Korat# with Data Generation:
A Library for Generating Test Structures and Data

by

Karlo Martin Zatylny, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2008

Korat# with Data Generation:
A Library for Generating Test Structures and Data

Approved by
Supervising Committee:

Sarfraz Khurshid

Dewayne Perry

Dedication

For my wife and kids.

Acknowledgements

I would like to acknowledge the help of my supervising professor, Dr. Sarfraz Khurshid, for teaching me about the original Korat project and helping me understand the fun and complexity of software testing.

December 2008

Abstract

Korat# with Data Generation:
A Library for Generating Test Structures and Data

Karlo Martin Zatylny, MSE

The University of Texas at Austin, 2008

Supervisor: Sarfraz Khurshid

This paper introduces Korat#, and the process of translating the original Java written Korat to a C# equivalent. Although many C-based languages share common syntax, the different run-time environments, compilers, and coding styles provide significant challenges when translating a program from one language to another. Moreover, as languages continue to enhance syntactical features, once similar languages become increasingly dissimilar. In this paper, we discuss the translation of Java written Korat, a test generation tool, to C#. In addition to a direct translation, the new .NET version of the Korat software enhances the current feature set with a regular expression data-generation engine to provide valid string-based input for the given methods and libraries. Many production applications written in .NET and other platforms are

increasingly dependent on string-based data like XML and regular expression based inputs. Hence, the auto-population of test data with random but rule obeying strings is extremely beneficial to the practice of test generation implementations. This paper explains the implementation of the regular-expression string-generator and its cohesion in Koart#.

Table of Contents

List of Tables

List of Figures

List of Illustrations

1.0 INTRODUCTION

Many times useful applications are written in a language outside of a developer's current expertise or environment. Sometimes, an application in a given language may target specific environments or technologies that make it difficult to integrate into another existing infrastructure. In these cases, there are often valid reasons for translating the program into a new target environment for a development team. The program, consequently, becomes more usable and maintainable by the currently available talent.

The Korat[1] project, written in Java, provides a powerful way for the instrumentation of existing compiled code for generating test cases far beyond that of manually written unit tests. Due to the intimate nature of the program with the Java environment, a translated .NET version seemed like a valid step towards using this tool in a completely different environment. Korat manipulates compiled Java code, dynamically creates new classes, and uses the Java Modeling Language (JML)[5] to identify pre and post method validation for tests. These Java specific implementation factors make the translation into another language a better option than trying to retrofit the existing Java code to manipulate some other environments library files and compilation methodologies.

Moreover, many existing applications in the industry rely heavily on performing operations on string-based data. XML is abundant throughout many applications and is a de facto standard for almost all web-based applications. Moreover, many software applications store data to a database in string format (SQL varchar or text). These strings often need to obey certain rules for format and can commonly be verified by regular expressions or XML validating methods. In fact, many validation frameworks, like Microsoft's Validation Enterprise Library, are written to instrument code in such a way that the framework can automatically verify the string input without having to burden the code with repeated validation checks. Since Korat generates non-isomorphic data

structures for testing the methods of a class, the addition of a wider variety of input data could yield more realistic test cases for applications that are heavily string dependent. This paper explains the methodology and implementation of taking a regular expression and generating useful test data to populate the Korat data structures with. The resulting data structures and valid data should provide a more robust set of automatically generated test cases.

This report contributes the following items to the computer science community. First, this report explains the implementation of Korat in C# for use in the .NET community. Next, the report gives the implementation of a string generation utility for generating data from regular expressions. Finally, this report outlines the combination of the Korat in C# implementation with the data structure population from the data generation utility.

2.0 CODE TRANSLATION

The movement from Java to C# seems mostly superficial on the surface as both languages are grounded in C and obey most of the same logic-based syntax tokens. In order to exhaust all options, a variety of attempts to translate the code via different methodologies was done to try and minimize the amount of manual coding necessary.

The upfront limitations of any automatic code translation tool are quite obvious when translating a project of code such as Korat. Korat directly manipulates the Java byte code of the compiled Java libraries. These byte code insertions are very specific to Java and are most certainly not easily translated to any other language or run time, even those based on a virtual machine. Korat also generates new Java classes via embedded strings and formatting application programming interfaces. All of this is done with an open source Java library called *Javassist*[10]. The code using this library is not easily translated by some automated tool. Thirdly, Korat interoperates with JML, which is a markup language and engine specifically geared at the instrumentation and testing of Java libraries. Hence, any direct reference to this code would not be easily translated by some automated tool. Also, when some third party library outside of the defined Java libraries is used in a project, an automatic tool usually does not have intimate knowledge or syntactical equivalents for these pieces of code.

2.1 Microsoft JLCA

Microsoft provides the Java Language Conversion Assistant (JLCA)[2] in Visual Studio 2005. This tool takes the root directory of a project of Java source code files and provides a best attempt at a complete translation. The tool also outputs developer-friendly comments into the resulting C# code to notify the programmer about possible behavioral differences due to language syntax, and also notifies the developer when

certain methods cannot be effectively translated automatically. JLCA will also attempt to write additional helper classes to substitute in for common Java classes, such that some of the original syntax from the Java source files remains intact.

After running JLCA on the Korat code structure, approximately 1000 warnings and errors existed in the resulting C# code base. Most of the errors were in the expected sections of the byte code injection and the Javassist code generation pieces, but many other pieces were not translated as well. Java allows wild card generic class definition with the insertion of a ? inside of the generic < > definition. JLCA had no C# equivalent mapped for this syntax structure. Also, the Java Collections library has a rich set of classes including `Set`, `HashSet`, `HashMap`, `Tree`, `BinaryTree`, and `List` which do not always behave the same as the .NET equivalents, or simply have no library equivalent in the default .NET libraries. Hence, many places where these data structures existed in the Korat code, the JLCA was unable to reliably duplicate the behavior of the original code in the output C#. JLCA did successfully translate most of the Korat interfaces and base classes as would be expected.

2.2 Search and Replace with Macros

Another attempt to translate the code was performed using search and replace macros. Since Korat heavily uses interfaces and is mainly C-like syntax structures throughout the code base, an attempt was made to simply rename all the .java files to .cs, add the resulting file set to a C# Visual Studio project and utilize the search and replace capabilities of the IDE to simply replace Java specific syntax with C# specific syntax. In many cases, the IDE's regular expression search and replace functionality was used to identify more complex replacements. The following table outlines most of the major search and replaces performed.

Java Syntax	C# Syntax
import	using
package	namespace
throws Exception	
final	static, sealed
.class	.GetType()
super	base
<?>	<object>
implements	:
extends	:
for (:)	foreach (in)

After the initial known map of search and replaces were performed, the C# compiler was used to help identify the next set of syntactical violations. Most of these violations were class names not in the .NET libraries. The following map of classes was using in the translation:

Java Class	C# Class
HashSet	Dictionary< >
Set	List< >
HashMap	Dictionary< >
boolean	bool

This translation effort provided much of the necessary translation, similar to the effort of the JLCA. However, unlike the JLCA that would omit code it could not translate. The developer was able to comment out pieces that would not compile, place an easily found comment for later location, and move on to easier issues.

Part of the program translation process is for the developer to understand how the implementation is meant to function. Having a developer manually translate the more difficult parts of the code is usually inevitable and aids in the necessary familiarization of the code base. In complex programs the overall time spent translating may grow if done manually, but aids in the long term support of the project as the developer will be more

intimate with the entire code base and the relationship of classes created by the initial developer.

2.3 Fresh Implementation

After spending less than a man-day on the manual translation, the direction of the translation was changed. The translator had become familiar enough with the Java implementation that using the actual Java code as a guide, a start-from-scratch methodology could be more easily performed. Moreover, since Java and C# share the most basic fundamental value types and interface definitions, a bottom up approach was ultimately utilized.

Starting with just a sub-set of the Java implementation's translated C# interface classes, a new C# project was created that was almost always compile ready. Thus, the initial framework of object relationships was established with the framework of Korat interfaces. From this compilation point, individual class files or groups of files were added and translated until the project was again at a fully compiled state.

The Korat Java project is divided into seven major folders. Each folder represents a major piece of functionality within the Korat program. The Config folder contains the classes necessary for the initial reading of command line parameters and population of the global configuration of the Korat program for a given execution cycle. The Finitization folder contains all the files necessary for describing the structures necessary for the finitization definitions. The Gui folder contains the files necessary for some of the visualization and output of Korat. The Instrumentation folder contains the necessary classes for the byte code and code generation pieces. The Loading folder contains the necessary classes for loading individual portions of a package for instrumentation. The Testing folder contains the classes necessary for determining the test cases as well as a

set of custom Exception classes. The Utils folder contains a set of classes to help facilitate input/output, reflection, and a set of classes for dealing with the candidate vectors.

The simplest classes to first translate were the finitization classes. This set of classes is mainly custom classes defined around value types. Most of these classes were easily translated to C# with minimal changes.

The config folder had the main configuration structure, which was mainly value types, and a class to read the input from the command line to populate the configuration structure. This piece had to be completely rewritten to use C# centric methodologies of reading configurations. To facilitate the most common .NET configuration method, a configuration file was added to the solution in order to provide an XML based way for all the configurations to be defined and manipulated by native .NET classes.

The instrumentation folder was where the most custom .NET code was required. The code injection and class generation code was completely rewritten from the code using Javassist to code using the .NET System.Reflection library, the Mono.Cecil library, and the .NET System.CodeDom library. The translation is explained in more detail later in the paper.

The loading folder also required many changes as the code necessary for loading and inspecting .NET code varies greatly from that of Java compiled libraries. Many of the JLCA translated classes provided valuable insight as far as mapping classes and technologies between Java and .NET reflection.

The gui folder was completely rewritten to use .NET WPF as the main user interface implementation for the Korat# library. These changes along with some syntactical uses

in the translation make it necessary for Korat# to be compiled and run with .NET 3.5. However, a .NET 2.0 version is also available without the UI portion.

The testing folder was easily translated as far as the interfaces and Exception classes were concerned. Similar structures exist in Java and .NET for defining, constructing, and throwing exceptions. The test classes required more intimate translation to take care of certain Java specific syntax issues. Also, some pieces of the testing folder used class creation methodologies that required use of the .NET System.CodeDom library.

The utils folder was also a place of heavy manual translation as Reflection, IO, and collections are not easily mapped between the two languages in a way that an experience .NET programmer would write the code.

2.4 Language Standards

Translating code from one language to another often requires the translator to learn the existing implementation or already be familiar with the source languages common practices in order to semantically translate those statements and algorithms to the common practices of the target language. Java and C# have a few syntax differences that have formed different coding practices in each language. The most common difference is the use of fields with `getField` and `setField` methods in Java versus a private field with a public property that exposes a `get` and `set` method. Although, semantically equivalent in most cases, the most common .NET practices have the implementer use private variables exposed by public property accessors. This common programming practice was taken into consideration both in the syntactical translation of Korat as well as in the code injection portion of the implementation.

Korat injects code into classes to determine when a field is accessed by the method (commonly called `RepOk`) so that it can determine how to properly structure the candidate vector and thereby generate the non-isomorphic test cases for a given class. Since many C# developers at a minimum expose the get method of a property for a private field, the .NET version of Korat was easily able to utilize code injection into the get methods of the properties in order to determine the field access and the candidate vector generation. This proves easier to implement in C# while not sacrificing the overall functionality of Korat. As such, the standard for writing `RepOk` methods in C# must utilize the get methods of the fields validated in order to facilitate the IL code injection usability.

3.0 CODE INJECTION

The IL code injection method had three potential candidates for the C# implementation. The first was simply using the .NET System.Reflection library. This library provides an API to learn about the classes, find the methods to instrument and finally instrument the code. This was quickly dismissed as the best option as the other code injection libraries had already built-in helper functions for the code injection and was able to load a .NET library without loading the dependencies of the library.

RAIL[3], the Runtime Assembly Instrumentation Library, provided a way to load assemblies without their dependencies and inject IL code into the library and save the newly instrumented library to a different location. Moreover, the library contained many helper methods for easy instrumentation of code at the beginning or end of each method, code copying, code redirection, and a handful more ways of manipulating the compiled version of a .NET library. Familiarization of the RAIL library showed that it was in turn using a Mono library, specifically the Mono.PEToolkit. Hence, an investigation to the Mono libraries was conducted.

The Mono.Cecil[4] library was identified as the implementation of choice as it had all the advantages of RAIL with a slightly easier to learn API and some added flexibility. The strategy for injecting the necessary code for Korat# was done with the following algorithm:

1. For each public concrete class in the target library.
2. If the class has the `RepOk` and `Finitization` methods.
3. Locate all the properties with publically visible get methods.
4. Insert the necessary IL code into the beginning of the get methods to call back to Korat such that Korat knows the field is being accessed.

5. Save the resulting library to a new file.

This instrumentation was most easily accomplished by having an external library that contained a static class that could be called with the proper notifying parameters and a public event that Korat could attach to at test generation time for notification of field access. Each property's get method which started as this:

```
public int MyProperty
{
    get { return myField; }
}
```

Becomes instrumented like this:

```
public int MyProperty
{
    get
    {
        NotifyPropertyAccess.PropertyAccessed(this, "MyProperty");
        return myField;
    }
}
```

This simple instrumentation gives Korat# the ability to track where the failure of the `RepOk` method occurs, thus allowing manipulation of the input structure to be changed as to generate a valid input for the classes' methods.

4.0 CODE GENERATION

The code generation methodology was easily accomplished by the .NET System.CodeDom library. This library provides API based abstractions of common code constructs for creating types and all of the internal pieces of a type. Moreover, the flexibility to simply add a string of code in a method leaves the ultimate flexibility to the implementer. The library was more than complete to provide Korat# with the necessary functionality to generate the necessary test classes for test case generation.

In addition, the CodeDom library also provides the ability to save the resulting classes in text .cs files, or compile the set of new generated classes into a dynamic runtime library that can be saved or loaded into the current AppDomain as an ad hoc piece of functionality.

5.0 REGULAR EXPRESSION DATA

In order to facilitate the data generation of strings in the Korat# solution, a new section of code was added to aid in the interpretation and generation of string data. An attribute could be added to a testable class' public string property such that this attribute would contain a validation string in the format of a regular expression. If a property is of the type string and contains the `RegExGenerator` attribute, the Korat# library detects the string parameter to the attribute, and generates a set of strings that conforms to the given regular expression.

For example, if a property was attributed with the regular expression "a+", the resulting set of candidate strings might contain the strings: "a", "aa", "aaa", "aaaa", and so on. When the time comes in Korat# for a string property to get populated with a value, the Korat# Regular Expression Engine randomly selects a string from the regular expression set and inserts it into the structure.

5.1 Implementation

Since no native .NET class provides this functionality, this code is new to both the Korat family and is a unique implementation for the .NET community. The original code for parsing a regular expression from a string to a NFA and then to a DFA was originally written in C++ by Amer Gerzic[9]. Given a regular expression, the Korat# Regular Expression Data Generator behaves in the following way:

1. Parse the regular expression to a Nondeterministic Finite Automata (NFA).
2. Translate the NFA to a Deterministic Finite Automata (DFA).
3. Given the DFA, walk a deterministic number of the branches in order to generate a set of valid strings in a finite amount of time.

To reliably generate this data, the DFA structure has some special properties and methods along with a custom input structure that monitors the traversal of the DFA graph and gives terminating cases in the case of a cyclic graph. The basic structure of the DFA state contains:

```
namespace RegExFinal
{
    public class KoratFAState
    {
        //! Transitions from this state to other
        /// <summary>
        /// A map of characters to the next DFA(NFA) state(s) in
        /// the graph.
        /// </summary>
        public Dictionary<char, List<KoratFAState>> m_Transition;

        //! True if this state is accepting state
        /// <summary>
        /// A flag to determine if this state is a terminating state
        /// </summary>
        public bool m_bAcceptingState;
    }
}
```

The token structure that is the input to the generate strings method is defined with the following properties:

```
namespace RegExFinal
{
    public class DataToken
    {
        /// <summary>
        /// Determines the maximum total DFA states the engine can
        /// visit before searching for a terminating state.
        /// </summary>
        public int MaximumLength

        /// <summary>
        /// Determines the maximum number of times a given DFA state
        /// can be visited before the option will try to find a
        /// terminating state.
        /// </summary>
        public int MaximumVistsPerState
```

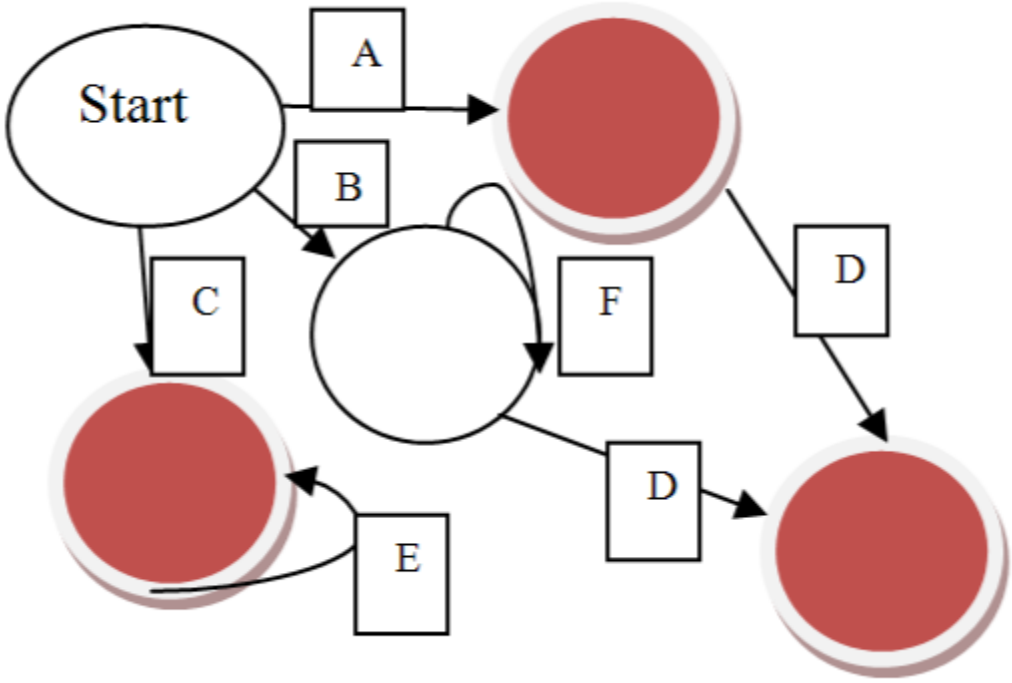
```
        /// <summary>
        /// determines the maximum number of paths allowed to be taken
        /// from any one state to the following state.  For example, if
        /// this value is set to 2 and a given DFA state has 3 paths,
        /// only two will be randomly selected to take.
        /// </summary>
        public int MaximumForksPerState

        /// <summary>
        /// the set of strings that is the generated set of valid
        /// strings found to any given point.
        /// </summary>
        public Set<string> GeneratedData

        /// <summary>
        /// The list of visited states, in order, of the current
        /// matching string.
        /// </summary>
        public PathTracker VisitedStates

        /// <summary>
        /// The string that matches the path as it has been traversed
        /// to this given state.
        /// </summary>
        public string CurrentInput
    }
}
```

Given this terminating structure, any cyclic DFA will eventually try to seek a state where `IsFinal` is true or in some cases will abandon the current path. The forking methodology of walking a maximum number of edges leaving a given node may be a place to improve the existing code such that given a regular expression; an optimal branching strategy can also be input.

Whenever a final DFA state is reached, the string to that point is added to the data set so that multiple terminating states can exists on the path and each will be given representative string candidates in the result.

For example, given the following simple DFA state graph and terminating token values, the final given set of strings would be in the table.

```
MaxNumPaths = 3, MaxNumVisits = 10
```



A	BD	BFD	BFFD
BF{3}D	BF{4}D	BF{5}D	BF{6}D
BF{7}D	BF{8}D	BF{9}D	BF{10}D
C	CE	CEE	CEEE
CEEEE	CEEEEE	CE{6}	CE{7}
CE{8}	CE{9}	CE{10}	AD

For brevity sake, the shorthand X{#} gets interpreted as the letter preceding the {#} is repeated # times in the string.

One way to write the regular expression for this DFA might be:
(AD?|BF*D|CE*)

Given this set of strings, when a property with this regular expression needs to be populated, the calculation to select an item from the result array is done via a randomly

generated double value between 0.0 and 1.0 multiplied by the number of elements in the array minus one, and converted to an integer index.

5.2 Sample Generation

A sample data structure and the corresponding generated strings are outlined in the following code segments:

```
namespace KoratExamples
{
    public class Address : IComparable
    {
        [RegExGenerator("(1|2|3|4|5|6|7|8|9)(1|2|3|4|5|6|7|8|9|0)?
(Main|Center) (Street|Ave)")]
        public string Address1 { get; set; }

        [RegExGenerator("APT (1|2|3|4|5|6|7|8|9)(a|b|c)")]
        public string Address2 { get; set; }

        [RegExGenerator("(Springfield|Jackson)")]
        public string Locality { get; set; }

        [RegExGenerator("(CA|AZ|TX|WA)")]
        public string Region { get; set; }

        [RegExGenerator(
"90(1|2|3|4|5|6|7|8|9|0)(1|2|3|4|5|6|7|8|9|0)(1|2|3|4|5|6|7|8|9|0)")]
        public string PostalCode { get; set; }

        [RegExGenerator("(US|USA|United States)")]
        public string Country { get; set; }
    }
}
```

For each of the given regular expressions, a given set of strings gets generated by the engine. A sample output from the data engine would consist of the following assignments:

```
Address address = new Address()
{
    Address1 = "3 Main Street",
    Address2 = "APT 7b",
    Locality = "Jackson",
```

```
    Region = "AZ",
    PostalCode = "90967",
    Country = "USA"
};
```

While the generated strings are not an actual address in this case, the generated data is valid enough for running test cases against the `Address` class. Obviously, more exact and complete regular expressions lead to more accurate generated data.

5.3 Optimizing the DFA

Once a DFA is created with this algorithm, there is a high probability that the DFA will have many similar states. A similar state is defined as a state that matches another state in the DFA in both its transition attributes and the acceptance attribute. For example, many of the DFAs created by this algorithm will create many terminal states that have no outgoing transitions and are marked as an acceptance state. The aim of an optimization stage of the processing would be to reduce the number of duplicate states such that the memory footprint of the resulting DFA is as small as possible. This stage also will optimize semantically similar regular expressions that syntactically have resulted in differing DFAs.

For example, if we are to create a regular expression for a byte of an IPv4 address, we will want the string results of the numbers 0 through 255 without the leading zero[s] on any numbers. To do this, we could use the following regular expression:

```
((1?(1|2|3|4|5|6|7|8|9)?(0|1|2|3|4|5|6|7|8|9))|(10(0|1|2|3|
4|5|6|7|8|9))|2(((0|1|2|3|4)(0|1|2|3|4|5|6|7|8|9)|5(0|1|2|3
|4|5))))
```

This regular expression leads to a DFA with the following state table.

State ID	Transition Character	Destination State	Is Acceptance State
1	1	2	False
	2	3	
	3	4	
	4	5	
	5	6	
	6	7	
	7	8	
	8	9	
	9	10	
	0	11	
2	1	43	True
	2	44	
	3	4	
	4	5	
	5	6	
	6	7	
	7	8	
	8	9	
	9	10	
	0	45	
3	1	21	True
	2	22	
	3	23	
	4	24	

State ID	Transition Character	Destination State	Is Acceptance State
	5	25	
	6	17	
	7	18	
	8	19	
	9	20	
	0	26	
4 – 10, 43, 44	1	12	True
	2	13	
	3	14	
	4	15	
	5	16	
	6	17	
	7	18	
	8	19	
	9	20	
	0	11	
11 – 20, 27 – 42, 46 - 55			True
21-24, 26	1	27	True
	2	28	
	3	29	
	4	30	
	5	31	
	6	32	
	7	33	
	8	34	

State ID	Transition Character	Destination State	Is Acceptance State
	9	35	
	0	36	
25	1	37	True
	2	38	
	3	39	
	4	40	
	5	41	
	0	42	
45	1	46	True
	2	47	
	3	48	
	4	49	
	5	50	
	6	51	
	7	52	
	8	53	
	9	54	
	0	55	

The layout of this table clearly shows the duplicate states within the DFA. Hence, an algorithm is needed to set equivalent states into one merged state and replace all transitions to the merged state for the entire DFA.

The algorithm chosen to do this was a nested breadth first search of the entire graph. The following code outlines the general algorithm:

```
public void ReduceDfaIdenticalStates()
```

```
{
    if (m_DFATable != null)
    {
        bool foundIdenticalStates = false;
        do
        {
            Dictionary<int, Dictionary<int, KoratFAState>> equalStates
            = new Dictionary<int, Dictionary<int, KoratFAState>>();
            LinkedListNode<KoratFAState> state = m_DFATable.First;
            Queue<KoratFAState> removeQueue =
                                new Queue<KoratFAState>();
            while (state != null)
            {
                LinkedListNode<KoratFAState> loop = m_DFATable.First;
                int stateId = state.Value.GetIntStateID();
                while (loop != null)
                {
                    int loopId = loop.Value.GetIntStateID();
                    if (stateId != loopId &&
                        state.Value.DfaStatesAreEqual(loop.Value))
                    {
                        int primary = stateId;
                        int secondary = loopId;
                        KoratFAState primaryState = state.Value;
                        KoratFAState secondaryState = loop.Value;
                        if (equalStates.ContainsKey(loopId))
                        {
                            primary = loopId;
                            secondary = stateId;
                            primaryState = loop.Value;
                            secondaryState = state.Value;
                        }
                        if (!equalStates.ContainsKey(primary))
                        {
                            equalStates[primary] =
                                  new Dictionary<int, KoratFAState>();
                        }
                        if
                        (!equalStates[primary].ContainsKey(secondary))
                        {
                            equalStates[primary][secondary] =
                                                    primaryState;
                        }
                        if (!removeQueue.Contains(secondaryState))
                        {
                            removeQueue.Enqueue(secondaryState);
                        }
                    }
                    loop = loop.Next;
                }
                state = state.Next;
```

```
            }

            foreach (KeyValuePair<int, Dictionary<int, KoratFAState>>
                     item in equalStates)
            {
                foreach (KeyValuePair<int, KoratFAState> stateItem in
                         item.Value)
                {
                    state = m_DFATable.First;
                    while (state != null)
                    {
                        state.Value.ReplaceStateTransition(
                        stateItem.Key, stateItem.Value);
                        state = state.Next;
                    }
                }
            }
            foundIdenticalStates = removeQueue.Count > 0;
            while (removeQueue.Count > 0)
            {
                m_DFATable.Remove(removeQueue.Dequeue());
            }
        } while (foundIdenticalStates);
    }
}
```

Given the structure of the DFA, this algorithm favors states with lower IDs. After one iteration of this algorithm, the DFA is significantly reduced in the total number of states from the original, but still contains some duplicate states. Another iteration of the reduction is required to eliminate this new set of duplicates. Hence, we encapsulate the reduction algorithm in a `while` loop that checks that we have some states to delete at the end of the algorithm. If there was at least one state to delete, then there is a chance that further reduction is possible, so we need to check the DFA again. The final state table for the DFA is as follows:

State ID	Transition Character	Destination State	Is Acceptance State
1	1	2	False
	2	3	
	3	4	

State ID	Transition Character	Destination State	Is Acceptance State
	4	4	
	5	4	
	6	4	
	7	4	
	8	4	
	9	4	
	0	11	
2	1	4	True
	2	4	
	3	4	
	4	4	
	5	4	
	6	4	
	7	4	
	8	4	
	9	4	
	0	4	
3	1	4	True
	2	4	
	3	4	
	4	4	
	5	25	
	6	11	
	7	11	
	8	11	
	9	11	

State ID	Transition Character	Destination State	Is Acceptance State
	0	4	
4	1	11	True
	2	11	
	3	11	
	4	11	
	5	11	
	6	11	
	7	11	
	8	11	
	9	11	
	0	11	
11			True
25	1	11	True
	2	11	
	3	11	
	4	11	
	5	11	
	0	11	

The proof of the DFA being an optimal DFA is given by the total number of unique outputs versus the number of total edge transitions required during a full graph generation. This final DFA does generate the 256 different strings required for the IP address byte value, and does so with the following calculation result:
The SUM of all the states incoming edges times the number of outgoing edges.
The following table outlines the states, their respective edges, and the totals.

State ID	# Incoming Edges	# Outgoing Edges	Total
1	1 (assumed start)	10	10
2	1	10	10
3	1	10	10
4	22	10	220
11	21	0	0
25	1	6	6
Grand Total:			256

Note that this calculation works only for regular expressions that do not contains the * or + operators. Those operators cause cycles in the graph. Therefore, if the resulting DFA is acyclic then the above statement holds true, otherwise, the number of matching strings is infinite and the resulting sum is meaningless.

Tests run on other variations of the IP address byte string regular expression also yielded the same equivalent DFA (despite some differences in state identifiers). Hence, the initial burden of writing an optimal acyclic regular expression for a given set of strings is not necessary, as the Korat regular expression engine optimizes the DFA to a reduced state of both memory and transition efficiency. For example, the following regular expression also yields the same post reduction DFA as the original and consequently outputs the same string results despite the syntactic difference from the earlier example.

```
((1|2|3|4|5|6|7|8|9)?(0|1|2|3|4|5|6|7|8|9))|((1(0|1|2|3|4|5
|6|7|8|9)(0|1|2|3|4|5|6|7|8|9))|2(((0|1|2|3|4)(0|1|2|3|4|5|
6|7|8|9)|5(0|1|2|3|4|5)))
```

6.0 USING THE REGULAR EXPRESSION LIBRARY

The regular expression generator operates both within Korat# and as a separate set of methods. In both cases, the initial instrumentation of the source code is the same. In both generation cases, the properties within the target classes need to contain the proper attributes that contain the desired regular expression strings that will be used to generate the data for testing or simple output. The regular expression library can operate outside of Korat# for purposes outside of test case generation. In many cases, sample data for a database, or even complex XML files may be required outside of automated testing. The regular expression generator as a standalone library can easily be integrated and used to create XML files or CSV files for import into a database. Moreover, custom data writers are easily plugged into the library in order to generate data in a specific format or sent to a specific location.

As was stated earlier, the regular expressions are currently limited to a subset of the regular expressions allowed in the `System.Text.RegularExpressions.Regex` class. Only the concatenation, union, star, plus and question mark operators are legal. Also, parenthesis can be used to specify grouping. Given these slight limitations, the majority of character based regular expressions is still easily possible. Future versions of the library will include enhancements to this set to include a syntactically more complete implementation.

First, to instrument the C# code with attributes that will be recognized by the regular expression generator, the project needs to create the regular expression generator attribute class. This class is automatically recognized by the library and takes the string parameter as its only input.

```
[System.AttributeUsage(AttributeTargets.Property)]
public class RegExGeneratorAttribute : Attribute
{
    private string _regEx;
```

```
    public string RegEx
    {
        get { return _regEx; }
    }

    public RegExGeneratorAttribute(string regEx)
    {
        _regEx = regEx;
    }
}
```

Here the Address1 property is attributed with a regular expression that will generate some sample address lines. Output from this regular expression would be similar to this:

```
[RegExGenerator("(1|2|3|4|5|6|7|8|9)(1|2|3|4|5|6|7|8|9|0)?
(Main|Center) (Street|Ave)")]
public string Address1 { get; set; }
```

Since the regular expression library uses the Mono.Cecil library and not System.Reflection, each library will need to define its own custom Attribute so the regular expression generator can load the type without necessarily have to load any libraries outside of the one being read.

Once the classes and properties are completely instrumented, it is time to run the generator. This part is done automatically if using Korat#. However, uses of the library outside of Korat# will need to use to important pieces of the regular expression generator API. A free and simple command line tool comes with the regular expression generator that allows for the selection of a library or executable file, the classes within the file to generate for, and the output type through a .NET application configuration file. However, if additional functionality is required, a developer can utilize the simple API to generate output for an entire library, or by `Type`.

These two methods allow for the complete generation of string data for classes with a controlled output. The library contains two implementations of the `IDataWriter`, one that outputs XML and the other outputs CSV. The XML output is used to store data generated for future use of Korat#, reducing the processing time necessary during automated testing. There is more information about this functionality in a later section. For example, given the following figure of a C# implementation, the corresponding CSV data generation outlines the first few lines of output:

Each individual type is output to a separate output file in the default implementation. Thus, the instrumentation of an entire library will yield as many XML or CSV files as there are instrumented types. In this way, creating imports to databases or generating large disconnected test cases are easily done and archived. The default implementation also guarantees that the output of each row of data is unique as a whole even though individual components may be repetitively used to fill in columns that have not produced as much generated data.

7.0 BUILDING YOUR OWN REGULAR EXPRESSION DATA GENERATOR

This section outlines a non-language specific way to create code that will take a regular expression as an input and generate the necessary data structures to build a set of strings that satisfy the expression. In order to successfully build the generator the supporting language must support some degree of primitives like integers and strings (character arrays) so that the necessary data structures are possible.

7.1 The Structures

The most basic structures of the Regular Expression Data Generator are the State Node and the Graph Walker. The State Node has the following basic structure:

- Integer: unique identifier
- Boolean: is final state indicator
- Hash Table: map of character transitions to other State Nodes

In C#, this definition looks like the following:

```
public class State
{
    int StateID;
    bool AcceptingState;
    Dictionary<char, List<State>> Transition;
}
```

The next data structure used is for walking the final deterministic finite automata, and is the structure that both keeps track of the set of match strings and keeps a history of visited State Nodes. The Graph Walker has the following basic structure:

- Character String: potential match string to this point in the DFA
- Set: current set of matched character strings
- Integer Array: list of states visited to this point of the path
- Integer: maximum times allowed to visit a given state
- Integer: maximum match string length

- Integer: maximum graph edges to traverse per State Node visit

In C#, this definition looks like the following:

```
public class GraphWalker
{
    string CurrentInput;
    Set<string> GeneratedData;
    int[] Ids;
    int MaximumLength;
    int MaximumVistsPerState;
    int MaximumForksPerState;
}
```

Where the generic Set structure is defined with this class:

```
public class Set<T>
{
    private Dictionary<T, int> _internalList =
                                        new Dictionary<T, int>();
    private object _syncLock = new object();
    public void Add(T item)
    {
        lock (_syncLock)
        {
            _internalList[item] = 0;
        }
    }

    public T[] Values
    {
        get { return new List<T>(_internalList.Keys).ToArray(); }
    }
}
```

7.2 The Algorithms

Next, a few well documented algorithms[9] are used for the generation of the NFA[8] and finally the DFA[7]. The basic operations of a regular expression are the concatenation, union, many-or-none, many-or-one, or one-or-zero operators. In ASCII these operators are commonly represented by: the lack of any other operator, |, *, +, and ? respectively. For the initial generation of the NFA, the algorithm first builds a start node and an end node. An important feature of the graph edges is that an edge that is marked by what is commonly referred to as the “Epsilon” character is an edge that is freely

traversed at any time. However, the algorithm only inserts these transitions where necessary for later collapse into the DFA.

Intermediate states and transitions are added to yield a graph that when traversed would necessitate a character appendage to an input string, or not, as indicated by epsilon transitions. Each character of the input string is determined to be a traversal character or an operator. The class `KoratRegEx` in the library fully outlines all the necessary algorithms for generating the NFA and the algorithm necessary for collapsing the NFA to a DFA. This paper does not attempt to summarize or describe these algorithms in detail since their implementation and description are well documented by the references.

7.3 Walking the DFA for String Generation

Once the DFA is finalized, the string generation is possible from a terminating standpoint. Throughout the DFA, one or more states may be marked as final states, meaning that the string until this point is valid. The basic algorithm is to start at the first node of the DFA with an empty input string and concatenating a character for each edge transition while adding the string to that point each time a final state is reached. However, in order to avoid infinite loops and allow for an incomplete set of matching strings for minimizing execution time, certain points of the algorithm check for reasons to terminate outside of not having an edge to traverse. In a full traversal of a DFA, the algorithm creates the empty string and enters the first state. For each state transition from that state, the algorithm creates a unique version of the empty string plus the character that maps the transition. Every subsequent state does the same iteration over each of the edges and performs likewise string concatenation operations. Again, each time a final state is reached, the input string to that point is added to the final set.

To save on execution time, the graph walker takes an integer configuration that specifies a maximum number of edges it can traverse. In some generation cases, only a subset of sample match strings is required. Thus, by limiting the number of traversal edges, the algorithm effectually prunes parts of the graph that need not be traversed. In order to do this randomly, the algorithm checks the number of edges leaving the current state. If the number of edges leaving is greater that the number allowed, then it randomly selects an edge the maximum number of times. While this does leave the chance for duplicates, it is faster than checking if one random edge has already been traversed.

The integer value in the graph walker that indicates the maximum number of times to visit a given state is to take care of circular graphs. Any regular expression that contains either the zero-or-more or one-or-more operators will be cyclic. To avoid any sort of infinite loop, a maximum number of times to visit any single state is used to terminate a given trail once the count of visits exceeds the set maximum. At that point, the method simply returns before starting any additional edge traversals.

8.0 UNITY OF METHODS

The union of the Korat data structure generation and the regular expression data generator is a powerful tool for test case generation. The generated structures are not only non-isomorphic, but the generated data that populates the structures is also valid. An example from the Korat# code base is found in the `KoratExamples` library and the `SortedList` class. This class is both structural and data driven in nature. The structure is in its sorted linked-list type structure. The data complexity comes in at the content of the elements within the list. In a sorted list, each element must be comparable by some criteria such that one item can be uniquely determined to be greater than, less than, or equal to any other distinct element in the list. Most high-level programming languages allow for complex definitions of the comparison operators and C# is no exception. Hence, we can expand on our address example from this paper to make a comparable address class that can be sorted in a list.

From the original Address class implementation noted earlier, we add the implementation of the `IComparable` interface class and provide the corresponding implementation.

```
        #region IComparable Members

        public int CompareTo(object obj)
        {
            int retval = -1;
            Address a = obj as Address;
            if (a != null)
            {
                retval = string.Compare(PostalCode, a.PostalCode);
            }
            return retval;
        }

        #endregion
```

Now, when Korat# initially starts up, it first checks the target library for classes that have the regular expression attributes above their properties. The sample objects are generated

and stored in a set of dictionaries. For the finitization class to access these generated samples, it only need to ask the `DataGenerator` class for a sample object from the generated data. The `DataGenerator` picks some random pieces of data from the generated strings, converts them to the properties target type, and populate a data structure for use in the Korat candidate vectors.

```
            public static IFinitization finSortedList(int minSize,
                        int maxSize, int numEntries, int numElems)
            {
                  IFinitization f =
                     FinitizationFactory.create(typeof(SortedList));

                  IObjSet entries = f.createObjSet(
                                                typeof(Entry), true);
                  entries.addClassDomain(f.createClassDomain(
                                          typeof(Entry), numEntries));

                  IIntSet sizes = f.createIntSet(minSize, maxSize);

                  IObjSet elems = f.createObjSet(typeof(Address));
                  IClassDomain elemsClassDomain =
                              f.createClassDomain(typeof(Address));
                  elemsClassDomain.includeInIsomorphismCheck(false);
                  RegExFinal.Output.DataGenerator dataGenerator =
Korat.Base.KoratBaseManager.GetInstance<RegExFinal.Output.DataGenerator
>();
                  for (int i = 1; i <= numElems; i++)
                      elemsClassDomain.addObject(
                              dataGenerator.GetSampleObject<Address>());
                  elems.addClassDomain(elemsClassDomain);
                  elems.setNullAllowed(true);

                  f.SetItem("Header", entries);
                  f.SetItem("Size", sizes);
                  f.SetItem(typeof(Entry), "Element", elems);
                  f.SetItem(typeof(Entry), "Next", entries);
                  f.SetItem(typeof(Entry), "Previous", entries);
                  return f;
            }
```

Now that some generated data exists in the Korat# engine. The test candidate generation will use the generated sample objects as the values for the sorted list. This results in a set of results that is both structurally accurate from the sorted list standpoint and has quality

sample data from the `Address` class standpoint. The resulting set of non-isomorphic structures is consequently satisfying of both syntactic and semantic qualifications.

In order to generate a new set of testable classes in a new library, the developer needs to perform the following steps in C#:

1. Create a library that has references to the `Korat.Base` and `RegExFinal` projects or libraries.
2. Create the `RegExGeneratorAttribute` class in this new library.
3. Create a class with attributed properties for data generation according to the rules of the regular expression grammar.
4. Create a class with structurally unique combinations for test case generation and implement all the testable properties with public `get` and `set` methods.
5. Implement the `repOK` method in this new class.
6. Implement the `finitization` method in this new class.

Note that these implementations can be optionally compiled out when not in the Release mode, and subsequently, the dependencies on the Korat libraries will be optimized out by the compiler. Hence, the developer need not deploy the application with the Korat libraries.

9.0 CONCLUSION

While the conversion of a piece of software from one language to another may have many obstacles, the result can often be a piece of software that can be utilized for years to come. Moreover, as libraries of code become publically available to the community with useful feature sets, additional uses are often invented for an even broader reach of applications. The creation of Korat in C# with the regular-expression data-generator consisted of two main portions of code translation. Each code base presented its own unique set of obstacles. Since the two source languages and the destination language were heavily based on C-style syntax and keywords, the translation could also be somewhat automated by translators and macros. Some of the highlights of translation are outlined in this section.

The main difficulty of the code translation was in the instrumentation of the Korat code. The libraries to insert byte code into Java are in no way similar to the libraries that insert IL into C#. This challenge also proved to be the most rewarding piece of the translation as the knowledge of how to manipulate already compiled code to perform additional executions is a powerful tool in any developer's skill set. During the translation, some monotonous parts were the simple search and replace pieces. Since Java and C# are similar enough to run some macros on, this methodology was employed to translate parts of the code. However, the effort was sometimes both beneficial and troublesome as some replacements led to bugs in the code. These bugs were most often in how the two languages handle nulls. Often, the C# code required additional null checks in order for the dynamic nature of the code to behave as expected. This process of finding these code statements took a long time to find and correct, because finding one would often uncover the next missing location and so on.

Another set of challenges in language translation projects arises in the differences of developer coding habits due to language variation, community mentality, or recommended practices. C# uses the notion of properties which are language structures that wrap two methods commonly used for the getting and setting of a class' private variables. So while the Java implementation had to consider the direct access of member variables, the C# implementation needed to consider this common practice of properties as the way to detect when an item had been accessed during the Korat candidate generation. This C# coding standard ended up being a significant advantage in the C# implementation of Korat as the properties are easily identifiable in compiled code and the get and set methods are easily added or manipulated due to the common structure that most implementations share. Some of the instrumentation methods of the Java Korat implementation were not necessary by using this common C# practice and while this may not be the most powerful implementation, it is definitely within the realm of common practices that does not hinder the use of the C# Korat in the .NET community.

Similarly, the translation from C++ to C# for the NFA and DFA generation had more than one challenge. Most of the code translated well due to the similar code structure and keywords shared between the two languages. However, since C++ is a pointer based language, all pointer based arithmetic in the original source code had to be translated to a non-pointer based algorithm. The similar data structures available to each language made finding equivalents easy despite some noticeable implementation nuances. The main work item in this code translation was the migrating of pointer-based structures and algorithms to the managed environment.

The translation of Korat to C# has the possibilities of being a forerunner of open source test generation tools for consumption by the .NET community. Also, the individual sections of Korat including the code injection, code generation, and regular expression

string generation can also be utilized in part or in combination to create even further software testing implementations.

REFERENCES

[1] C. Boyapati, S. Khurshid and D. Marinov Korat: Automated Testing Based on Java Predicates. (ISSTA '02) http://www.ece.utexas.edu/~khurshid/testera/korat.pdf.

[2] Microsoft Corporation – JLCA: Java Language Conversion Assistant. http://www.microsoft.com/downloads/details.aspx?familyid=664B33F5-B8F9-4642-990C-2C45BAB94A0F&displaylang=en

[3] Marques P. RAIL – Runtime Assembly Instrumentation Library. http://cisuc.dei.uc.pt/view_project.php?id_p=28

[4] Novell Mono Project – http://www.mono-project.com/Main_Page

[5] Leavens G. Java Modeling Project. http://www.cs.ucf.edu/~leavens/JML/

[6] Software Abstractions: Logic, Language, and Analysis. Daniel Jackson, pp. 366. The MIT Press, 2006. ISBN 978-0262101141.
Alloy Software Modeling Program – http://alloy.mit.edu/

[7] DFA Definition and Algorithms: http://en.wikipedia.org/wiki/Deterministic_finite_automaton

[8] NFA Definition and Algorithms: http://en.wikipedia.org/wiki/Nondeterministic_finite_state_machine

[9] Amer Gerzic Regular Expression DFA Creator: http://www.codeguru.com/cpp/cpp/cpp_mfc/parsing/article.php/c4093

[10] Javassit - Java Programming Assistant: http://www.csg.is.titech.ac.jp/~chiba/javassist/

VITA

Karlo Zatylny was born in Calgary, Alberta, Canada on June 13, 1975, the son of Clarence Zatylny and Anja Zatylny. He received his Bachelor of Science in Computer Science from Arizona State University in 2001. He was employed as a software developer at many different companies from 2000 in Arizona and Texas. In January 2007, he entered the Graduate School at the University of Texas at Austin.

This report was typed by Karlo Zatylny.