Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 3–18
www.elsevier.com/locate/entcs

Translate One, Analyze Many: Leveraging the Microsoft Intermediate Language and Source Code Transformation for Model Checking

Jesse McGeachie[1] Juergen Dingel[2]

School of Computing
Queen's University
Kingston, Canada

Abstract

In this paper we present a source transformation-based framework to support model checking of source code written with languages belonging to Microsoft's .NET platform. The framework includes a set of source transformation rules to guide the transformation, tools to support assertion checking, as well as a tool for the automation of deadlock detection. The framework results in both executable and formally verifiable artifacts. We provide details of the tools in the framework, and evaluate the framework on a few small case studies.

Keywords: Source transformation, model checking, TXL, Microsoft Intermediate Language, Java bytecode.

1 Introduction

Software development has seen a variety of emerging technologies and programming languages in recent years. Microsoft's .NET platform [13] and its associated languages (e.g., C#, VB .NET, J#) is experiencing increasing popularity. The Microsoft Intermediate Language (MSIL) [9] forms the core of .NET to which all languages that belong to .NET are compiled to. With the help of MSIL, .NET thus generalizes Java's "write once, run everywhere" concept into "write many, run everywhere".

Recently, software model checking has been a very active research topic. A number of techniques have been proposed that intend to achieve two goals. First, the

[1] Email: jesse.mcgeachie@gmail.com
[2] Email: dingel@cs.queensu.ca

doi:10.1016/j.entcs.2007.02.057

benefits of a fully automatic, highly optimized, yet exhaustive state space exploration should be brought to bear on programs written in standard, high-level languages such as Java, C/C++, or C#. Second, the pitfalls of an error prone manual translation of the programs into the input language of established model checkers such as Spin [6] and SMV [2] should be avoided. With the growing trend towards more concurrency in everyday software (e.g., to reap performance benefits [17] or to implement embedded systems) this research seems particularly timely. Existing software model checkers either translate the program into the input language of an existing model checker (e.g., Bandera [3], or Feaver [7]), or augment the execution environment of the language to carry out the model checking (e.g., Java Pathfinder [18], Verisoft [5], or Zing [1]).

In this paper, we investigate the development of a software model checker for .NET. More precisely, we show how a common representation like MSIL can be leveraged for analysis purposes in general, and model checking in particular. In principle, MSIL allows the analysis of multiple languages with only a single transformation: the transformation of MSIL to the input language of the analysis tool. The contribution of this work is a framework (MSILCAD) for the automatic transformation of a subset of MSIL to Bandera Intermediate Representation (BIR), the input language of the Bogor model checker [15]. At the core of our framework is the Microsoft Intermediate Language-to-Java Bytecode transformation (MSIL2JBC). The MSIL2JBC transformation has been designed to support a subset of MSIL that is the result of a reasonably sized subset of both C# and J# programming languages. The framework also contains tools that support assertion violation checking as well as deadlock detection. To the best of our knowledge, the only other model checker for MSIL is Zing [1]. In contrast to Zing, our framework is translation-based, and leverages the capabilities of the Bandera toolset.

We have chosen to implement the MSIL2JBC translation with TXL [4], a programming language specifically designed to support structural source transformation. The TXL program responsible for the MSIL2JBC translation applies a number of transformation rules to the MSIL input, which are separated into rule-sets, each with its own specific transformational purpose. For instance, one rule-set handles the creation of threads, while another rule-set provides the ability to support method invocation, and so on.

In the remainder of this paper, we provide background on MSIL, the Java assembler format JASMIN [11], TXL and Bandera in Section 2, and a complete overview of MSILCAD in Section 3. Section 4 explains the evaluation of MSILCAD and example experiments, and Section 5 discusses related work. Finally, in Section 6 conclusions are given followed by future work.

2 Background

2.1 Microsoft Intermediate Language

At the heart of Microsoft's .NET platform is an intermediate language to which multiple source languages can be compiled. It is similar to Java bytecode but it is

more complex due to the fact that it is designed specifically to be the target of many languages. Source code is compiled to this intermediate language before it is just-in-time compiled to native code for some target platform. The work presented in this paper is concerned with a restricted subset of this language and the remainder of this section will focus only on that subset.

MSIL is a stack-based language. The subset of MSIL that we have focused on has instructions to load literal values onto the stack (`ldc`), to create arrays (`newarr`), to load and store values between fields (`ldfld`, `ldsfld`, `stfld`, `stsfld`), to load method arguments (`ldarg`), as well as some others. Instructions handling standard arithmetic and boolean comparisons are supported. Moreover, instructions to do nothing (`nop`), to branch conditionally (`br`), or unconditionally (`brfalse`, `brtrue`), and to return from a method (`ret`) are handled. Finally, limited support has been included for concurrency and object synchronization by allowing thread and monitor objects.

The translation is restricted to a subset of MSIL code that is the result of a subset of C# and J# source code, which does not include any unsafe code (unmanaged pointers) possible within C#.

2.2 JASMIN

Java bytecode for a Java program resides in the program's classfiles. It is in binary format so it cannot be easily manipulated. By using the *javap* disassembler provided by Sun Microsystems, ASCII bytecode instructions can be produced. For the reverse direction (ASCII to JBC), Sun has not defined a standard Java assembler format, so there does not exist a standard tool for assembling Java programs from bytecode in ASCII format.

JASMIN [11] is a Java Assembler Interface that takes ASCII descriptions for Java classes, written in a simple assembler-like syntax using the Java Virtual Machine [10] instruction set, and converts them into binary Java classfiles. Since its creation, JASMIN has become the de-facto standard assembly format for Java. Our work uses the JASMIN syntax as the target language of our transformation so that we can leverage the JASMIN Assembler to produce Java binaries.

2.3 Source Code Transformation with TXL

TXL [4] is a programming language that is specifically designed to support source transformations. TXL supports unification, deep pattern match and implied iteration, and combines features from both functional and rule-based programming.

A TXL program consists of two parts: a context-free, possibly ambiguous grammar which describes the overall syntactic structure of the artifacts to be transformed, and a set of structural transformation functions and rules that use pattern-replacement pairs to describe the transformations that are desired. The implementation and formal semantics of TXL are based on tree rewriting where matching transformation rules are applied to the input until a fixed point is reached. For instance, when invoked on a file containing a sequence of numbers, the TXL program

```
define input
    [repeat number]
end define

rule main
    replace [repeat number]
        N1 [number] N2 [number] Rest [repeat number]
    where
        N1 [> N2]
    by
        N2 N1 Rest
end rule
```

Fig. 1. TXL program to sort the numbers in a file

in Figure 1 sorts the contents of the file.

The initial **define** statement expresses that the input to the program is a possibly empty sequence of numbers where "number" is a built-in non-terminal that matches any unsigned integer or real beginning with a digit. The rule matches every pair of consecutive numbers n_1 and n_2 in the input that are not ordered and replaces that pair by n_2 and n_1. The rule will be applied as long as it can be matched. A TXL program terminates if no match for any of the rules can be found.

2.4 Bandera

Bandera is a component-based model extractor and model checker for Java programs. Its current implementation handles all of Java, including object synchronization, multi-threading, and assertions. Its component-based architecture for model extraction is designed for scalability, flexibility and extensibility. Our framework utilizes Bandera for extracting optimized models from Java code, and model checking these models for assertion violations and deadlock.

3 Overview of MSILCAD

The overall architecture of MSILCAD (MSIL Check Assertions & Deadlock) is illustrated in Figure 2. The user invokes MSILCAD with one explicit input: the C# or J# source code to be translated and analyzed. The source code is first processed by the Pre-Transformation Processor, which ensures that assertions in the source code are handled correctly by the MSIL2JBC translator. The modified source code is

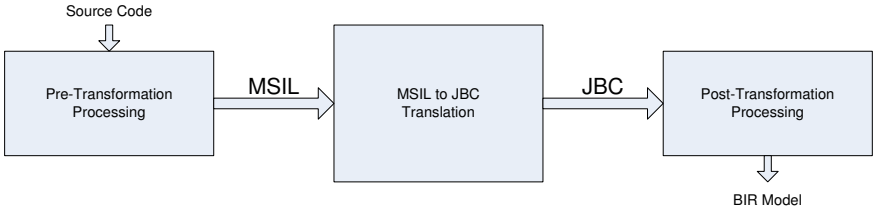

Fig. 2. MSILCAD Architecture Overview

then disassembled to MSIL code and is passed directly to the MSIL2JBC translator. The MSIL2JBC translator produces Java bytecode and Java source code, both corresponding to the MSIL code the translator was passed. Finally, the Java bytecode and source code are passed to the Post-Transformation Processor where Bandera is prepared for analysis, assertion code is finalized, a BIR model is extracted from the Java source, and Bandera is used to check for assertion violations and deadlock. Details for all steps of MSILCAD are given in the remainder of this section.

3.1 Pre-Transformation Processor

An overview of the Pre-Transformation Processor architecture is illustrated in Figure 3.

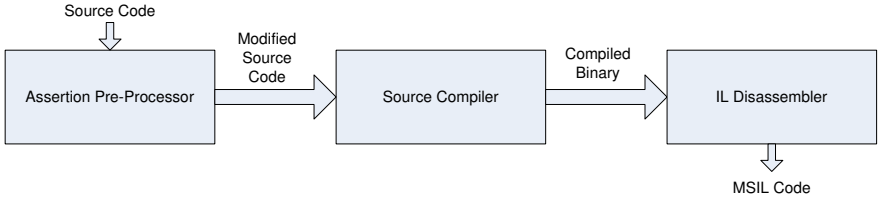

Fig. 3. Pre-Transformation Processor Architecture Overview

3.1.1 Assertion Pre-Processor.

The MSIL2JBC translator does not handle the use of assertions at the bytecode level due to the complicated nature of producing assertions in the Java source during the assembly from bytecode. The Assertion Pre-Processor replaces assertions in the source code with something that the MSIL2JBC is able to translate such that it is flagged for later discovery. Afterwards, during the Post-Transformation Processing stage, these replacements are located and assertions are realized. This step is achieved with a simple Visual Basic Script.

3.1.2 Source Compiler.

The modified C# or J# source code is then compiled with its respective compiler, resulting in an executable binary. This step is necessary because disassembling the

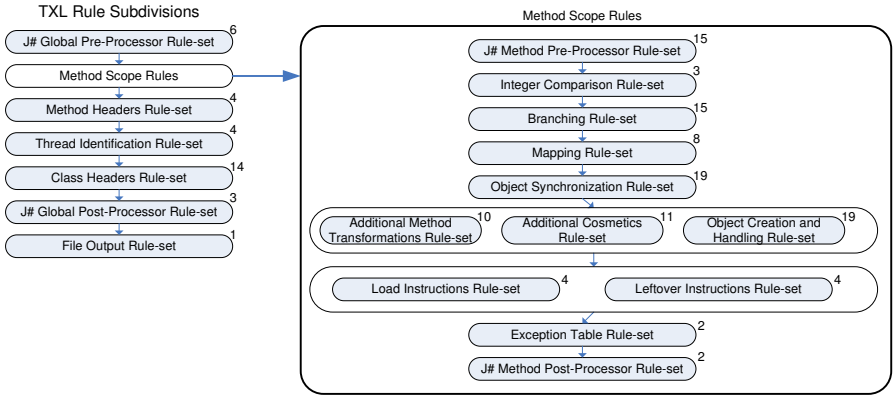

Fig. 4. TXL Rule-set Hierarchy

MSIL code requires binary as input.

3.1.3 Intermediate Language Disassembler.

The IL Disassembler takes as input the binary program produced by the compiler, disassembles it using the *ildasm* utility provided with Visual Studio .NET, resulting in an ASCII format of the MSIL code corresponding to the original C# or J# source code.

3.2 MSIL to JBC Transformation

This section describes the TXL program that accomplishes the MSIL to JASMIN source code translation. The translation has been broken up into 18 separate rule-sets that serve individual transformational purposes, with a total of 144 rules. Some rule-sets are completely independent of all other rule-sets, while others are very dependent on the results of rule-sets before them for accomplishing their purposes. Figure 4 illustrates the hierarchy of the set of rule-sets, as well as the number of rules in each set. The rule-sets must be applied in the illustrated hierarchy in order to ensure the correctness of the translation. Only the most interesting rule-sets will be discussed below due to space restrictions.

3.2.1 TXL Transformation.

J# Global Pre-Processor Rule-set: Rather than creating an entire new set of translation rules designed specifically for J#, we identified the differences between J# and C# MSIL code for our original supported subset of C# and we created a number of rules for translating J# MSIL to equivalent C# MSIL. More specifically, the rules in this set are used to remove unnecessary J# MSIL and to make minor adjustments to method and class headers to satisfy the syntax of C# MSIL. By leveraging our existing MSIL2JBC translation rules we reduced the overhead of

adding support for J# significantly.

Method Header Rule-set: The Method Header Rule-set has three important tasks:

(i) Translating MSIL method headers to equivalent JASMIN headers.

(ii) Translating method arguments in method headers as well as in any other context.

(iii) Translating type declarations in method headers as well as in any other context.

For translating type declarations, a TXL rule with a static mapping table is used. Figure 5 illustrates the rule `c_convertTypes`.

The static mapping table is stored in the `typePairs` construct. The leftmost columns of the table represent the type declarations in MSIL, and the rightmost column represents the corresponding JASMIN type declarations. The rule matches all type declarations in MSIL, performs a lookup in this table, and replaces the MSIL type declaration with the correct JASMIN declaration. For example, if `class [mscorlib]System.Threading.Thread` is matched, it would be replaced by `Ljava/lang/Thread;`.

Thread Identification Rule-set: The purpose of this rule-set is to locate the presence of threads in the MSIL code, and to translate these threads to semantically equivalent JASMIN code. Threads are more flexibly implemented in C# than Java, resulting in a non-trivial translation. In order to alleviate some of the inherent difficulty of this translation, the way threads are allowed to be used is restricted: code intended to be run as a separate thread must be contained within a separate class from any other thread, and only a separate class containing the `main` method is permitted to instantiate any new Thread object. Note that this restriction is always possible with some manipulation of the source code.

Class Header Rule-set: The Class Header Rule-set translates all MSIL class headers and class field declarations to equivalent JASMIN. These translations are relatively straightforward as the differences between MSIL and JASMIN are minor: the information describing the class is in a different order, and MSIL contains class information that is meaningless in JASMIN, meaning that is it specific to the .NET architecture.

Integer Comparison Rule-set: The Integer Comparison Rule-set translates three MSIL instructions for comparing integer values (`ceq`, `clt`, `cgt`) to equivalent JASMIN instructions. Table 1 illustrates one of these translations.

The `ceq` instruction compares two integer values, n_1 and n_2, and if n_1 is equal to n_2, then 1 is pushed onto the stack, otherwise 0 is pushed. There is no single JASMIN instruction equivalent to this functionality so a series of JASMIN instructions are required to mimic this exact behavior.

```
rule c_convertTypes
    construct typePairs [repeat type_pair]
      'void 'V
      'int8 'I
      'int16 'I
      'int32 'I
      'int64 'I
      'int8 '['] '[I
      'int16 '['] '[I
      'int32 '['] '[I
      'int64 '['] '[I
      'bool 'Z
      'bool '['] '[Z
      'class '[ 'mscorlib '] 'System.Threading.Thread 'Ljava/lang/Thread;
      'class '[ 'mscorlib '] 'System.Threading.Thread '['] '[Ljava/lang/Thread;
    replace [type]
    deconstruct * [typeMatches] argMemory
      anytype [type]
    deconstruct * [type_pair] typePairs
      anytype corr [type]
    by
      corr
end rule
```

Fig. 5. Static Mapping Table for Type Declarations

Mapping Rule-set: Keeping track of a method's arguments and local variables is necessary due to the fact that MSIL and Java bytecode treat the storage and access of data in different ways. MSIL must access arguments and variables from two different storage locations, where Java bytecode need only access a single location. This results in MSIL requiring two different instructions for accessing these locations (`ldarg`, `ldloc`) and JASMIN only requiring one. Moreover, the position of any given argument or variable in the storage location will differ

Table 1
`ceq` Instruction Translation

MSIL	JBC
ceq	ifeq BRANCH1 iconst_0 goto BRANCH2 BRANCH1: iconst_1 BRANCH2:

Table 2
Dynamic Tables for Loading Instructions for Local Variables and Parameters

Local Variables Dynamic Table	Arguments Dynamic Table
000 –> `aload_0`	000 –> `aload_0`
0 –> `iload_1`	1 –> `aload_2`
999 –> `aload_4`	2 –> `iload_3`

between MSIL and JASMIN. To complicate things further, JASMIN instructions for storing and loading to and from the data storage location require a prefix representing the type of the data being stored or loaded. MSIL instructions do not require this type of information. The Mapping Rule-set creates a table dynamically that matches MSIL instructions for loading and storing to equivalent JASMIN instructions, complete with matching indices and type prefixes. The dynamic table is then referenced at any time within other rule-sets in order to produce the correct translation for loading and storing instructions. This dynamic table looks similar to the static table as previously mentioned in Figure 5, but must be created dynamically as the information stored in it is not known at compile time. Table 2 illustrates typical examples of local variables and arguments dynamic tables for loading instructions.

The first entry in each table simply represents a place holder to satisfy TXL syntax requirements. The second entry in the Loading Variables Dynamic Table states that position 0 in the local variable storage location should be mapped as type integer (`i` prefix), and to position 1 in JBC (`_1` suffix). The last entry from the same table is labeled as position 999, and is used to represent the location where exception objects are stored. Exception objects are reference types, so the load instruction is prefixed with an `a`, representing a reference type. The Arguments Dynamic Table (ADT) is to be interpreted similarly.

Each table is located in a global variable, and is imported and used as needed by a number of TXL rules. Figure 6 illustrates how the ADT is used by the `c_loadArguments` TXL rule, from the Load Instructions Rule-set.

```
rule c_loadArguments
    import argMemory [repeat typeMatches]
    replace [methodBodyItem]
      'ldarg. somenumber [integernumber]
    deconstruct * [typeMatches] argMemory
      somenumber '–>corrInstrLOAD [methodBodyItem]
    by
      corrInstrLOAD
end rule
```

Fig. 6. `c_loadArguments` TXL Rule

A `ldarg` instruction with some position value (`somenumber`) is matched by the rule, the position value is extracted and used during a lookup in the ADT (`argMemory`), and the corresponding MSIL instruction (`corrInstrLOAD`) is returned from the table and used in the replacement.

Monitor Rule-set: The Monitor Rule-set provides support for the use of object synchronization. More specifically, the following methods from the .NET Monitor class are supported: `Enter`, `Exit`, `Wait`, `Pulse`, and `PulseAll`.

Exception Table Rule-set: Exceptions are stored in a dynamic global variable as they are discovered during the translation. At the end of the translation the finalized exception table is added to the JASMIN code.

3.2.2 JASMIN Assembly.

Immediately following the completion of the MSIL to JASMIN source transformation, the JASMIN source files are assembled into actual Java bytecode, in binary format, otherwise known as Java classfiles.

3.2.3 JAD Decompiling.

Finally, the Java classfiles are decompiled with a command-line enabled Java decompiler called JAD [8]. The result of decompiling the classfiles is Java source code.

3.3 Post-Transformation Processor

Figure 7 illustrates the overall architecture of the Post-Transformation Processor.

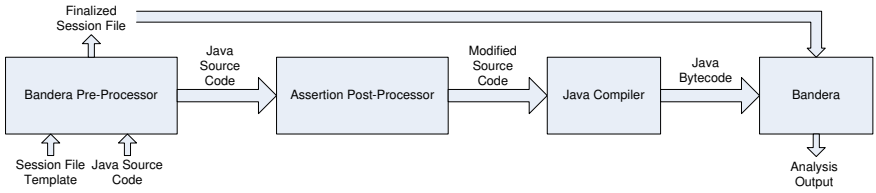

Fig. 7. Post-Transformation Processor Architecture Overview

3.3.1 Bandera Pre-Processor.

Bandera requires three inputs: Java source code, Java bytecode, and a Bandera session file.

Bandera requires a session file that informs itself of the location of the source code and bytecode on the host machine, as well as which of the classes involved contains the `main` method, and which tools included in the Bandera toolset should be applied during the analysis.

The Java source code and bytecode have already been produced at this point, but the session file needs to be created. We have implemented a script in Visual Basic Script that updates a template Bandera session file with the locations of the Java source code and bytecode, as well as which class contains the `main` method. The session file contains commands to invoke Bogor for deadlock detection and assertion violation checking, as well as a tool to output the BIR model of the program to a file. When the Bandera Pre-Processor completes, the result is a fully functional Bandera session file, and a verifiable BIR model of the program.

3.3.2 Assertion Post-Processor.

The Assertion Post-Processor scans the Java source files for any indication of user defined assertions created by the Assertion Pre-Processor (as discuessed in Section 3.1.1). If the use of assertions is detected, what has been located is replaced by valid Java assertion statements. This step is achieved by a simple Visual Basic Script.

3.3.3 Source Compiler.

Since there is the possibility that the Java source has changed due to the Assertion Post-Processor, the source code needs to be re-compiled in order to ensure that our Java classfiles (bytecode) correctly reflect the updated source code. The Java compiler provided with the Java SDK is used for this compilation.

3.4 Analysis with Bandera

The final step in MSILCAD is the extraction of a BIR model from the Java source code that has been generated. The generated session file is input to Bandera, the BIR model is extracted, and the *BogorTool* is invoked to analyze the BIR model for deadlock and assertion violations. The results of the analysis are displayed as

```
<toolResults id="bogorTool"
   class="edu.ksu.cis.projects.bogor.tool.BogorTool" time="26s 47ms">
   <toolResult id="Result">Errors</toolResult>
   <toolResult id="Print"><![CDATA[   [Time: 3516 ms, Depth:
      282] Error found: Invalid end state    Total memory before
      search: 93,079,928 bytes (88.77 Mb)  Total memory after
      search: 93,257,000 bytes (88.94 Mb)  Total search time: 3563
      ms (0:0:3)  States count: 281  Matched states count: 9  Max
      depth: 282  Done!  ]]></toolResult>
```

Fig. 8. Bandera Results: Deadlock

C#

```
public class BufferImpl
{
   private bool [] buf;
   private int inn;
   private int outt;
   private int count;
   private int size;
   private int i;

   public BufferImpl (int size)
   {
      this.size = size;
      buf = new bool [size];
      for (i=0; i<size; i++)
         buf[i]=true;
         count = 0;
   }

   public void put(bool b){
      lock(buf){
         while (count==size) Monitor.Wait(buf);
       buf[inn] = b;
       count++;
       inn = (inn + 1) % size;
       Monitor.Pulse(buf);
      }
   }

   public void get(){
      lock(buf){
         while (count==0) Monitor.Wait(buf);
       buf[outt] = false;
       count--;
       outt=(outt + 1) % size;
       Monitor.Pulse(buf);
      }
   }
}
```

Java

```
public class BufferImpl
{
   private boolean buf [];
   private int inn;
   private int outt;
   private int count;
   private int size;
   private int i;

public BufferImpl(int j)
   {
      size = j;
      buf = new boolean[j];
      for(i = 0; i < j; i++)
         buf[i] = true;
         count = 0;
   }

   public void put(boolean flag){
      synchronized(buf){
         while(count == size) try{buf.wait();}
            catch(InterruptedException
               interruptedexception) { }
       buf[inn] = flag;
       count++;
       inn = (inn + 1) % size;
       buf.notify();
      }
   }

   public void get(){
      synchronized(buf){
         while(count == 0) try{buf.wait();}
            catch(InterruptedException
               interruptedexception) { }
       buf[outt] = false;
       count--;
       outt = (outt + 1) % size;
       buf.notify();
      }
   }
}
```

Fig. 9. Bounded Buffer Transformation

plain text on the console.

4 Evaluation of Transformations and Experiments

To evaluate our framework we used three examples: the Dining Philosophers, Bounded Buffer, and Peterson's tie-breaker algorithm. For each example, our evalu-

ation involved programming the algorithms in C# and J# languages and verifying (by hand) that our transformation to Java was performed correctly. We demonstrated that semantics were well preserved across all of the transformations by checking that the execution and model checking behavior of the target matched the original semantics of the source. Finally, we applied model checking for deadlock detection and inserted assertion statements to test for assertion violations.

4.1 Dining Philosophers

We implemented two versions of the Dining Philosophers problem using five philosophers and monitors for synchronization. The first version was deadlock free, while the second was not. We analyzed both versions with Bandera to test for deadlock and both analyses produced the expected results. Figure 8 illustrates the *BogorTool* section of the analysis results. It indicates that deadlock has been discovered in the example allowing deadlock.

4.2 Bounded Buffer

The Bounded Buffer algorithm is an interesting example for transformation as it contains object locking, various monitor methods (`Wait`, `Pulse`) and exception handling. Figure 9 illustrates the complete source transformation from C# to Java for the `BufferImpl` class, taken from the Bounded Buffer example.

4.3 Peterson's Tie-Breaker Algorithm

Peterson's n-process tie-breaker algorithm solves the mutual exclusion problem for an arbitrary number of processes [12]. We used MSILCAD and Bandera's *BogorTool* to check that our J# implementation of the n-process tie-breaker algorithm guaranteed mutual exclusion for 2 and 3 processes. For larger numbers of processes our translation was successful, but Bogor ran out of memory.

4.4 Selected Transformation Metrics

Table 3 illustrates the number of lines of code (LOC) for each artifact produced, for both the Dining Philosophers and Bounded Buffer examples. As expected, the LOC for the source languages (C# and Java) are similar, for both examples. Java contains more LOC, in most part due to Java's need for explicit exception handling. The number of LOC for the bytecode artifacts (MSIL, JASMIN) are comparable as well. MSIL has a higher count as it contains a "manifest" containing assembler directives required by the runtime that are not required by the JVM.

All transformations were applied on an Intel Pentium 4, 1.6Ghz CPU with 512MB of RAM and the transformation times were negligible.

Table 3
Metrics for Translation Artifacts

	Dining Philosophers	Bounded Buffer
C#	67	88
MSIL	276	364
JBC	232	298
Java	81	122

5 Related Work

Current software model checkers roughly fall into two categories. The first is based on a translation framework which transforms a program in the source language into an equivalent finite state machine represented in the input language of an existing model checker. Bandera (for Java) [3] and Feaver (for C) [7] are based on this concept. In the second category of software model checkers the standard runtime environment is replaced by a customized one that implements the checking. Examples in this category include Java PathFinder (for Java) [18], VeriSoft (for C/C++) [5], and Zing (for MSIL) [1].

Just like Zing, our approach targets MSIL. However, it differs from Zing in that it is translation-based. Moreover, we leverage the existing optimization and analysis capabilites of the Bandera toolset.

At the time of writing a new approach to model checking is in development for Java bytecode. The authors of Bogor and Bandera have introduced an early prototype of BogorVM [14], a model checker aimed directly at model checking Java bytecode, and not source code. BogorVM is in the early stages of development and has limitations, thus we used Bandera for our purposes.

A comparison between Microsoft's Common Language Runtime (CLR) and the Java Virtual Machine (JVM) was performed in [16]. This work includes a translation from a subset of the CLR to the JVM in order to provide a better understanding of the differences between these two architectures. Our work is similar in this respect but reached beyond the virtual machine comparison to the formal verification of the software by means of model checking, with the transformation framework acting directly upon the Microsoft source code, and producing the corresponding Java source code. Also, our transformation also supports the use of assertions, and the automatic verification of these assertions after the transformation by Bandera. Finally, we were able to leverage source code transformation techniques to implement an easily extensible translation in a convenient and high-level fashion.

6 Conclusions and Future Work

We have presented an approach to model check the .NET family of languages by leveraging the "many-to-many" feature of the Microsoft Intermediate Language (MSIL). The approach uses source code transformation to translate MSIL into Java

bytecode. The Bandera toolset is then used to optimize and analyze the resulting bytecode. We have implemented the approach in a prototype and evaluated it with promising results on several small examples. TXL proved to be a convenient vehicle to realize the translation. While there are a few language features currently not handled by our prototype, we do not see a reason why it cannot be extended to not only support the entire safe subset of MSIL, but also other .NET languages such as Visual Basic.

Immediate future work for this project includes extending the scope of the translation. More precisely, the translation will be improved to handle bigger subsets of J# and C# as well as other .NET languages like Visual Basic and C++. Moreover, to get the full benefit of model checking, support for temporal properties should be added. To make the analysis more user-friendly, any counter examples produced by Bandera should be mapped back to the original source. In other words, standard techniques should be applied to make our TXL translation "fully reversible". Finally, the direct translation of MSIL to BIR would make another, more long-term research topic.

References

[1] Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J. and Xie, Y. Zing: A Model Checker for Concurrent Software, Technical Report MSR-TR-2004-10, MSR (2004).

[2] Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M. *NUSMV: a new symbolic model checker*, International Journal on Software Tools for Technology Transfer. **2(4)** (2000), 410–425.

[3] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. *Bandera : Extracting Finite-state Models from Java Source Code*, 22nd International Conference on Software Engineering (2000).

[4] Cordy, J.R., Dean, T.R., Malton, A.J., and Schneider, K.A. *Source Transformation in Software Engineering using the TXL Transformation System*, Journal of Information and Software Technology. **44(13)** (2002), 827–837.

[5] Godefroid, P. *Software Model Checking: The VeriSoft Approach*, Formal Methods in System Design. **26(2)** (2005), 77–101.

[6] Holzmann, G.J. "The SPIN Model Checker: Primer and Reference Manual", Addison Wesley, 2004.

[7] Holzmann, G., and Smith, M.H. *Software model checking — Extracting verification models from source code*, Formal Methods for Protocol Engineering and Distributed Systems (1999).

[8] Kouznetsov, P. Jad - the fast JAva Decompiler. URL: http://www.kpdus.com/jad.html.

[9] Lidin, S. "Inside Microsoft .NET IL Assembler", Microsoft Press, 2002.

[10] Lindholm, T. and Yellin, F. "The Java Virtual Machine Specification", Addison Wesley, 1999.

[11] Meyer, J., and Reynaud, D. Jasmin Home Page (2004). URL: http://jasmin.sourceforge.net/.

[12] Peterson, G.L. *Myths about the mutual exclusion problem*, Information Processing Letters. **12** (1981), 115–116.

[13] Platt, D. S. "Introducing Microsoft .Net", Microsoft Press, 2003.

[14] Robby. BogorVM: Customizing Bogor for model checking Java programs homepage (2006). URL: http://bogor.projects.cis.ksu.edu.

[15] Robby, Dwyer, M.B. and Hatcliff, J. *Bogor: An Extensible and Highly-Modular Model Checking Framework*, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (2003).

[16] Shiel, S. and Bayley, I. *A Translation-Facilitated Comparison between the Common Language Runtime and the Java Virtual Machine*, Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (2005).

[17] Sutter, H. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, Dr. Dobb's Journal. **30(3)** (2005).

[18] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. *Model Checking Programs*, Automated Software Engineering Journal. **10(2)** (2003).