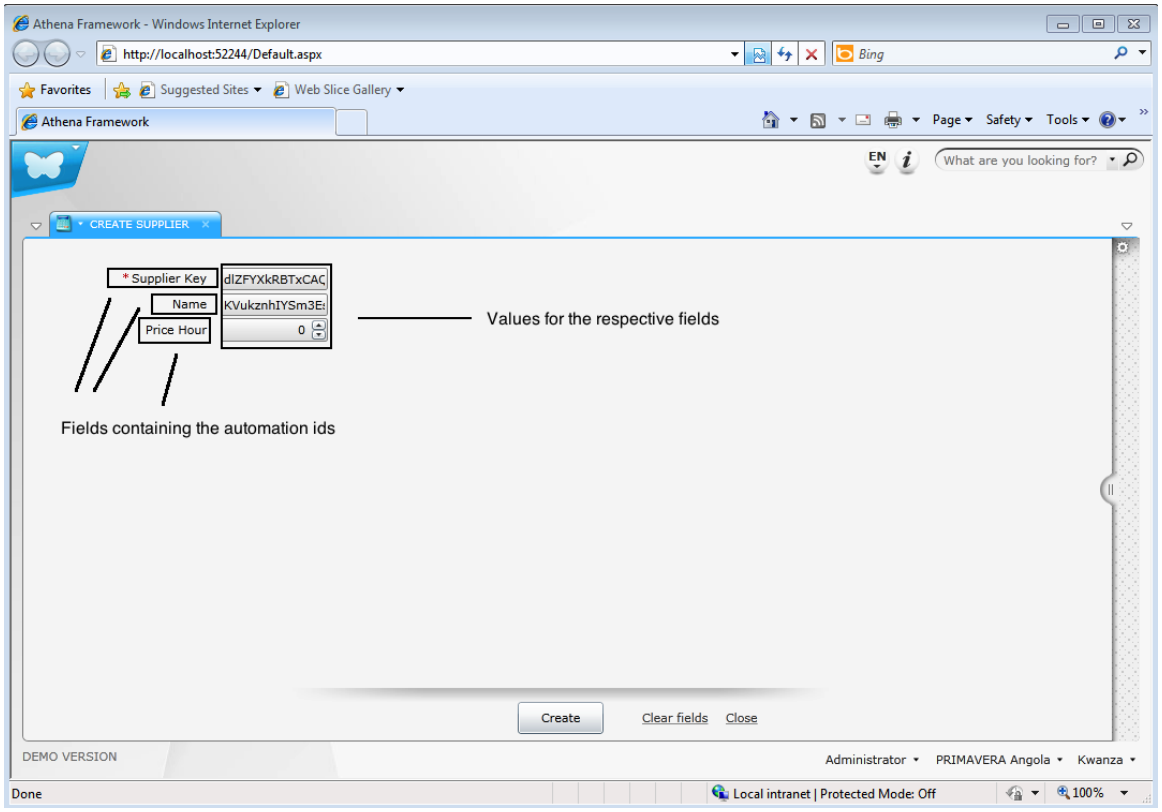

Figure 4.15: Inserting values

4.4.5 Close Application

The last line of the spreadsheet should have a test with type *Close Application*. This signals the script that all the tests are done executing and closes the application and generates a Visual Studio report for the executed tests.

Chapter 5

Concluding Remarks

5.1 Conclusion

This thesis presents the method we defined to add to the *Athena Framework* a module to automatically generate test cases. Overall, we think the project was a success since the main goals of the proposed project are totally achieved. Indeed we are convinced that a major step was given which will save a lot of the testers time and will created a good ammount of test cases that will prevent flaws in the developed applications.

Regarding the difficulties, we can point out that the understanding of the code generation process by the framework was complex. By not having and API or some sort of documentation explaining this process made it very time consuming. Another problem was the framework was still in development and having constant changes on its models and structure which had impact in the testing module. At some point, all the testing related module was not working at all and we were asked to fixed it. One of the things we came up at the end of the development is the huge ammount of negative test cases generated, i.e., the generated test cases are tests that are supposed to fail. It iss easy to understand why this happens. When the border values are generated for an attribute, the set of values which are supposed to fail is bigger than the valid ones. One way to avoid this would be to add a parameter which would add a percentage of test cases that are supposed to pass. One of the project objectives was to build a designer for testing in which the user could define the testrules for the attributes, inside the entities designer. This could not be done since the framework was still in development and it could have too much impact in its development.

5.2 Future Work

While I was developing the project, I thought of some features that could be added in the future in order to enhance the module.

- Add the test designer to the Entities Designer. This would make tester's life, because this way the XML file with the rules would be automatically generated.
- Define a paramenter to create tests that are supposed to pass. This way we could generate more positive tests.
- Add a colum to the spreadsheets which predicts the outcome of the test. This would enhance the test reports on Visual Studio.
- Add the MultipleRanges rule for all Attribute Types.

Bibliography

[AD97] Larry Apfelbaum and John Doyle. Model based testing. *Software Quality Week Conference*, May 1997.

[Ala01] M. Alam. Software test automation myths and facts. June 2001.

[Bac90] James Bach. Test automation snake oil. 1990.

[BBA75] Mahesh Gupta B. B. Agarwal, S. P. Tayal. Software engineering and testing: An introduction. 1975.

[DH07] Adam Kolawa Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. October 2007.

[EFW01] Ibrahim K. El-Far and James A. Whittaker. Model-based software testing. *Encyclopedia on Software Engineering*, 2001.

[Eli10] Elisabeth. Selenium meet-up. 2010.

[Exp08] Testing Experience. Test automation - does it make sense? 2008.

[GdHN+08] Patrice Godefroid, Peli de Halleux, Aditya Nori, Sriram Parajmani, Wolfram Schulte, and Nikolai Tillmann. Automating software testing using program analysis. pages 29--37, 2008.

[Gup09] Yogindernath Gupta. Pros and cons of data driven vs keyword driven automation frameworks. July 2009.

[Han08] Hans. Are test design techniques useful or not? September 2008.

[Hin09] Jeff Hinz. Fifth generation scriptless and advanced test automation technologies. pages 1--18, December 2009.

[Kan03] Cem Kaner. What is a good test case? 2003.

[KKL08] Richard Kuhn, Raghu Kacker, and Yu Lei. Automated combinatorial test methods - beyond pairwise testing. *Problems of Information Transmission*, 2008.

[Kor90] Bogdan Korel. Automated software test data generation. 1990.

[Kuh04] Richard Kuhn. Software fault interactions and implications for software testing. *IEEE Transations on Software Engineering*, 30:418--421, 2004.

[McC09] James McCaffrey. Pairwise testing with qict. *MSDN Magazine*, December 2009.

[Pre05] Alexander Pretschner. Model-based testing. May 2005.

[Puo] Olli-Pekka Puolitavail. Model-based testing tools.

[Tec] Questcon Technologies. Test automation: The promise vs. the reality.

[Utt05] Mark Utting. Position paper: Model-based testing. August 2005.

[vDK04] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, pages 1--17, 2004.

[vDPJ00] A. van DeurSen, P.Klint, and J.Visser. Domain-specific languages: An annotated bibliography. 2000.

[W.H87] W.Howden. *Software engineering and techonology: Functional program testing and analysis*. 1987.

Appendix A

API

A.1 Attribute

A.1.1 Members

Name	Type	Description
rules	Collection<TestRule>	Contains the attribute's TestRules
automationId	string	Contains the AutomationId of the Attribute
testData	Collection<TestData>	Contains the TestData of the Attribute

Table A.1: Attribute Members

A.1.2 Properties

Name	Description
TestRules	Gets the TestRules
AutomationId	Gets or Sets the AutomationId
TestData	Gets or Sets the TestData

Table A.2: Attribute Properties

A.1.3 Public Methods

Name	Arguments	Description
AddTestRule	TestRule rule	Adds the rule to the attribute
AddTestData	TestData testData	Adds the TestData to the attribute
SetDefaultTestData	Model model, Entity entity	Sets the default TestData
SetNumberTestData	int min, int max	Sets the default TestData for Number attribute
SetListTestData	Model model	Sets the default TestData for List attribute
SetDecimalTestData	double min double max	Sets the default TestData for Decimal attribute
SetBooleanTestData		Sets the default TestData for Boolean attribute
SetDateTestData	DateTime min DateTime max	Sets the default TestData for Date attribute
SetTextTestData	int minLength int maxLength	Sets the default TestData for Text Attribute
GenerateViewTestData	Model model	Generate default TestData for View attribute
GenerateBoolean	TestRule testRule	Generate TestData for Boolean attribute based on testRule
GenerateNumber	TestRule testRule	Generate TestData for Number attribute based on testRule
GenerateDecimal	TestRule testRule	Generate TestData for Decimal attribute based on testRule
GenerateDate	TestRule testRule	Generate TestData for Date attribute based on testRule
GenerateGridValue	TestRule testRule	Generate TestData for List attribute based on testRule
GenerateBoolean	TestRule testRule	Generate TestData for Boolean attribute based on testRule
ParseValue	string value	Parses the value

Table A.3: Attribute Public Methods

A.1.4 Private Methods

Name	Arguments	Description
GenerateDateRangeRule	RangeRule rangeRule	Generate TestData for Date attribute based on rangeRule
GenerateDecimalTestData	double min double max	Generate TestData for Decimal attribute based on rangeRule
GenerateTextRangeRule	RangeRule rangeRule	Generate TestData for Text attribute based on rangeRule
GenerateNumberTestData	int min int max	Generate TestData for Number attribute based on rangeRule
GenerateEmbeddedViewRule	Model model Attribute att	Creates TestRule for Embedded View attribute
GenerateValueListRule	Model model Attribute att	Creates TestRule for ValueList attribute

Table A.4: Attribute Private Methods

A.2 Entity

A.2.1 Members

Name	Type	Description
qictData	Collection<string>	Holds the qict data
qictGridData	Collection<string>	Holds the qict data for grids

Table A.5: Entity Members

A.2.2 Properties

Name	Description
QictData	Gets or sets the qictData
QictGridData	Gets or sets the qictGridData

Table A.6: Entity Properties

A.2.3 Public Methods

Name	Arguments	Description
GeneratePairwiseDataForGrid	Entity target string value	Generates the Qict syntax structure for Grid
GeneratePairwiseData	string value	Generates the Qict syntax structure
SetAutomationIds		Sets the automation ids for the attributes
GenerateDefaultTestData	Model model	Generates the default TestData
ReturnRuleValue	RuleType ruleType IList<TestRule>	Returns the rule with the ruleType
GetChildEntityByAttribute	Model model Attribute att	Gets the entity based on the foreign key attribute
GetAttribute	string name	Get attribute by its name
GenrateValue		Generates the values for the entity
GetTargetChildEntities		Gets the child entities of an entity
GetListAttribute		Returns the List type attributes of an entity
HasListAttribute		Returns true if the entity has a List type attribute
GenerateTestCases	string output string task string application string signin string outputFilePath	Generates the test cases

Table A.7: Entity Public Methods

A.2.4 Private Methods

Name	Arguments	Description
EmbeddedViewQictHelper	TestData td Attribute att string automationId	Qict Helper for the EmbeddedView attribute type
DateTimeQictHelper	TestData td Attribute att string automationId	Qict Helper for the Date attribute type
GenericQictHelper	TestData td Attribute att	Qict Helper for the other types
TransformArray	string rawInput	Transforms the array to create Qict syntax
TransformArrayHeader	string[] data	Transforms the array with the headers to create Qict Syntax
GenerateGridInserts	string filePath Entity target	Generates queries for Grids
GenerateValueListValues	Attribute att	Generates values for ValueList attribute type
GenerateEmbeddedViewValues	Attribute att	Generates for EmbeddedView attribute type
GenerateListValues	Attribute att	Generates values for List attribute type
GenerateViewValues	Attribute att	Generates values for View attribute type
GenerateGenericValues	Attribute att	Generates values forValueList attribute
GetAttributeByAutomationId	string automationId	Get attribute by automation id
PluralToSingular	string name	Transforms singular to plural

Table A.8: Entity Private Methods

A.3 TestRule

A.3.1 Members

Name	Type	Description
ruleType	RuleType>	Holds the type of the rule
data	object	The rule value
name	string	The name of the rule
result	RuleResult	If the value should Pass or Fail

Table A.9: TestRule Members

A.3.2 Properties

Name	Description
Name	Gets or sets the name
RuleType	Gets or sets the rule type
Data	Gets or sets the data
Result	Gets or sets the result
DefineRule	Creates a rule

Table A.10: TestRule Properties

A.4 TestData

A.4.1 Members

Name	Type	Description
data	object	Holds the value for testing
expectedResult	bool	Holds the expected result

Table A.11: TestData Members

A.4.2 Properties

Name	Description
Data	Gets or sets the data
Result	Gets or sets the result

Table A.12: TestData Properties

A.5 UserInterfaceTestHelper

A.5.1 Public Methods

Name	Arguments	Description
GetEntityByName	Model model string entityName	Gets Entity by its name
GetEntityByPluralName	Model model string entityName	Gets Entity by its plural name
SerializeTestRules	Model mode string filePath	Serializes the test rules to a XML file
DeserializeTestRules	Model model strin filePath	Deserializes the XML file containing the test rules

Table A.13: UserInterfaceTestHelper Public Methods

A.5.2 Private Methods

Name	Arguments	Description
SerializeRule	XmlWriter writer Attribute att TestRule rule	Serializes a rule
SerializeAllowedValuesRules	Attribute att TestRule rule	Serializes an AllowedValues rule
SerializeDefaultRule	Attribute att TestRule rule	Serializes a rule of the other types
SerializeRangeRule	Attribute att RangeRule rule	Serializes a RangeRule rule
DeserializeRule	XmlNodeList rulesXml Attribute att	Deserializes a rule

Table A.14: UserInterfaceTestHelper Private Methods

A.6 Generate

A.6.1 Public Methods

Name	Arguments	Description
RandomNumber	int valueA int valueB	Generates an int between valueA and valueeB
RandomDouble	double valueA double valueB	Generates a double between valueA and valueB
RandomString	int maxSize	Generates a string with maxSize length
RandomDate	DateTime a DateTime b	Generates a date between a and b

Table A.15: Generate Public Methods

Appendix B

Spreadsheet Example

Figure B.1 shows an example of a generated spreadsheet.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Enabled	TestType	TaskFriendlyName	TaskId	ListFor	FileName	Arguments	Name	Type	BrowserType	Username	Password	NewPa	Authenticationf	Workspaceltem	RememberI	RememberI	SupplierKeyTextFieldId	NameTextFieldId	PriceHourNumberFieldId
2	TRUE	OpenApplication				iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa			P.BSS	FALSE	FALSE			
3	TRUE	Login				iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa			P.BSS	FALSE	FALSE			
4	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	KVukzmhYSm3EaAqBPW	
5	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR	XvJuR8FJmUOOVbetrpP	
6	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	DukxDcISosFeMNNHeTq7	19893018
7	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl		19893018
8	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR	KVukzmhYSm3EaAqBPW	19893018
9	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	XvJuR8FJmUOOVbetrpP	0
10	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	DukxDcISosFeMNNHeTq7	0
11	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR		0
12	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	KVukzmhYSm3EaAqBPW	-1
13	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	XvJuR8FJmUOOVbetrpP	1
14	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR	DukxDcISosFeMNNHeTq7	999999999
15	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a		999999998
16	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	KVukzmhYSm3EaAqBPW	10000000000
17	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	XvJuR8FJmUOOVbetrpP	-1
18	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	DukxDcISosFeMNNHeTq7	1
19	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR	KVukzmhYSm3EaAqBPW	999999998
20	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a		999999999
21	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	XvJuR8FJmUOOVbetrpP	10000000000
22	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR	DukxDcISosFeMNNHeTq7	-1
23	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR		1
24	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	KVukzmhYSm3EaAqBPW	999999999
25	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	dZFYXk8BTcCAQ8JPcnl	XvJuR8FJmUOOVbetrpP	999999998
26	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	qzCB4rTYL7iOPS122TrPR	DukxDcISosFeMNNHeTq7	10000000000
27	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a		-1
28	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a		10000000000
29	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	KVukzmhYSm3EaAqBPW	0
30	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	KVukzmhYSm3EaAqBPW	1
31	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	XvJuR8FJmUOOVbetrpP	19893018
32	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	XvJuR8FJmUOOVbetrpP	999999999
33	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	DukxDcISosFeMNNHeTq7	
34	TRUE	Insert	Create Supplier	CreateEntitySupplier	FALSE	iexplore.exe	http://localhost:5244/Default	Athena Framework	Silverlight	IE	admin	aa		FALSE	P.BSS	FALSE	FALSE	YqB1KENG1huFrr73a	DukxDcISosFeMNNHeTq7	999999998

Figure B.1: Spreadsheet example for Entity Supplier