An Experiment in Automatic Conversion of Legacy Java Programs to C#

Mohammad El-Ramly
Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK
mer1@le.ac.uk

Rihab Eltayeb
Department of Computer Science, Sudan University of Science and Technology, Sudan
rahbon@hotmail.com

Hisham A. Alla
hmanssor@hotmail.com

Abstract. *Source-to-source transformation is an important tool for migrating key legacy programs to modern languages and platforms and giving them new life. Many organizations cannot do without their legacy systems on the one hand, but are stuck in an old technology on the other hand. Converting to a newer programming language can ease integration with modern technologies, give access to a wider developers population and/or lower maintenance costs. Serious language conversion efforts use automated tools, since manual conversion is out of question for non-trivial programs. We present our experiment in building a Java to C# transformer, Java2C#, that partially converts legacy Java code (version 1.1 or earlier) to C#. Java2C# is written in TXL, a language specially designed for program transformation, using tree re-writing. We explore and discuss the challenges and issues to consider when automatically transforming Java to C# and when building automated language transformers in general.*

1. Introduction

Source-to-source transformation (S2ST) is an instance of the wider problem of program transformation. S2ST has many variants, e.g., compilers perform S2ST to pre-process source code, expand macros, optimize code, etc. Obfuscators change source code or bytecode to make it harder to reverse engineer and understand. [3] Another application of S2ST is language conversion, e.g.:

- Converting a program to a newer version of the same language (Cobol 68 to Cobol 85) [1]
- Converting a program to a version of the same language under a different programming paradigm (Cobol 85 to Object Cobol) [6]
- Converting a program to a modern procedural or object-oriented language (Cobol to C or Java) [2]
- Migrating an application to a different system that supports a different dialect of the same language (Cobol on IBM Mainframe to AS/400 Cobol) [1]
- Structuring unstructured programs (Removing goto statements from Cobol programs) [9]
- Bug fixing and preventive maintenance, e.g., solving Year 2000 problem [10,11]

In this paper we present our experimental work on an instance of the S2ST problem. Specifically, we built an experimental Java to C# transformer, called Java2C#, to study the issues and challenges in converting legacy Java systems to C# under .Net framework. Java2C# can be downloaded from www.txl.ca. While Java is a young language, the rapid evolution of Java specifications quickly created legacy Java applications. As happened with many other technologies, the advent of C# and .Net created a demand for tools and techniques for converting legacy applications to the new language and platform. We built a partially automatic experimental transformer to transform a subset of Java 1.1 to C#. Partial here has two meanings. The first is that it coverts only a subset of the Java language, since it was designed as an experiment not for full commercial deployment. The second is that for certain language constructs, automatic conversion is almost impossible and some manual transformation is required. For these later cases, the transformer documents the issue and leaves comments in the transformed source code for the developer to complete the transformation manually. We studied the similarities and differences between the two languages and identified the main challenges in this conversion process.

Because of the time limit we had (8 months part-time during the M.Sc. project of the 2nd author) and the huge API of both languages, we focused on a subset of the core Java 1.1 and did not work on API translation.

To build the transformer, we chose an eloquent functional rule-based language, TXL (Turing Extender Language) [4,5], which is designed as a generic S2ST language.

In the following, Section 2 introduces TXL basics. Section 3 discusses the Java 1.1 subset currently supported in Java2C# and its similarities/differences with C#. Section 4 discusses Java2C# implementation with examples of the transformations supported. Section 5 presents related work. Section 6 presents the lessons learnt and conclusions.

2. TXL

TXL is a programming language for S2ST. TXL is specifically designed for manipulating and experimenting with programming language notations and features using S2ST. [4]. TXL is a functional rule-based language. It takes as input an arbitrary context-free grammar in EBNF (Extended BNF) notation [7], and a set of show-by-example transformation rules to be applied to the input programs.

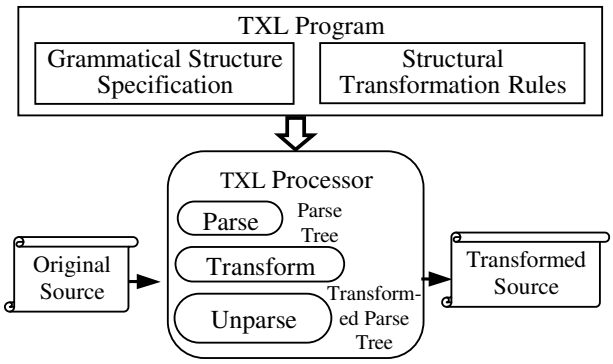

Figure 1. TXL Processing Engine [8]

TXL automatically parses input programs in the language described by the grammar, even if ambiguous or recursive, and then successively applies the transformation rules to the parsed input until they fail. It outputs the transformed source. Figure 1 shows TXL processing engine.

A TXL program has two components: (1) a description of the structure to be transformed in EBNF grammar in context-free ambiguous form and (2) a set of transformation rules and functions specified by example, using pattern/replacement pairs. A rule has the form:

`Left-HS → Right-HS IF Condition`

where `Left-HS` and `Right-HS` are term patterns. `Condition` is optional. The application of a rule to a term succeeds if the term matches the `Left-HS` pattern and the condition is true. The result is the instantiation of the `Right-HS` pattern. Rules are applied recursively until they fail. Functions are similar to rules but are applied once on the entire function input. A sample rule is shown in Figure 2. This rule applies to a sequence of one or more Java modifiers recursively. If a modifier is `final`, `transient` or `volatile`, it is removed since there is no equivalent to these in C#. This rule breaks a sequence of modifiers, represented by **`repeat`** `modifier`, into a `CurrentModifier` and the remaining modifiers in `RemainingModifiers`. Then if `CurrentModifier` is one of those unavailable in C#, it is replaced by the `RemainingModifiers` in **`by`** clause. The **`where`** clause contains the condition for applying the rule, where `isFinal`, `isTransient` and `isVolatile` are functions that check if a modifier is as the function name suggests. TXL uses % for comments and ' for strings.

TXL has several applications in software engineering and other areas including VLSI layout, database migration, and others. Example TXL uses in software engineering are [4]:

- Transforming between C, Pascal and Turing.
- Transforming between ISL, C++, Modula II and Ada.
- Y2K Bug Fixing (4.5 Billion LOC).

3. Java vs. C#

We studied the similarities and differences between Java and C# and classified the necessary transformations for converting Java programs to C# to four categories. The first is "same" (or no) transformation where the syntax of both languages is identical and the code is reproduced in the converted program. The second is "direct" transformations where one-to-one mapping between Java and C# exists and some rules for minor adaptation are needed. The third is "indirect" transformations where some tricks are needed to map a Java construct to a C# one. The fourth is "challenging" transformations where C# has no equivalent for a Java construct and clever tricks, intelligent techniques and manual intervention are needed to do a transformation. Appendix 1 lists the main transformations we found in each category. We added a fifth category of transformations, which are the ones we did not study because of the focused scope of this study. This category includes Java extensions and Java API transformations. Some Java APIs are directly translatable to .Net APIs and some are quite challenging.

```
% Rule [1-1-6]-----------------------------
% Remove Java final, transient and volatile
% modifiers. C# does not have them
rule removeNonCS
   replace [repeat modifier]
      CurrentModifier[modifier]
      RemainingModifiers [repeat modifier]
   Where CurrentModifier [isFinal]
       [isTransient] [isVolatile]
   by RemainingModifiers
end rule
% Function [1-1-19]------------------------
function isVolatile
   match [modifier]
      'volatile
end function
```

Figure 2. A TXL Rule That Deletes `final`, `transient` and `volatile` Modifiers.

C# was intended to exceed Java while still looking familiar. So, most Java concepts were preserved in C#. The number of new concepts introduced is more than those that are not supported in C#. This makes forward transformation from Java to C# easier than backward transformation from C# to Java. However, C# does not have its own API. So, calls to Java API need to be replaced by calls to .Net API, which may require thousands of TXL rules to transform.

4. Java2C# Implementation

This section describes the Java2C# transformer. First, it describes the primary requirements set for Java2C#, then the design process and design decisions made. Next, it describes the implantation and its components. Lastly, it provides examples of transformation rules and their applications.

4.1 Java2C# Requirements

Since this transformer is experimental and is meant to study the challenges in converting Java to C#, some requirements were set to ease extending and experimenting with it and to ensure the quality of the produced code:

1- The transformer should be easily extendable to include more transformation rules when needed.
2- The transformer should be easily updated to cover more recent versions of Java and C#.
3- Identifiers must keep their names after transformation if possible.
4- Original programmers' comments must be preserved and reproduced in the same locations in code, if possible.
5- Messages should be issued to the programmer as comments in the converted code when some manual transformation is needed.

4.2 Java2C# Design Process

In the beginning, a research on the similarities and differences between Java and C# was carried out to know the areas that need transformation and classify the required transformations according to the level of difficulty as explained in Section 3. The first step in writing a transformer, that uses tree rewriting via a parse-transform-unparse process, is writing working grammars for both the target and the source language and then writing a union grammar that accepts constructs from both languages. A grammar for Java 1.1 is available from TXL Web site [5]. Writing a C# grammar would be quite time consuming especially considering the limited time of this project. So, we resorted to a quicker but not ideal solution. Instead of writing a C# grammar and then writing a union grammar, we directly extended Java grammar to support C# syntax by using TXL redefine statement as in the following example:

```
%New C# set of modifiers for constants
define constant_modifier
    'const      % additional modifiers
    |'readOnly % used in C#
end define
redefine modifier
    ... % Now includes both Java and
    |[constant_modifier] % C# modifiers
end redefine
```

Next, we built the transformation engine. It consists of TXL rules and functions grouped in sets or modules. Each set of rules and functions transforms one language construct of Java (class headers, declarations, statements, etc.) to the equivalent in C#. These rules are logically grouped in a number of transformation sub-engines, each in one file.

4.3 Java2C# Implementation

The current implementation of Java2C# transformer is organized in modules. Each module is stored in a separate file. The overall structure of the transformer is shown in Figure 3. It contains the main module that starts the program and invokes the rules of other modules, which are 4 transformation modules and 2 utility modules. The role of different modules is briefly explained below:

- **JavaToC#.Txl** is the main module from which the transformation begins. It is used by TXL to match an input Java program against Java grammar and call the transformation rules to apply on the input program.
- **TranslateMembers.Rul** contains rules and functions to transform constructors, methods and nested declarations.
- **TranslateInitializers.Rul** contains rules and functions to transform instance initializers and static initializers.
- **TranslateFieldDeclaration.Rul** transforms field declarations (variable or constant declarations) when they are declared as class members.
- **TranslateBlockStatements.Rul** transforms the blocks that form the bodies of methods or constructors and transforms the statements, variable or constant declarations and control structures within the blocks.
- **DataStructures.Grm** contains the Java grammar extension to accept C# constructs beside the definition of the different mappers or tables used to directly map Java's constructs to C# equivalents.
- **HelperRules.Rul** contains common rules that are used in more than one place during the whole transformation.

The original Java source is left untouched after applying the transformation. Java2C# outputs a transformed source file. This file contains the result of the transformation where direct automatic conversion from Java to C# was possible. It also contains guiding comments where no transformation was applied.

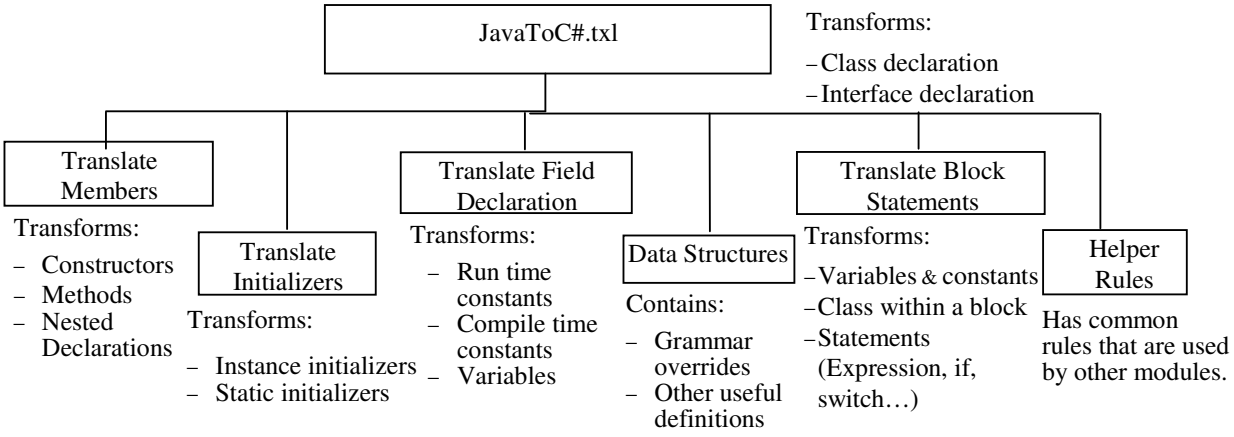

Figure 3. The Modules of Java2C#

Java

ClassDeclaration:
ClassModifiers$_{opt}$ class *Identifier*
Super$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

Super:
extends *ClassType*

Example

```
class Point {
  int x, y;
  Point (int x, int y)
  { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
  static final int WHITE=0,BLACK =1;
  int color;
  ColoredPoint(int x,int y,int color)
  {super(x,y); this.color = color;}
}
```

C#

ClassDeclaration:
Attributes$_{opt}$ *ClassModifiers*$_{opt}$
class *Identifier* *ClassBase*$_{opt}$ *ClassBody*

ClassBase:
: *ClassType*
: *InterfaceTypeList*
: *ClassType, InterfaceTypeList*

Example

```
class Point {
  int x, y;
  internal Point (int x, int y)
  { this.x = x;  this.y = y; }
}
class ColoredPoint : Point {
  const int WHITE =0, BLACK = 1;
  int color;
internal ColoredPoint(int x,int,y,int color):base (x,y)
  {this.color = color;}
}
```

Figure 4. Java and C# Syntax for Expressing Inheritance

4.4 Transformation Examples

In this section we provide some transformation examples. We show Java and C# syntax of the subject construct in EBNF and the TXL rules and functions that apply the transformation and explain the idea behind the transformation. These examples are selected to represent the different categories of transformations mentioned earlier.

Example 1: Direct Transformation - Inheritance Syntax

Both languages support the concept of single inheritance, which is having only one parent (base class) for the child (subclass). The difference is only in the syntax. C# uses C++ syntax (a colon followed by a name) and Java uses `extends` keyword instead. Figure 4 shows the EBNF grammar of class definition in both languages and an example class definition.

Transformation: The extends_clause in Java grammar is redefined to include C# inheritance syntax. changeExtend function does the transformation as below:

```
redefine extends_clause
    ...%Java
    |': [list qualified_name+]   % C#
end redefine
function changeExtend
    replace [opt extends_clause]
        'extends Enames[list type_name+]
    construct AllNames
        [repeat qualified_name]
        _[^ Enames]
    construct NewListEnames
        [list qualified_name]
        _[toQualifiedName each AllNames]
    by ': NewListEnames
end function
```

Example 2: Indirect Transformation–Instance Initializer

An instance initializer in Java is simply a block of code in a class that is not in any method. It is executed when an instance of a class is created. A Java block is enclosed between two curly brackets. C# doesn't permit a block to be present by itself as a class member declaration so the block in the instance initializer must be transformed to a method block to be a valid C# member declaration. Figure 5 shows the relevant grammar and examples.

Transformation: Because there is no direct class member declaration in C# that maps to Java's instance initializer, the instance initializer is transformed into a method by calling toMethods rule in Java2C# which builds a new method with the instance initializer as the method body and gives the method a unique name. The next step is to place a method call inside all constructors. If there is no constructor, a new constructor is created by the function setDefaultConstructor. If a constructor or more are provided there is a probability that a constructor may call its base class's constructor by placing this call as the first statement in its body. As a result a check must be done to see whether a base call is present and preserve the order of statements. In all cases the method call is added to the constructor body. This transformation is accomplished by the functions containSuper, containThis, addCallsToSuper and addCalls. The rule toMethods is presented below:

```
rule toMethods
    replace [class_body_declaration]
        Block[block]
    %name begin with initialMethod
    construct MethodID [id]
        initialMethod
```

Java	C#
InstanceInitializer: *Block*	*StaticConstructorDeclaration:* *Attributes$_{opt}$ StaticIdentifier ( ) Block*
Example	**Example**

`class InstanceInit2{`
`  {`
`  int tmp = 1;`
`  int x = 2;`
`  int y = 10;`
`  }`
`InstanceInit2(){`
`  super();`
`  byte b;`
`  }`
`}`
	`class InstanceInit2{`
	`private void initialMethod1 () {`
	`  int tmp = 1;`
	`  int x = 2;`
	`  int y = 10;`
	`  }`
	`internal InstanceInit2():base () {`
	`  initialMethod1 ();`
	`  sbyte b;`
	`  }`
	`}`

Figure 5. Java and C# Syntax for Expressing an Instance Initializer

Java	C#
FieldDeclaration: *FieldModifiersr$_{opt}$ Type* *VariableDeclarators* ; *FieldModifier:* public protected private final static transient volatile	*FieldDeclaration:* *Attributesr$_{opt}$ FieldModifiersr$_{opt}$ Type* *VariableDeclarators;* *FieldModifier:* new public protected internal private static readonly
Example	**Example**

`class FieldDeclaration {`
`  final int i1 = 10;`
`  static final int i2= 20;`
`  final boolean DONE = true;`
`  public static final long`
`  x1 = new Date().getTime();`
`  final Object v = new Object();`
`  final float f;`
`  FieldDeclaration() {`
`  f = 17.21f;`
`  }`
`}`
	`class FieldDeclaration {`
	`  const int i1 = 10;`
	`  const int i2 = 20;`
	`  const boolean DONE = true;`
	`  public static readonly long`
	`  x1 = new Date().getTime();`
	`  readonly Object v = new Object ();`
	`  readonly float f;`
	`  internal FieldDeclaration () {`
	`  f = 17.21f;`
	`  }`
	`}`

Figure 6. Java and C# Syntax for Constant Fields Declaration

```
%Add a number to the name to be unique
   construct MethodName[id]
       MethodID[!]
   construct MethodCall
      [declaration_or_statement]
      MethodName();
   import InitCalls
      [repeat declaration_or_statement]
   %Add new method call to previous calls
   export InitCalls
      InitCalls [. MethodCall]
   %Lastly the method itself
   construct initialMethod
      [member_declaration]
      'private 'void MethodName()
      Block
   by initialMethod
end rule
```

Example 3: Challenging Transformation – Constant Field Declaration

A variable that is declared as a member in a class is called a field variable. Java keyword `final` is used to express a named constant value that should not change during the execution, and the initial value is provided as part of the declaration. A final variable that is not initialized in its declaration is called a blank final. A non-static blank final variable can be left uninitialized when declared but must be assigned a value exactly once in an instance initializer or exactly once in each constructor. The static blank final variable can be left uninitialized when declared but it must be assigned a value exactly once in a static initializer. Although `const` is a Java reserved word it is not a keyword. C# provides the keyword `const` for compile time constants and `readonly` for the runtime constants. Figure 6 shows the relevant grammar and examples.

Java

```
package Txl;
class SwitchEx {
   public static void main (
                String args[]) {
      int i = 2;
      switch(i) {
         case 1:
            System.out.println("one");
         case 2:
            System.out.println("Two");
            break;
         case 3:
            System.out.println("Three");
             break;
         default:
           System.out.println("Default");
      }
   }
}
```

C#

```
using System;
namespace Txl
{
  class SwitchEx {
    public static void Main
                  (String [] args) {
       int i = 2;
       switch (i) {
          case 1 :
             Console.WriteLine ("one");
             goto case 2;
          case 2 :
             Console.WriteLine("Two");
             break;
          case 3 :
             Console.WriteLine("three");
             break;
          default :
             Console.WriteLine("Default");
             break;
        }
    }
  }
}
```

Figure 7. Full Program Transformation

Transformation: Java compile time constants (with primitive values) are transformed to C# constants and `static` keyword is removed from Java's declaration, if it is there, because C# `const` is implicitly static. The runtime constants are transformed to readonly constants. This is done by 7 functions translateFieldDeclaration, changeField, checkPrimitiveConstatnts, isCompileTime, checkRunTimeConstatnts, finalToConst, and finalToReadOnly. Two of them follow:

```
%for constants that has no immediate
%value in their declaration
function checkRunTimeConstants
   replace[field_declaration]
      Modifiers[repeat modifier]
      TypeSpecifier[type_specifier]
      VarDecl[variable_declarator];
   where Modifiers[containFinal]
   where not VarDecl[isCompileTime]
   by Modifiers[finalToReadOnly]
      [removeNonCSModifiers]
      [changeProtected]
      TypeSpecifier [changeDataTypes]
      [changeArrayTypes]
      VarDecl ;
  end function
%change the keyword final to readonly
function finalToReadOnly
   replace *[repeat modifier]
      CurrentModifier[modifier]
      RemainingModifiers
      [repeat modifier]
   where CurrentModifier[isFinal]
   by 'readonly
      RemainingModifiers
  end function
```

Example 4: Transformation of a Small Program

Java2C# was tried on several small size programs. Figure 7 is an example of full program transformation using some of the rules discussed above and others. It is a very small example due to space limitation. We provide the necessary explanation and details of how the transformation happens using Java2C# rules and functions at the end of the example.

The main transformation rule matches against the whole Java program, which consists of a package header, import declaration and type declaration. The package header is redefined to include the C# notation and changePackageToNamespace function transforms it to a C# namespace declaration. The import declaration is optional and the program does not contain an import clause. The default System namespace in C# is added automatically to the program because it contains most API and utility classes needed for simple programs.

The program consists of one type declaration, which is the class itself. changeClassHeader is matched by the class declaration and used to treat the whole class. It divides the class to its header and body. A new class header is created after calling changeModifiers, changeImplement, and changeExtend to do more refined work. A class body is transformed by translateEmptyBody for an empty body and changeClassBody for body with fields, instance initializers, static initializers, constructors and method declarations.

The class body contains a method declaration matched by translateMethods. The function pattern match is the special main method. As a method it consists of modifiers, type specifier, an optional throws clause and method body. The special thing about Java 'main' method is that its name should be transformed to 'Main' with capital M in C#.

changeMethodDeclarator function is used to capitalize the first letter from the name and also change the formal parameters declaration from String args[] to String [] args which is the valid C# array declaration. changeFormalParamsDataTypes is used to change data types between the two languages. The method body is sent to translateBlock to transform it.

The method block consists of one or more declaration or statement. The declaration of the variable `i` is matched by translateVarDeclaration which further calls checkLocalVars,checkLocalConstants, changeArrayDimensions, checkLocalRunTimeConstants and checkLocalBlankConstants functions to cooperate in transforming the local variable declaration.

The body also contains a switch statement, which is matched by translateStatementInBlock function and it selects the function changeSwitch to transform the switch statement. changeSwitch function calls changeExpression to transform the expression, addBreak to add a break statement to the case alternative statements if it is not provided and fallThrough to prevent falling through the next case alternative. It also calls changeSwitchStmts function to look after transforming the statements inside the case. The result of the effort of these functions together is the addition of a break statement to the default case in the program, the goto statement is constructed and added to case 2 to prevent falling through case 3. The System.out.println API call is transformed to its functional equivalence Console.writeLine call.

Every rule/function matches its pattern, does a successful transformation job and replaces the input Java code with the C# code. The result of transformation the C# program in Figure 7. Below are three of the functions used during the transformation process

```
%Change the method name and parameters
%If it is main method, rename it as Main
function changeMethodDeclarator
   replace [method_declarator]
      Name [method_name]
      '( FormalParms [list
      formal_parameter] ')
      Dim [repeat dimension]
      %change every parameter
   construct NewFormalParms
      [list formal_parameter]
      _[changeFormalParamsDataTypes   each
      FormalParms ]
   by Name[changeMain] '(
      NewFormalParms ')Dim
end function
function changeMain
   replace[method_name]
      'main
   by 'Main
end function
function translateVarDeclaration
   replace*[repeat
      declaration_or_statement]
      Var[local_variable_declaration]
      Remaining[repeat
         declaration_or_statement]
   by Var[checkLocalVars][
      checkLocalConstants]
      [checkLocalRunTimeConstants]
      [checkLocalBlankConstants]
      [changeArrayDimensions]
      Remaining[translateVarDeclaration]
end function
%In C# last alternative statements
%have to have a break
function addBreak
   replace[repeat switch_alternative]
      SwitchAlters [repeat
      switch_alternative]
   construct Length [number]
      %how many statements?
      _[length SwitchAlters]
   construct Index[number]Length[- 1]
      %get the last option
   construct LastAlter[repeat
      switch_alternative]
      SwitchAlters [tail Length]
      %start from tail
   deconstruct LastAlter
      %divide it into its contents
      Label[switch_label]
      Stmts
       [repeatdeclaration_or_statement]
   %no break ?
   where not Stmts[ContainBreak]
      % all alternatives before
      % the last one
   construct BeforeLastAlter
      [repeat switch_alternative]
      SwitchAlters [head Index]
      %a new break to be added
   construct Break [repeat
      declaration_or_statement]
      break ;
   construct NewStmts [repeat
      declaration_or_statement]
      Stmts[. Break]
      %add new break statement to others
   %new statements with break
   construct NewLastAlter[repeat
      switch_alternative]
      Label NewStmts
   construct NewSwitchAlters[repeat
      switch_alternative]
      BeforeLastAlter[. NewLastAlter]
   by NewSwitchAlters
end function
```

5. Related Work

Many specific and some generic research and commercial language conversion tools can tackle some S2ST problems with different levels of success. The larger the semantic and

syntactic gap between the source and target languages, the harder the conversion is [13]. The closest work to Java2C# is Microsoft Java Language Conversion Assistant (JLCA), currently available in version 2.0 and Beta version 3.0 [12]. It can convert Java applications to J++ and C# and Java API calls to native .Net Framework calls, 80% automatically, as the owners claim. See [14] for a discussion of JLCA, an example of code conversion, some of the issues that rise during API conversion and the manual coding required to finish up the conversion task. It is unclear what conversion technology is used in JLCA, as it is proprietary. It is also unclear how flexible and extendable JLCA is. Java2C# is unique in using parse tree rewriting via by-example style of rule specification, as in TXL. Extensions to cover other Java versions, addition of new transformation rules or change of existing ones are well supported by the flexibility of the transformer. Java2C# is unique in that it pays attention to the detailed and subtle differences between both languages. For example, Java2C# is watchful of the default access (when no access modifiers specified in the code) and is able to transform it. Java default access is friendly access while C# default access is private. Ignoring such defaults will be problematic in the resulting code.

6. Discussions, Lessons Learnt and Future Work

In this paper we presented our experience in building an experimental language transformer, Java2C# using tree rewriting via functional rule-based programming with TXL. We have learnt some lessons from this experience:

1. The primary reason for language conversion is migrating an application to a modern language or platform. The migration decision is affected by many factors including language and platform support, developers' availability, cost, performance and speed, market expansion and third party product availability. However, a crucial factor for making the decision is the availability of good tool support for the conversion; otherwise, only trivial programs can be manually converted cost effectively. Good tool support requires significant investment. Java2C# took 8 months part-time to build and it covers only a sub-set of Java 1.1 to C# language conversion, not including Java API. If we add the work needed to convert every single Java API to .Net API, excluding Java third party APIs, the effort needed will be enormous. If we consider that several versions of Java exist and that Java continues to evolve, we can easily imagine how things will scale up.
2. Adopting TXL to build the transformer allows incremental updates of the language transformer. Whenever a newer version of Java is released, the older TXL Java grammar can be updated and incremented with `REDFINE` statements and other languages constructs. Then the transformation rules can be updated as well.
3. Fully automated conversion is far from real in the current technology. This is because human intelligence and understanding is needed to deal with the cases when a language construct or feature in the source language is lacking in the target language, and there is no straightforward replacement. Thus, there will be always need for human effort to complete the missing pieces. However, using some intelligent algorithms, custom built solutions for specific programming patterns can help convert some of these cases.
4. Transforming the core Java language to core C# is a non-trivial task. But transforming the Java API, which has thousands of classes and methods, is a huge task. Considering how such APIs evolve and change, it becomes very important to develop methods for automatic API transformation. Graph transformation technology might be a possible solution for this problem, where one can define a graphical model of both the source and target APIs and then define graph transformation rules from the source to the target. However, this idea needs further investigation.

For future work, Java2C# can be enhanced to be more interactive and user friendly by generating additional comments and reports about the transformation process, transforming packages instead of one file at a time and adding a wizard-oriented graphical user interface.

This work can be extended to apply to new Java specifications after 1.1 in order to transform more recent Java programs. This involves upgrading the Java grammar in TXL by providing the grammar rules for newer features. Including other Java technologies such as JSP, Servlets, and Swing will be easier because Java2C# transforms the core java code. As an example JSP (Java Server Pages) code consists of special tags and objects, Java code and HTML tags. To transform JSP to ASP.Net we only need rules to map the JSP tags and objects to ASP.

Acknowledgements: The authors like to thank Jim Cordy and Thomas Dean for their valuable advice and help during the implementation of Java2C#.

References

[1] H. Sneed, *Risks Involved in Reengineering Projects*, Proc. of 6th Working Conf. on Reverse Eng. (WCRE), pp. 204-, 1999.

[2] M. Mossienko, *Automated Cobol to Java Recycling*, Proc. of 7th European Conf. on Software Maintenance and Reengineering (CSMR), pp. 40-50, 2003.

[3] C. Collberg, C. Thomborson, and D. Low. *A Taxonomy of Obfuscating Transformations*. Tech. Rep. 161. Dept. of Comp. Sc., The Univ. of Auckland, New Zealand, July 1997.

[4] J. Cordy, *TXL - A Language for Programming Language Tools and Applications*, Proc. ACM 4th Int. Workshop on Language Descriptions, Tools and Applications (LDTA 2004), pp. 1-27, 2004. Electronic Notes in Theoretical Computer Science 110, pp. 3-31 (Keynote paper).

[5] J. Cordy, *, The TXL Programming Language, Version 10.4*, 2005. Available at www.txl.ca.

[6] K. Cremer, A. Marburger and B. Westfechtel, *Graph-Based Tools for Re-engineering*, J. of Software Maintenance and Evolution: Research and Practice, 14(4), pp. 257-292, 2002.

[7] ISO/IEC, *Information technology - Syntactic Metalanguage -Extended BNF, ISO/IEC 14977:1996(E)*, 1996

[8] T. Dean, J. Cordy, A. Malton and K. Schneider, *Grammar Programming in TXL*, Proc. IEEE 2nd Int. Workshop on Source Code Analysis and Manipulation (SCAM'02), pp. 93-102, 2002.

[9] N. Veerman, *Restructuring Cobol Systems Using Automatic Transformations*, M.Sc. Thesis, Vrije Universiteit Amsterdam, 2001.

[10] J. Cordy, T. Dean, A. Malton and K. Schneider, *Software Engineering by Source Transformation - Experience with TXL*, Proc. IEEE 1st Int. Workshop on Source Code Analysis and Manipulation (SCAM'01), pp. 168-178, 2001.

[11] T. Dean, J. Cordy, K. Schneider and A. Malton, *Experience Using Design Recovery Techniques to Transform Legacy Systems*, Proceedings IEEE Int. Conf. on Software Maintenance (ICSM), pp. 622-631, 2001.

[12] Microsoft, *Java Language Conversion Assistant 3.0* (Beta), 2004.

[13] A. Terekov and C. Verhoef, *The realities of language conversion*, IEEE Software 2000, 17(6): 111 124, 2000.

[14] N. Nanda and S. Kumar, *Migrating Java applications to .Net*, JavaWorld Jan. 2003. Available at www.javaworld.com/javaworld/jw-01-2003/jw-0103-migration_p.html

Appendix 1

The following table shows the main transformations identified and classified in different categories of transformations that are needed to transform Java legacy applications to C#. Identical constructs are not shown here.

Direct Transformations

Java Concept	C# Concept	Supported
Package	Namespace	Fully
`import` declaration	`using` clause	Fully
`main` method	`Main` method	Fully
`extends` clause	Colon followed by a name	Fully
`finalize` method	~ class name	No
`synchronized`	`lock`	Fully
`final`	`const/readonly`	Fully
`byte`	`sbyte`	Fully
`boolean`	`bool`	Fully
`final` class	`sealed` class	Fully
`native`	`extern`	Fully
No modifier	`virtual`	Fully
Final method	No modifier (default is final)	Fully
`protected`	`protected internal`	Fully
Shift operator >>>	Shift operator >>	Fully
`instanceof`	`is`	No
Multidimensional arrays	Jagged arrays (of same length)	Partially
Default access is `friendly`	Default access is `private`	Fully
`static` initializer	`static` constructor	Fully
Nested top level class or interface (static)	Static nested classes	Fully
Block with semi colon	Block with nothing inside it	Fully

Indirect Transformations

Java Concept	C# Concept	Supported
`super ()`	`Base ()`	Fully
`transient` modifier	Non-existent	Fully
`volatile` modifier	Non-existent	Fully
`Implements` clause	":" followed by a name	Fully
Instance initializer	Non-existent	Fully
Data type variable name [ ]	Data type [ ] variable name	Fully
`System.out.print` `System.out.println`	`Console.write` `Console.writeline`	Fully

Challenging Transformations

Java Concept	C# Concept	Description	Supported
Run time and blank local constants	C# doesn't allow local constants without initial constant value	Local constants with no value in their declaration.	Fully
`throws` clause	Non-existent	All C# exceptions are unchecked	Fully
`java.lang`	Non-existent (System class contains most functionality)	Default package (API calls)	Partially
Interface constant fields	Non-existent	Shared constants declared in an interface	A comment is added for manual inter-vention.
Inner classes (non static)	Non-existent	A class defined as a member (non static) of another.	
Local class	Non-existent	A class defined in a block of code.	
Anonymous class	Non-existent	Unnamed class defined within an expression.	
Anonymous arrays	Non-existent	Unnamed arrays	No
`switch` Statement	`switch` Statement	Multiple-selection structure	Fully
`break` to label	`goto` statement with the same label name after it	Transfers control to a label	Fully
`continue` to a label	`goto` statement with the same label name after it	Transfers control to a label	Fully

Not Studied

Java Concept	C# Concept	Description
Java Swing	Windows Forms Applications	Provide GUI support
Java Collections Framework	. NET Collection Classes	APIs and main language packages
Applets	Windows user controls	Web pages client side technology
JSP and Servlets	ASP.Net	Dynamic web pages
JDBC Java Database Connectivity	ADO.Net	Database access
EJB Enterprise Java Beans	. Net managed Components	Application server classes