INFERRING LIKELY DESIGN CONTRACTS IN EXISTING CODE

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Brian J. Bell

SPRING
2012
INFERRING LIKELY DESIGN CONTRACTS IN EXISTING CODE

A Project

by

Brian J. Bell

Approved by:

________________________, Committee Chair
Dr. Cui Zhang

________________________, Second Reader
Dr. Du Zhang

________________________
Date
Student: Brian J. Bell

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

Dr. Nik Faroughi
Graduate Coordinator

Date

Department of Computer Science
Abstract

of

INFERRING LIKELY DESIGN CONTRACTS IN EXISTING CODE

by

Brian J. Bell

There are no existing tools for automatic inference of likely design contracts in existing code for the .NET environment. Design contracts are useful for program verification, program testing, and program maintenance. Automatically inferring likely design contracts adds the benefit that possible contracts, which might otherwise be overlooked, are highlighted. Additionally, automatic inference of likely code contracts provides a base set of contracts, which eases the adoption of Design by Contract™.

The expense in time and effort of annotating existing programs to include design contracts, was the motivation for this project and for creating the tool named IcsRewrite, which was developed to reduce this cost. The inferring of contracts performed by IcsRewrite is based on a set of constraint relaxation algorithms specifically developed for this purpose. This project demonstrates that automatic inference of likeley code contracts via runtime analysis is viable and useful.

__________________________, Committee Chair
Dr. Cui Zhang

__________________________
Date
DEDICATION

This work is dedicated to
my parents, Ken & Regina Bell.
ACKNOWLEDGMENTS

I would like to thank, my advisor, Dr. Cui Zhang for her time and encouragement in finishing this Master’s project.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Dedication</th>
<th>Acknowledgements</th>
<th>List of Figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
<td>vii</td>
<td>viii</td>
</tr>
</tbody>
</table>

## Chapter

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 BACKGROUND AND RELATED WORK</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Design by Contract and Implementations</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Related Work for Creating Design Contracts of Existing Programs</td>
<td>4</td>
</tr>
<tr>
<td>3 ICSREWRITE DESIGN</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Instrumentation</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Contract Inference via Constraint Relaxation</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Precondition versus Postcondition Inference</td>
<td>14</td>
</tr>
<tr>
<td>3.4 Applications and Examples of IcsRewrite</td>
<td>14</td>
</tr>
<tr>
<td>4 ICSREWRITE RESULTS</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Numeric Pre/PostConditions</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Sequence Pre/PostConditions</td>
<td>20</td>
</tr>
<tr>
<td>4.3 Generic Pre/PostConditions</td>
<td>22</td>
</tr>
<tr>
<td>5 ICSREWRITE PERFORMANCE EVALUATION</td>
<td>26</td>
</tr>
<tr>
<td>5.1 Instrumentation Performance</td>
<td>26</td>
</tr>
<tr>
<td>5.2 Instrumented Program Performance</td>
<td>27</td>
</tr>
<tr>
<td>6 CONCLUSION AND FUTURE WORK</td>
<td>29</td>
</tr>
<tr>
<td>Appendix Source Code</td>
<td>30</td>
</tr>
<tr>
<td>References</td>
<td>49</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 IcsRewrite Workflow</td>
<td>1</td>
</tr>
<tr>
<td>2.1 Daikon Implementation</td>
<td>4</td>
</tr>
<tr>
<td>3.1 IcsRewrite Runtime Flow</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Program Instrumentation</td>
<td>9</td>
</tr>
<tr>
<td>3.3 IL Code</td>
<td>9</td>
</tr>
<tr>
<td>3.4 Instrumented Negate IL Code</td>
<td>10</td>
</tr>
<tr>
<td>3.5 Non-Numeric/Non-Sequence Variable State Machine</td>
<td>12</td>
</tr>
<tr>
<td>3.6 Numeric Variable State Machine</td>
<td>13</td>
</tr>
<tr>
<td>3.7 Sequence Variable State Machine</td>
<td>13</td>
</tr>
<tr>
<td>3.8 Example Program &amp; Implicit Contracts</td>
<td>15</td>
</tr>
<tr>
<td>3.9 MinFirstLast Test Suite</td>
<td>16</td>
</tr>
<tr>
<td>3.10 Results MinFirstLast</td>
<td>17</td>
</tr>
<tr>
<td>4.1 NumericTests.cs</td>
<td>19</td>
</tr>
<tr>
<td>4.2 NumericTestsInferredContracts.txt</td>
<td>20</td>
</tr>
<tr>
<td>4.3 SequenceTests.cs</td>
<td>21</td>
</tr>
<tr>
<td>4.4 SequenceTestsInferredContracts.txt</td>
<td>22</td>
</tr>
<tr>
<td>4.5 ParamTests.cs</td>
<td>24</td>
</tr>
<tr>
<td>4.6 ParamTestsInferredContracts.txt</td>
<td>25</td>
</tr>
<tr>
<td>5.1 Instrumentation Times</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Instrumented versus Non-Instrumented Time</td>
<td>28</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

With the release of .NET 4.0, Microsoft provided their Code Contracts library, which is an implementation of Design by Contract\textsuperscript{TM} [1]. The inclusion of the Code Contracts library in .NET 4.0 means that a large percentage of the software development community can easily use Design by Contract\textsuperscript{TM} in their programs. A limitation is the cost of annotating existing programs to include design contracts. This report presents a tool named IcsRewrite, which has been developed, aimed at reducing this cost. The name IcsRewrite is a shortening of “Inferred Contracts Rewrite”, the rewrite portion of the name is because it rewrites byte-code. IcsRewrite analyzes the runtime execution of .NET programs and automatically infers likely design contracts. Thus, IcsRewrite reduces the amount of manual coding needed to add design contracts to existing .NET programs.

IcsRewrite provides the benefit of automatically inferring likely design contracts in an existing .NET program. For IcsRewrite to work two things are needed: (1) a .NET program, and (2) a test suite for the .NET program. The completeness of the program test suite dictates the “goodness” of the contracts that IcsRewrite infers. The inferring of contracts performed by IcsRewrite is based on a set of constraint relaxation algorithms specifically developed for this purpose. Figure 1.1 shows the IcsRewrite workflow.

Figure 1.1: IcsRewrite Workflow
The Mono implementation of .NET is chosen due to its open source nature and ease of modification. Microsoft’s Code Contracts library is closed source, which makes it difficult to add functionality. The Mono implementation of .NET is reasonably complete and is widely used in commercial applications such as Unity.

The rest of this report is organized as follows: Chapter 2 discusses background information on Design by Contract and related work, with regards to inferring likely contracts; Chapter 3 discusses the design of IcsRewrite and shows several examples to demonstrate its usage; Chapter 4 gives the results of this project; Chapter 5 discusses several performance evaluations of IcsRewrite; and Chapter 6 gives the conclusion of this project and its future work.
2.1 Design by Contract and Implementations

Design by Contract™ is a way of ensuring that code is correct and defect free. It consists of three parts.

- Preconditions of methods
- Postconditions of methods
- Class invariants

Preconditions and postconditions are defined on methods, and class invariants are defined on data of objects. Preconditions are conditions on the parameters passed to a method and on the program state. They are conditions on the state of the object before the method is executed. Similarly, postconditions are conditions on method return values and the resulting program state. Object invariants check that an object state satisfies certain conditions.

The original idea and implementation of Design by Contract™ is due to Bertrand Meyer, in his Eiffel programming language for which he wrote the definitive book “Eiffel: The Language”[^3]. Microsoft .NET implements Design by Contract™ in two ways, first via runtime checking in which the design contracts are checked at runtime and also via static checking in which the contracts are checked at compile time. The name of their implementation is called “Code Contracts” and is available for use in C#, VB.NET, and other .NET programming languages.

Mono[^2] is an alternative open source implementation of .NET. Chris Bacon, as part of Google Summer of Code 2010[^4], created an open source implementation of .NET Code Contracts for Mono. It was decided to use this open source implementation of Code Contracts as the basis for this project. The reasons are that (1) it’s open source and hence easier to debug or modify, and (2) Linux was used as the main development environment, which precludes using Microsoft’s .NET implementation.
2.2 Related Work for Creating Design Contracts of Existing Programs

In Meyer’s paper introducing Design By Contract\textsuperscript{TM}[3], he does not explicitly mention how to deal with large legacy programs that are written without design contracts. His paper assumes that Design by Contract\textsuperscript{TM} will be used in writing new programs/components.

Meyer and Arnout’s paper [5] on finding implicit contracts in .NET libraries, similarly to this project, looks for implicit contracts in existing code bases. For their work, they did not use automated runtime analysis for extracting contracts, instead they used manual inspection to find implicit contracts. Their paper focused on determining if contracts are inherent in reusable libraries. They searched for implicit contracts in the .NET collection library, specifically in the “ArrayList” class. They concluded that contracts are implicit in the .NET base class libraries and that making them explicit would be beneficial to users of the libraries. In section 9 of their paper they discuss the possibilities of automatic contract inference, they say regularity of the contracts they inferred in the .NET base class libraries shows possibility for automatic contract inference.

Michael D. Ernst’s PhD thesis “Dynamically Discovering Likely Program Invariants” did groundbreaking work on using runtime analysis for finding program invariants [6]. As the name of his thesis states, his thesis was in the same area as this project. For his thesis he wrote a tool called Daikon that traced the execution of a program while it was running and from that deduced likely program invariants. His method of tracing program execution was to capture variable values at interesting points of program execution. He showed that (1) his method can rediscover formal specifications, and (2) that the invariants discovered are of use to programmers. Figure 2.1 shows the process of how Daikon detects program invariants. To instrument a program Daikon runs the

![Figure 2.1: Daikon Implementation](image-url)

Figure 2.1: Daikon Implementation
program through a parser that modifies the program source code to include logic for capturing variable values. There are instrumenter implementations for C and Java, which parse C and Java code to include the needed modifications for capturing variable traces. The output of the instrumentation is the modified C or Java code which is then compiled and run.

This project, unlike Diakon, does not parse the program source code, instead byte-code injection is used [7] to add the instrumentation logic.

Ernst’s article “The Daikon system for dynamic detection of likely invariants” [8] gives a practical guide for using Daikon for inferring program invariants. Ernst shows how the invariants detected can be used for generating test cases, correcting inconsistent data structures and several other tasks.

Another related work is the paper “Inference and enforcement of data structure consistency specifications [9].” This paper uses dynamic detection for inferring data structure invariants. The authors use the inferred data structure invariants to instrument the program to detect and automatically repair data structure corruption. One of their examples is using BIND (a widely used Internet name server), they show that their technique of automatic data structure repair ameliorated/eliminated two known security vulnerabilities in BIND.

Ernst’s paper “Verification for legacy programs” [10] gives an overview of why invariant detection is useful in legacy programs. He argues that automatic invariant detection via runtime analysis enables legacy programs to incorporate many of the benefits from formal specifications. Ernst’s paper also explores the use of operational semantics in predicting incompatible component upgrades. The two techniques they use are describing observed component behavior (via dynamic invariant detection) and comparing behavior differences in component versions. As one of their case studies, they applied their technique to the Comprehensive Perl Archive Network (CPAN) [11]; they verified the safety of some component upgrades and discovered incompatibilities in other upgrades (which could cause software failures).

Wei et al. in their paper “Inferring Better Contracts” [12], give a sophisticated approach that combines static and dynamic inference. They make use of the presence of simple programmer
written contracts to infer more complex and complete contracts. The process of inference uses a test suite to analyze the program’s classes and methods, along with the programmer written contracts. The test suite is automatically generated from the class and method API in addition to the programmer written contracts. They were able to infer 75% of the complete contracts.

A related area of work is automatically creating unit tests given an existing program. Typically unit test suites for programs are manually written and validated. For existing programs without test suites, creating a test suite is a large investment. Similarly, adding design contracts to legacy programs is a large investment. Microsoft Research produced a tool, Pex [13], that when given a program will automatically create interesting test cases, with high code coverage of the program. The use of Pex can reduce the amount of manual effort needed for adding test suites to existing programs.

Polikarpova et al. in the paper “A comparative study of programmer-written and automatically inferred contracts” [14], compared programmer written and automatically inferred contracts. The comparison used Eiffel’s standard library as the code base. They compared the already written design contracts for Eiffel’s standard library with Diakon’s automatically inferred contracts. In their conclusion they said that automatic contract inference infers \( \sim 60\% \) of the original programmer-written contracts, in addition to other contract elements that were not originally included in the programmer-written contracts.
ICSREWRITE DESIGN

IcsRewrite was designed to do the following: (1) take as input a .NET program, (2) instrument the program to include the necessary IcsRewrite instrumentation functions, and (3) write the modified program back to disk. When the instrumented program is run, the included instrumentation code logs all method invocations and infers possible contracts. Once the program exits the inferred contracts are written to a log file. Figure 3.1 shows the runtime flow of IcsRewrite for adding runtime contract inference.

The beauty of this design is when the modified assembly is executed, the contract analysis function is able to analyze the execution paths and infer possible contracts. When the assembly finishes executing, the inferred contracts are written to a log file.
The entire process of program instrumentation and inferring likely design contracts is automatic. The only manual steps are executing the program once it’s instrumented and inserting the inferred contracts back into the program source code.

3.1 Instrumentation

The first step in designing IcsRewrite was determining how to instrument a program. The Mono.Cecil \[15\] library was the best fit for the purpose. Mono.Cecil allows ease of loading a program, modifying the program to include instrumentation functions, and writing the modified program back to disk.

The instrumentation code was designed to use Mono.Cecil for loading a program from disk into memory via \textit{Mono.Cecil.AssemblyDefinition.ReadAssembly} and iterating through each class’s
methods to add instrumentation code. Instrumenting methods took the most thought and care, due to the complexities of handling overloading and properly handling inherited methods.

Instrumenting a method consists of two tasks: 1) adding intermediate language (IL) code for recording the method parameter values passed in and 2) adding IL for recording the method return value (not needed for methods with a void return type). For recording the parameter values passed in a MethodEnter function was written, which is called with the method name and the respective parameter values. Similarly, for recording the method return value the function MethodExit was written that records the method name and return value.

Figure 3.2 shows how MethodEnter and MethodExit are used for adding instrumentation.

A simple example illustrates the process of using MethodEnter and MethodExit. The example method is \( \text{int Negate(int } x) \{ \text{return } -1 * x; \} \), which compiles to the byte-code in Figure 3.3.

```
instance default int32 Negate (int32 x) cil managed
{
    // Method begins at RVA 0x2138
    // Code size 4 (0x4)
    .maxstack 8
    IL_0000: ldc.i4.m1
    IL_0001: ldarg.1
    IL_0002: mul
    IL_0003: ret
} // end of method Negate
```

Figure 3.3: IL Code

Instrumenting Negate consists of adding IL code to call MethodEnter before “IL_0000: ldc.i4.m1” and adding IL code to call MethodExit before “IL_0003: ret”. The instrumented version
of Negate looks like “int Negate(int x) MethodEnter(new object[] x); MethodExit(-1 * x); return -1 * x;”. For clarity, the IL code of the instrumented version of Negate is shown in Figure 3.4.

```
instance default int32 Negate (int32 x) cil managed
{
    // Method begins at RVA 0x20e8
    // Code size 43 (0x2b)
    .maxstack 8
    IL_0000: ldc.i4 1
    IL_0005: newarr [mscorlib]System.Object
    IL_000a: dup
    IL_000b: ldc.i4 0
    IL_0010: ldarg.1
    IL_0011: box [mscorlib]System.Int32
    IL_0016: stelem.ref
    IL_0017: call void class [ContractInference]ContractInference.
             Rewritter::MethodEnter(object[])
    IL_001c: ldarg.1
    IL_001d: ldarg.1
    IL_001e: mul
    IL_001f: dup
    IL_0020: box [mscorlib]System.Int32
    IL_0025: call void class [ContractInference]ContractInference.
             Rewritter::MethodExit(object)
    IL_002a: ret
} // end of method Negate
```

Figure 3.4: Instrumented Negate IL Code

With Mono.Cecil it is easy to insert the IL code for calling MethodEnter at the beginning of the IL code for Negate. It is also easy to iterate through the IL code of Negate and before every ret instruction insert IL code for calling MethodExit. Both MethodEnter and MethodExit are regular C# functions that reside in the IcsRewrite assembly.

### 3.2 Contract Inference via Constraint Relaxation

Contracts are only inferred for a constrained set of possible contracts. For preconditions, contracts are inferred only on the parameter values passed into the method, no analysis or inference is done for globals or other variables. Similarly, postconditions are inferred only for the method return value and are not inferred for anything else.
The design for contract inference is based upon constraint relaxation. The constraints start with the strongest possible pre/postconditions and as data is observed the pre/postconditions are relaxed to comply with the data. The particulars of the constraints and constraint relaxation are both dependent on the variable data type. Previous work on contract inference, such as Daikon, takes a slightly different method of inferring likely contracts. In particular, Daikon creates a list of potential pre/postconditions and as data is observed it rules out invalid pre/postconditions. Once the list of valid pre/postconditions is finished, they are filtered to exclude uninteresting contracts. Karine Arnout and Bertrand Meyer’s paper [5], on contract extraction, analyzed the .NET bytecode to extract exception cases as preconditions.

To limit the analysis to a feasible set of constraints, only the following constraints were used:

1. Over any variable:
   
   (a) constant value: \( x = a \) (indicates the variable is a constant)
   
   (b) non-null: \( x \neq \) null (indicates the variable is never null)
   
   (c) small value set: \( x \in \{a, b, c\} \) (indicates the variable takes on a small set of possible values)

2. Invariants over a single numeric variable:
   
   (a) range limits: \( x \geq a, x \leq b, \) and \( a \leq x \leq b \)
   
   (b) nonzero: \( x \neq 0 \) if the variable is never set to 0
   
   (c) modulus: \( x = a \mod b \) (indicates that \( x \mod b = a \) always)

3. Invariants over a single sequence variable (i.e. array)
   
   (a) range: minimum and maximum sequence values, ordered lexicographically, for instance this can indicate the range of a string or array values.
   
   (b) element ordering: whether the elements of each sequence are non-decreasing, non-increasing, or equal.
The above list of constraints is a subset of the invariants checked by Daikon, from Ernst’s PhD thesis [6].

The constraints and their method of relaxation are different for each of the above variable types (e.g. the constraints for a numeric variable are different than the constraints for a sequence variable). The following subsections give the details of the constraints for each variable type.

3.2.1 Generic Variables

For variables that are non-numeric and non-sequence the possibilities are: (1) no data, (2) no constraints, (3) the variable value is always non-null, or (4) the variable takes on a small set of possible values. The algorithm for constraint relaxation uses a state machine with states that correspond to (1), (2), (3), and (4). The state machine is initially set to the state “No Data” and when new variable values are observed the state machine determines the next state. Because one of the states is that the variable is constrained to a small set of possible values, the state machine records all observed values. Figure 3.5 shows the state machine, along with its state change decision process.

Figure 3.5: Non-Numeric/Non-Sequence Variable State Machine

3.2.2 Numeric Variables

Numeric variable constraints have four states: (1) no data, (2) constrained to a range, (3) constrained to non-zero, or (4) no constraints. Figure 3.6 shows the state machine for numeric variables.
3.2.3 Sequence Variables

For the purposes of this project, sequence variables consist strictly of variables with type $List<T>$, where $T$ is a numeric type such as $double$ or $int$. For example “$List<float>$ x” is a sequence variable but “$List<string>$ x” is not. User defined types that implement the $IEnumerable$ interface are not considered sequence variables, instead they are considered generic variables. The state machine for sequence variables is shown in Figure 3.7.

![Sequence Variable State Machine](image)

Figure 3.7: Sequence Variable State Machine

The following example illustrates the state machine transitions. Let “$List<double>$ x” be a sequence variable. Initially the state machine is in the state “No Data”. Suppose $x$ takes on the value $\{1, 1, 1\}$ then the state machine will transition to the state “Equal”, since “Equal” has the strongest constraints. Next, suppose $x$ takes on the value $\{2, 2, 3, 4, 5\}$ then the state machine will transition from the state “Equal” to the state “Non-Decreasing”, since the state “Non-Decreasing” has the strongest constraints that still satisfy the observed variable values. Finally, if $x$ takes on
the value \{3, 2\} then the state machine transitions to the state “No Constraints”, since none of the possible sequence constraints satisfy the observed variable values.

3.3 Precondition versus Postcondition Inference

Preconditions and postconditions are distinguished by preconditions being constraints on the parameter values passed to a method along with the program state and postconditions being constraints on the method return value along with the resulting program state. In particular, for methods with a void return type no postconditions are inferred, similarly for parameterless methods no preconditions are inferred. The design, of using constraint relaxation, is the same for preconditions and postconditions. For the inference process, both return values and parameter values are defined by a data type and the set of observed values.

3.4 Applications and Examples of IcsRewrite

IcsRewrite was developed to take a .NET program’s bytecode and rewrite the bytecode to include profiling information. When the rewritten .NET program is executed the added bytecode does runtime analysis to infer possible contracts. Figure 3.8 shows the results of running a C# program after its bytecode has been rewritten.
The following example illustrates using IcsRewrite. Figure 3.9 shows a program that finds the minimum of only the first and last values in a list. We want to infer the pre & post conditions for “FindMinFirstLast”. From inspecting the code, it is apparent that “FindMinFirstLast” has several flaws, in particular it does not check that “values” is non-null and contains at least two elements.
using System;
using System.Collections.Generic;
using System.Diagnostics;
namespace MinExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Debug.Assert(MinFirstLast(new[] { 1, 2, 3 }) == 1);
            Debug.Assert(MinFirstLast(new[] { 3, 2, 1 }) == 1);
            Debug.Assert(MinFirstLast(new[] { 1, 2 }) == 1);
            Debug.Assert(MinFirstLast(new[] { 2, 1 }) == 1);
            Debug.Assert(MinFirstLast(new[] { 2, 2 }) == 2);
        }
        static int MinFirstLast(int[] values)
        {
            return Math.Min(values[0], values[values.Length - 1]);
        }
    }
}

Figure 3.9: MinFirstLast Test Suite

In Figure 3.9 the Main function serves as the test suite for MinFirstLast. We note that it’s best
to have a comprehensive test suite which fully exercises the functions we are inferring contracts
for. By “comprehensive” we mean the test suite explores most of the function’s execution paths.
In the case of MinFirstLast the test suite (i.e. Main) is comprehensive, since it executes all of the
valid execution paths.

For this example the results of running the program after it has been instrumented by IcsRewrite are two preconditions and two postconditions. The preconditions are values ! = null and
values.length >= 2, the postconditions are ReturnValue >= Math.Min(values) and ReturnValue
<= Math.Max(values), where ReturnValue is the value returned by MinFirstLast. Figure 3.10
shows the commands for running IcsRewrite on MinFirstLast and the resulting output.
Commands
IcsRewrite.exe MinFirstLast.exe Instrumented_MinFirstLast.exe
Instrumented_MinFirstLast.exe Inferred_Contracts.txt

Contracts Inferred (written to Inferred_Contracts.txt)
Inferred PreConditions for Program.MinFirstLast
Contract.Assert(values ! = null);
Contract.Assert(values.length >= 2);

Inferred PostConditions for Program.MinFirstLast
Contract.Ensures(Contract.Result < int > >= Math.Min(values));
Contract.Ensures(Contract.Result < int > <= Math.Max(values));

Figure 3.10: Results MinFirstLast
CHAPTER 4
ICSREWRITE RESULTS

The design and implementation of IcsRewrite are checked using a suite of four test programs. The test programs exhibit behavior to check the handling of: 1) numeric pre/postconditions, 2) sequence pre/postconditions, and 3) generic pre/postconditions.

To check the inference of pre/postconditions for numeric variables, the test program NumericTests was written, similarly the test program SequenceTests was written to check inference of sequence pre/postconditions. For generic pre/postconditions the program ParamTests was written that exhibits the type of behavior typical of C# applications.

The tests show that the constraint relaxation design and implementation operates as expected. The tests show several areas where the design and implementation can both be improved. In particular the design uses no probabilistic reasoning to justify that the inferred contracts are true, given the data observed. For example, suppose the method, \texttt{void M1(int x)}, that’s tested just once, via the call \texttt{M1(1)}, then under the constraint relaxation design the precondition \texttt{Contract.Require(x == 1)} is inferred for M1. From a statistical perspective, inferring this precondition is not statistically justified. This line of reasoning is explored in Chapter 6.

4.1 Numeric Pre/PostConditions

The design and implementation of the constraint relaxation state machine for numeric variables were tested. In particular, testing was done for the constraint state transitions: NoData $\rightarrow$ RangeLimit, RangeLimit $\rightarrow$ NonZero, RangeLimit $\rightarrow$ NoConstraints, and NonZero $\rightarrow$ NoConstraints. Figure 4.1 shows the test methods and test data; the inferred contracts are shown in Figure 4.2. An area of improvement for the contract inference is smarter range limits. For example, suppose the method \texttt{void F(int x)} exists, and it’s called with the values 1, 2, and 10, then $[1,10]$ is inferred, as the range limit. A smarter inference is either $[1, \text{int.Max}]$ or $[0, \text{int.Max}]$, as the range limit.
using System.Diagnostics.Contracts;
using ContractInference; using System;
namespace ContractTests
{
    public class NumericTests
    {
        public static void Main()
        {
            Tst1(0); Tst1(10); Tst1(5); Tst1(3);
            Tst2(0); Tst2(20); Tst2(int.MaxValue); Tst2(4);
            Tst3Slope(-1 * int.MaxValue, -1 * int.MaxValue);
            Tst3Slope(1 * int.MaxValue, -1 * int.MaxValue);
            Tst3Slope(-1 * int.MaxValue, 1 * int.MaxValue);
            Tst3Slope(int.MaxValue, int.MaxValue);
            Tst3Slope(int.MaxValue, 0); Tst3Slope(int.MinValue, 0);
            Tst4_Y_Interceptor(1,1,2,2);
            Tst4_Y_Interceptor(int.MinValue,int.MinValue,int.MaxValue,int.MaxValue);
            Tst4_Y_Interceptor(int.MinValue,int.MinValue+1,int.MinValue+1,int.MaxValue);
            Tst4_Y_Interceptor(0,int.MinValue,1,1);
            Tst4_Y_Interceptor(-1,1,int.MaxValue,1);
            Tst4_Y_Interceptor(0,0,1,-1);
            Tst5_Ln(1); Tst5_Ln(0.0001);
            Tst5_Ln(int.MaxValue); Tst5_Ln(long.MaxValue);
            Tst5_Ln(float.MaxValue); Tst5_Ln(double.MaxValue);
            Tst5_Ln(Math.E); Tst5_Ln(Math.PI);
        }
        public static void Tst1(short s1){ return; }
        public static int Tst2(int s2){ return (int)Math.Sqrt(s2); }
        public static double Tst3Slope(double dx, double dy){ return dy/dx; }
        public static double Tst4_Y_Interceptor(double x1, double y1, double x2, double y2){
            double dx = , dy = ;
            return y1 - x1/((x2 - x1) * (y2 - y1)); // y-intercept.
        }
        public static double Tst5_Ln(double x){
            return Math.Log(x);
        }
    }
}

Figure 4.1: NumericTests.cs

The results validated the implementation of the constraint relaxation state machine for numeric variables.
4.2 Sequence Pre/PostConditions

The design and implementation of the constraint relaxation state machine for sequence variables were tested. Figure 4.3 shows the test methods and test data. In particular, the constraint state transitions were tested. Similarly to the numeric pre/postconditions, areas for improvement in the implementation of the constraints were found. In particular, the range limits can be improved. A corner case for the ordering constraints are single element sequences used as tests. For this case, instead of inferring that the ordering constraint state is “Equal”, instead no constraints should be outputted, since the data doesn’t justify inferring the “Equal” constraint.

Microsoft’s implementation of Design by Contract™ provides an interface for sequence preconditions in the form of the method “Contract.ForAll(...)”, which allows checking of the sequence elements. For postconditions no such method is provided, and as such IcsRewrite does not output ordering constraints for postconditions (they’re still inferred, they simply aren’t outputted). The inferred contracts for the sequence constraint tests are shown in Figure 4.4.

```
Contract.Requires(x2 != 0)
Contract.Requires(y2 != 0)
Begin PreConditions ContractTests.NumericTests.Test5_Ln
Contract.Requires(x != 0)
End PreConditions ContractTests.NumericTests.Test5_Ln
```
using System;
using System.Collections.Generic;
using System.Diagnostics.Contracts;
using System.Linq;

namespace ContractTests
{
    public class SequenceTests
    {
        public static void Main()
        {
            SequenceTest1(new List<short>(new short[] { 1 }));
            SequenceTest2(new int[] { 1, 2, 3 });
            SequenceTest3(new int[] { 3, 2, 1 });
            SequenceTest4(new int[] { int.MinValue, int.MaxValue });
            SequenceTest5(new int[] { int.MinValue, int.MaxValue });
            SequenceTest5(new int[] { int.MaxValue, int.MinValue });
        }
        public static void SequenceTest1(List<short> s) { return; }
        public static int SequenceTest2(int[] s) { return s.Sum(); }
        public static int SequenceTest3(int[] s) { return s.Length; }
        public static int SequenceTest4(int[] s) { return s.Min(); }
        public static int SequenceTest5(int[] s) { return s.Max(); }
    }
}
4.3 Generic Pre/PostConditions

Inference of preconditions and postconditions for variables of other types was tested, such as `string` and `string[]`. Figure 4.5 shows the test methods and test data; the inferred contracts are shown in Figure 4.6. In particular, testing was done for the constraint state transitions: NoData → ValueSet, NoData → NoConstraints, ValueSet →NonNull, ValueSet → NoConstraints, and
NonNull → NoConstraints. The design and implementation of outputting the constraints for the ValueSet state could be reworked. Currently, “object.GetHashCode” is used for obtaining a canonical representation for objects in the set of allowed values. This could be better, since the outputted contracts are not very informative for the programmer. Using XmlSerialization or a similar library would yield contracts that are more informative to the programmer. The inferred contracts for the ParamTest are shown in Figure 4.6
using System;
using System.Collections.Generic;
using System.Diagnostics.Contracts;
using System.Linq;
using ContractInference;
namespace ContractTests
{
    public class ParamTests
    {
        public static void Main()
        {
            ParamTest1(string.Empty); ParamTest1("paramTest1");
            for (int i = 0; i < 100; i++)
            {
                ParamTest2(i.ToString());
                ParamTest3(new List<string>() { "1" });
                ParamTest3(new List<string>() { "1" });
                ParamTest3(new List<string>() { "2" });
                ParamTest4(new List<string>() { "1" });
                ParamTest4(new List<string>() { "2" });
                ParamTest4(new List<string>() { "3" });
                ParamTest4(new List<string>() { "4" });
                ParamTest5(new List<string>() { "1" });
                ParamTest5(null);
                ParamTest6(new List<string>() { "1" });
                ParamTest6(new List<string>() { null });
            }
            public static void ParamTest1(string s) { return; }
            public static int ParamTest2(string s) { return 2; }
            public static int ParamTest3(List<string> s) { return 3; }
            public static int ParamTest4(List<string> s) { return 4; }
            public static int ParamTest5(List<string> s) { return 5; }
            public static int ParamTest6(List<string> s) { return 6; }
        }
    }

    public static void ParamTest1(string s) { return; }
    public static int ParamTest2(string s) { return 2; }
    public static int ParamTest3(List<string> s) { return 3; }
    public static int ParamTest4(List<string> s) { return 4; }
    public static int ParamTest5(List<string> s) { return 5; }
    public static int ParamTest6(List<string> s) { return 6; }
}

Figure 4.5: ParamTests.cs
Contract.Ensures(s == "" || s == "paramTest1");
End Preconditions ContractTests.ParamTests.ParamTest1
Contract.Ensures(s != null);
Contract.Ensures(s.GetHashCode() == 1982240768 || s.GetHashCode() == -961254464);

Figure 4.6: ParamTestsInferredContracts.txt
CHAPTER 5
ICSREWRITE PERFORMANCE EVALUATION

The time for adding program instrumentation and the impact of the added instrumentation on program performance was evaluated. The performance of the annotated code after the inferred contracts were included was not analyzed. The analysis focused on programs with no user interaction i.e. once the program starts it finishes, without user interaction.

5.1 Instrumentation Performance

The time used for program instrumentation was measured. The performance measure is time used for rewriting a program into its instrumented version e.g. to instrument ExampleProgram into InstrumentedExampleProgram.

The time used for program instrumentation against the number of methods in the program was measured. For data, the set of programs that were already developed for testing contract inference were used. In particular, the time for instrumenting NumericTests, SequenceTests, ParamTests, and GraphTests was measured. GraphTests was written to exhibit behavior closer to production programs that accomplish a task (the task for GraphTests is to take an ASCII image, parse it, and output an equivalent PNG image).

Figure 5.1 shows the results of measuring the time to instrument the test programs e.g. rewriting NumericTests into InstrumentedNumericTests. Each program was instrumented three times, these are the Time1, Time2, and Time3 columns. The “Average Time” column is the average of Time1, Time2, and Time3. The “# Methods” column is the number of methods in the program. As Figure 5.1 shows, the time for instrumentation is positively correlated with the number of methods. Figure 5.1 also shows that program instrumentation typically uses little time and since program instrumentation is only done once, optimizing it is not a priority.
5.2 Instrumented Program Performance

The performance impact of running with the instrumentation was evaluated. This was only for the purpose of determining the impact of instrumentation on program performance.

The purpose of measuring the impact of instrumentation on program performance is for use in automated build workflows. For example, IcsRewrite could be used as one of the steps in a project’s automated build. Adding IcsRewrite as part of a project’s automated build has the benefit that contract inference would be computed on every check-in. In this case, optimizing the instrumented program performance is useful, since the instrumented program is run on every check-in.

The total time was measured for running the instrumented version of the program versus the total time for running the non-instrumented version of the program. In Figure 5.1 the Time1, Time2, and Time3 columns are the time taken by the program to execute. Each program was executed three times to even out the execution times. The “Average Time” column is the average of the Time1, Time2, and Time3 columns. The results show that instrumented version of each was substantially slower than the original version. The slowdown, due to the added instrumentation, varied from 14x to 3x. For future work, it would be very useful to reduce this large performance impact.
<table>
<thead>
<tr>
<th>Program</th>
<th>Time1</th>
<th>Time2</th>
<th>Time3</th>
<th>Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumericTests</td>
<td>0.090s</td>
<td>0.046s</td>
<td>0.077s</td>
<td>0.071s</td>
</tr>
<tr>
<td>InstrumentedNumericTests</td>
<td>1.216s</td>
<td>0.951s</td>
<td>0.919s</td>
<td>1.029s</td>
</tr>
<tr>
<td>SequenceTests</td>
<td>0.111s</td>
<td>0.093s</td>
<td>0.084s</td>
<td>0.096s</td>
</tr>
<tr>
<td>InstrumentedSequenceTests</td>
<td>0.317s</td>
<td>0.346s</td>
<td>0.324s</td>
<td>0.329s</td>
</tr>
<tr>
<td>ParamTests</td>
<td>0.099s</td>
<td>0.100s</td>
<td>0.094s</td>
<td>0.098s</td>
</tr>
<tr>
<td>InstrumentedParamTests</td>
<td>1.277s</td>
<td>1.065s</td>
<td>1.055s</td>
<td>1.132s</td>
</tr>
<tr>
<td>GraphTests</td>
<td>0.417s</td>
<td>0.402s</td>
<td>0.393s</td>
<td>0.404s</td>
</tr>
<tr>
<td>InstrumentedGraphTests</td>
<td>21.483s</td>
<td>23.801s</td>
<td>23.703s</td>
<td>22.996s</td>
</tr>
</tbody>
</table>

Figure 5.2: Instrumented versus Non-Instrumented Time
CHAPTER 6
CONCLUSION AND FUTURE WORK

The expense in time and effort of annotating existing programs to include design contracts, was the motivation for this project and for creating the tool named IcsRewrite, which was developed to reduce this cost.

IcsRewrite provides the benefit of automatically inferring likely design contracts in an existing .NET program. The inferring of contracts performed by IcsRewrite is based on a set of constraint relaxation algorithms specifically developed for this purpose.

This project demonstrated that contract inference is a viable method of automatically adding preconditions and postconditions to C# programs. In particular, the constraint relaxation algorithms developed for this purpose are a reasonable design for contract inference.

Areas of future work are more intelligent constraint relaxation, improved output of contracts for complex structures, and performance improvement of instrumented programs. Details on possible future work for each of the three areas are:

- **Improved Constraint Relaxation**
  As Chapter 4 said, some of the contracts inferred via constraint relaxation are not statistically justified. An area of future work is adding statistical analysis in the constraint relaxation process.

- **Better Contract Output**
  As mentioned in the section on generic preconditions and postconditions in Chapter 4, we do not output programmer friendly contracts for types that are neither numeric nor sequence. An improvement would use the .NET reflection API for more sophisticated contract output.

- **Improved Performance**
  Because the performance of instrumented programs is drastically reduced, it is worthwhile to analyze the cause(s) of the degradation and, if possible, substantially reduce it.
# define CONTRACTS_FULL
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Mono.Cecil;
using Mono.Cecil.Cil;
using System.Diagnostics.Contracts;

namespace ContractInference
{
    public class Rewriter
    {
        public AssemblyDefinition Assembly {get; set;}
        public Rewriter (AssemblyDefinition assembly)
        {
            Assembly = assembly;
        }

        public string Rewrite(string outputFile)
        {
            if (Assembly == null){
                return "failure";
            }
            foreach (ModuleDefinition module in Assembly.Modules) {
                //Console.WriteLine("module.FullyQualifiedName " + module.FullyQualifiedName);
                var allMethods =
                    from type in module.Types
                    from method in type.Methods
                    select method;
                foreach (MethodDefinition method in allMethods.ToArray()) {
                    this.RewriteMethod (module, method);
                }
            }
            Assembly.Write(outputFile);
        return "success";
        }

        public void RewriteMethod ( ModuleDefinition module, MethodDefinition method)
        {
            //if (this.rewrittenMethods.Contains(method)) {
            //    return;
            //}
            //var overridden = this.GetOverriddenMethod (method);
            //if (overridden != null) {
            //    this.RewriteMethod (module, overridden, contractsRuntime);
            //}
            bool anyRewrites = false;
            var baseMethod = this.GetBaseOverriddenMethod (method);
            //Console.WriteLine("baseMethod != method: " + (baseMethod != method));
            if (baseMethod != method) {
                //...
            }
        }
    }
}
// TODO
/*
// Contract inheritance must be used
var vOverriddenTransform = this.rewrittenMethods[baseMethod];
// Can be null if overriding an abstract method
if (vOverriddenTransform != null) {
    if (this.options.Level >= 2) {
        // Only insert re-written contracts if level >= 2
        foreach (var inheritedRequires in vOverriddenTransform.ContractRequiresInfo) {
            this.RewriteIL(method.Body, null, null, inheritedRequires.RewrittenExpr);
            anyRewrites = true;
        }
    }
} */

// Console.WriteLine("method.HasBody: " + method.HasBody);
if (method.HasBody) {
    this.RewriteIL(method.Body, module);
    anyRewrites = true;
} // TODO
/*
var vTransform = this.TransformContracts(module, method, contractsRuntime);
if (vTransform.ContractRequiresInfo.Any()) {
    anyRewrites = true;
} */

// this.rewrittenMethods.Add(method, vTransform);
/* if (anyRewrites) {
    Console.WriteLine(method);
} */

private void RewriteIL(MethodBody body, ModuleDefinition module)
{
    var numParams = body.Method.Parameters.Count;
    if (numParams == 0)
        return;
    // Console.WriteLine("Enter: RewriteIL");
    // Console.WriteLine("RewriteIL:method:{0}". body.Method.Name);
    InsertMethodEnterInstructions(body, module);
    InsertMethodImplExitInstructions(body, module);
}

public void InsertMethodImplExitInstructions(MethodBody body, ModuleDefinition module)
{
    var il = body.GetILProcessor();
    List<Tuple<Instruction, Instruction>> insertions = new List<Tuple<Instruction, Instruction>>();
    for (int i = 0; i < body.Instructions.Count; i++)
    {
        {
            insertions.AddRange(InsertMethodImplExitInstruction(body.Instructions[i], body, module));
        }
    }
    foreach (var insertion in insertions)
    {
        il.InsertBefore(insertion.Item1, insertion.Item2);
    }
}
public List<
    Tuple<
        Instruction, Instruction>
    >> InsertMethodExitInstructions(
        Instruction instInsertBefore, MethodBody body, ModuleDefinition module)
    {
        var outputMethodExitInfo = typeof (Rewriter).GetMethod("MethodExit");
        MethodReference methodReference = module.Import(outputMethodExitInfo);
        var returnType = body.Method.MethodReturnType.ReturnType;
        // Nothing to do if method doesn't return anything.
        if (returnType.FullName == typeof (void).FullName)
            return new List<
                Tuple<
                    Instruction, Instruction>>();
        List<
            Tuple<
                Instruction, Instruction>> instructions = new List<
                Tuple<
                    Instruction, Instruction>>();
        // need to box.
        if (returnType.IsPrimitive)
            if (returnType.FullName == "System.Int32")
                instructions.Add(Instruction.Create(OpCodes.Box, module.Import(typeof(int))));
            else if (returnType.FullName == "System.Int64")
                instructions.Add(Instruction.Create(OpCodes.Box, module.Import(typeof(long))));
            else if (returnType.FullName == "System.Single")
                instructions.Add(Instruction.Create(OpCodes.Box, module.Import(typeof(float))));
            else if (returnType.FullName == "System.Double")
                instructions.Add(Instruction.Create(OpCodes.Box, module.Import(typeof(double))));
            else if (returnType.FullName == "System.Boolean")
                instructions.Add(Instruction.Create(OpCodes.Box, module.Import(typeof(bool))));
            else
                throw new Exception(string.Format("Unknown declaring type \{0\} ", returnType.DeclaringType));
        instructions.Add(Instruction.Create(OpCodes.Call, methodReference));
        List<
            Tuple<
                Instruction, Instruction>> insertions = new List<
                Tuple<
                    Instruction, Instruction>>();
        foreach (var inst in instructions)
            insertions.Add(new Tuple<
                Instruction, Instruction>(instInsertBefore, inst));
        return insertions;
    }

public void InsertMethodEnterInstructions(MethodBody body, ModuleDefinition module)
    {
        Instruction instInsertBefore;
        var il = body.GetILProcessor();
        var numParams = body.Method.Parameters.Count;
        instInsertBefore = body.Instructions[0];
        var outputMethodEnterInfo = typeof (Rewriter).GetMethod("MethodEnter");
        MethodReference methodReference = module.Import(outputMethodEnterInfo);
        var returnType = module.Import(typeof(object));
        var ldcCountInst = Instruction.Create(OpCodes.Ldc_I4, numParams);
        var createArrayInst = Instruction.Create(OpCodes.Newarr, objectTypeReference);
        List<
            Instruction> instructions = new List<
                Instruction>();
        instructions.Add(ldcCountInst);
        instructions.Add(createArrayInst);
        foreach (var param in body.Method.Parameters)
            {
            var argIdx = param.Index + (body.Method.IsStatic ? 0 : 1);
            instructions.Add(Instruction.Create(OpCodes.Dup));
            instructions.Add(Instruction.Create(OpCodes.Ldc_I4, param.Index));
            switch (argIdx) {
case 0:  
    instructions. Add(Instruction.Create(OpCodes.Ldarg.0));  
    break;

case 1:  
    instructions. Add(Instruction.Create(OpCodes.Ldarg.1));  
    break;

case 2:  
    instructions. Add(Instruction.Create(OpCodes.Ldarg.2));  
    break;

case 3:  
    instructions. Add(Instruction.Create(OpCodes.Ldarg.3));  
    break;

default:  
    instructions. Add(Instruction.Create(OpCodes.Ldarg, argIdxs));  
    break;
}

// need to box before adding to array.
if (paramType.IsPrimitive) {
    if (paramType.FullName == "System.Int16"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Short))));
    } else if (paramType.FullName == "System.Int32"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Int))));
    } else if (paramType.FullName == "System.Int64"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Long))));
    } else if (paramType.FullName == "System.Single"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Float))));
    } else if (paramType.FullName == "System.Double"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Double))));
    } else if (paramType.FullName == "System.Boolean"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Boolean))));
    } else if (paramType.FullName == "System.Char"){
        instructions. Add(Instruction.Create(OpCodes.Box, module.Import(typeof(Char))));
    } else {
        throw new Exception(string.Format("Unknown declaring type {0}", paramType.FullName));
    }
}

instructions. Add(Instruction.Create(OpCodes.Stelem_Ref));

foreach(var inst in instructions){
    il.InsertBefore(instInsertBefore, inst);
}

public static Dictionary<string, List<PreCondition>> ics = new Dictionary<string, List<PreCondition>>();
public static Dictionary<string, PostCondition> PostConditions = new Dictionary<string, PostCondition>();

public static Dictionary<string, int> NumCallsMethod = new Dictionary<string, int>();
public static Dictionary<string, bool> ContractOutputted = new Dictionary<string, bool>();

const int NumCallToOuputContract = 2;
public static void MethodExit(object returnValue)
{
  var stackTrace = new System.Diagnostics.StackTrace(1);

  // Avoid self referential loops (e.g., if an app overloads Class.Equals then we can
  // enter a self recursive loop)
  foreach (var frame in stackTrace.GetFrames()) {
    if ((frame.GetMethod().Name == "MethodEnter" ||
         frame.GetMethod().Name == "MethodExit") &&
        frame.GetMethod().Module.Name == "ContractInference.dll")
    {
      return;
    }
  }

  var stackFrame = stackTrace.GetFrame(0);
  if (mi == null || mi.ReturnType == typeof(void))
  {
    return;
  }

  string methodName = method.DeclaringType + "." + method.Name;
  if (!PostConditions.ContainsKey(methodName)){
    PostCondition pc = new ContractInference.PostCondition {
      MethodName = methodName,
      VarType = mi.ReturnType,
      pc.Constraints = ConstraintFactory.CreateConstraint(mi.ReturnType),
    pc.Constraints.AddDataPoint(returnValue);,
    PostConditions[methodName] = pc;
  }

  OutputContracts();
}

public static void MethodEnter(object[] args)
{  
  Contract.Assert(args != null);
  string txt = string.Empty;
  var stackTrace = new System.Diagnostics.StackTrace(1);
  // Avoid self referential loops (e.g., if an app overloads Class.Equals then we can
  // enter a self recursive loop)
  foreach (var frame in stackTrace.GetFrames()) {
    if ((frame.GetMethod().Name == "MethodEnter" ||
         frame.GetMethod().Name == "MethodExit") &&
        frame.GetMethod().Module.Name == "ContractInference.dll")
    {
      return;
    }
  }

  var stackFrame = stackTrace.GetFrame(0);
  var method = stackFrame.GetMethod();
  var paramInfos = method.GetParameters();
  txt += "Implicit Contracts: Begin: Method enter\n";
  string methodName = method.DeclaringType + "." + method.Name;
  if (!ics.ContainsKey(methodName)){
    ics.Add(methodName, new List<PreCondition >());
    NumCallsMethod[methodName] = 1;
    ContractOutputted[methodName] = false;
  }

  else{
    NumCallsMethod[methodName]++;
  }
}
for (int paramIdx = 0; paramIdx < args.Length; paramIdx++)
    {
        var contracts = ics[methodName];
        // ImplicitContract ic = new ImplicitContract {MethodName = methodName, Contract = new ParamContract()};
        ic.Contract.VarName = paramInfos[paramIdx].Name;
        ic.Contract.ParamType = paramInfos[paramIdx].GetType();
        // ic.Contract.ParamValue = args[paramIdx];
        var existingContracts = contracts.FindAll(c => c.Contract.VarName == paramInfos[paramIdx].Name);
        System.Diagnostics.Debug.Assert(existingContracts.Count <= 1);
        if (existingContracts.Count == 1)
            {
                existingContracts[0].Contract.Constraints.addDataPoint(args[paramIdx]);
            }
        else
            {
                PreCondition ic = new PreCondition {MethodName = methodName, Contract = new VarContract()};
                ic.Contract.VarName = paramInfos[paramIdx].Name;
                ic.Contract.VarType = paramInfos[paramIdx].ParameterType;
                ic.Contract.Constraints = ConstraintFactory.CreateConstraint(ic.Contract.VarType);
                ic.Contract.Constraints.addDataPoint(args[paramIdx]);
                contracts.Add(ic);
            }
        PrintContracts();
    }
Console.WriteLine("Implicit Contracts: Finished: Method enter");
OutputContracts();

public static void PrintContracts()
{
    Console.WriteLine("Begin PrintContracts");
    foreach (var c in ics)
        {
        Console.WriteLine("c.Key: {0}", c.Key);
        foreach (var cc in c.Value)
            {PrintMethodContract(cc);}
    }
    Console.WriteLine("End PrintContracts");
}

public static void PrintMethodContract(PreCondition c)
{
    Console.WriteLine("Begin PrintContract, {0}, {1}, c.MethodName, c.Contract.VarName");
    Console.WriteLine("End PrintContract, {0}, {1}, c.MethodName");
}

// private static bool firstOutput = true;
public static void OutputContracts()
{
    /* if (firstOutput)
    {
        System.IO.File.WriteAllText("InferredContracts.txt", string.Empty);
        firstOutput = false;
    }*/
    System.IO.File.WriteAllText(GetAssemblyName() + "InferredContracts.txt", string.Empty);
    System.IO.File.WriteAllText(GetAssemblyName() + "InferredContracts.txt", string.Empty);
    var existingMethods = contracts.GetMethods();
private static string GetAssemblyName()
{
    return System.Reflection.Assembly.GetEntryAssembly().GetName().Name ?? string.Empty;
}

public static void OutputMethodContracts(string methodName)
{
    Contract.Assert(methodName != null);
    Contract.Assert(methodName != string.Empty);
    Contract.Assert(NumCallsMethod != null);
    Contract.Assert(NumCallsMethod.ContainsKey(methodName));
    /* if (NumCallsMethod[methodName] != NumCallToOutputContract ||
        ContractOutputted[methodName] == true){
        return;
    }
    ContractOutputted[methodName] = true; */
    foreach (var contract in ics[methodName])
    {
        OutputContract(contract);
    }
    if (PostConditions.ContainsKey(methodName))
    {
        OutputPostConditionContract(PostConditions[methodName]);
    }
}

public static void OutputContract(PreCondition contract)
{
    string contractTxt = string.Empty;
    {
        contractTxt += string.Format("Begin PreConditions {0}\n", contract.MethodName);
        contractTxt += contract.Contract.Constraints.PrintPreCondition(contract.Contract.VarName);
        contractTxt += string.Format("End PreConditions {0}\n", contract.MethodName);
    }
    System.IO.File.AppendAllText(GetAssemblyName() + "InferredContracts.txt", contractTxt);
}

public static void OutputPostConditionContract(PostCondition contract)
{
    string contractTxt = string.Empty;
    if (!string.IsNullOrEmpty(contract.Constraints.PrintPostCondition(contract.VarType)))
    {
        contractTxt += string.Format("Begin PostConditions {0}\n", contract.MethodName);
        contractTxt += contract.Constraints.PrintPostCondition(contract.VarType);
        contractTxt += string.Format("End PostConditions {0}\n", contract.MethodName);
    }
    System.IO.File.AppendAllText(GetAssemblyName() + "InferredContracts.txt", contractTxt);
}

private MethodDefinition GetOverriddenMethod(MethodDefinition method)
{
    if (method.IsNewSlot || !method.IsVirtual) {
        return null;
    }
    var baseType = method.DeclaringType BaseType;
    if (baseType == null) {

```csharp
using System;
namespace ContractInference
{
    public class ConstraintRelaxation
    {
        private ConstraintRelaxation()
        {
        }
    }
}
```

```csharp
using System;
namespace ContractInference
{
    public class VarContract
    {
        public string VarName { get; set; }
        /* NOT_IMPLEMENTED_FOR_MS_PROJECT */
        public bool IsNullCheck { get; set; }
        public bool IsPropertyContract { get; set; }
        public bool IsNullContract { get; set; }
        public string PropertyName { get; set; }
        public object PropertyValue { get; set; }
        public object ParamValue { get; set; }
        
        public Type VarType
        {
            get;
            set;
        }
        
        public IContractConstraint Constraints { get; set; }
    }
}
```
```csharp
using System;
using System.Collections.Generic;

namespace ContractInference
{
    public class PreCondition
    {
        public string MethodName { get; set; }
        public VarContract Contract { get; set; }
        public PreCondition ()
        {
            MethodName = string.Empty;
        }
    }
}
```

```csharp
using System;
namespace ContractInference
{
    public class PostCondition
    {
        public string MethodName { get; set; }
        public Type VarType
        {
            get;
            set;
        }
        public IContractConstraint Constraints { get; set; }
        public PostCondition ()
        {
            MethodName = string.Empty;
        }
    }
}
```

```csharp
using System;
namespace ContractInference
{
    public interface IContractConstraint
    {
    }
}
```
```csharp
namespace ContractInference
{
    public class NumericContractConstraint : IContractConstraint
    {
        /∗
        − invariants over a single numeric variable: [0/4]
        − range limits: \( x \geq a, x \leq b, \) and \( a \leq x \leq b \) (printed as \( x \) in \([a..b]\)) indicate
        the
        minimum and/or maximum value
        − nonzero: \( x \neq 0 \) if the variable is never set to 0
        See Section 4.5 for details on when such an invariant is reported.
        − NOT_DONE_FOR_MS_PROJECT modulus: \( x = a \mod b \) indicates that \( x \mod b = a \) always
        − NOT_DONE_FOR_MS_PROJECT nonmodulus: \( x \neq a \mod b \), reported only if \( x \mod b \)
        takes on every value besides \( a \)
        ∗/

        public enum NumericConstraintState { NoData, RangeLimit, NonZero, NoConstraints }
        public Tuple<double, double> Range { get; set; }
        public long Modulus;
        public NumericConstraintState State;

        public NumericContractConstraint()
        {
            State = NumericConstraintState.NoData;
            Range = new Tuple<double, double>(0, 0);
        }

        public string PrintPreCondition(string varName)
        {
            return PrintConstraint(varName, true);
        }

        public string PrintPostCondition(Type returnType)
        {
            return PrintConstraint(string.Format("Contract.Result\(\{0\}\)\", returnType.FullName), false);
        }

        private string PrintConstraint(string varName, bool isAssert)
        {
            Contract.Requires(!string.IsNullOrEmpty(varName));
            var formattedConstraint = string.Empty;
            switch (State)
            {
                case NumericConstraintState.NoConstraints:
```
break;
case NumericConstraintState.NoData:
    break;
case NumericConstraintState.RangeLimit:
    string formattedMin = GetFormattedMin(Range.Item1);
    string formattedMax = GetFormattedMax(Range.Item2);
    formattedConstraint = string.Format("{0} <= \{1\} && \{1\} <= {2}\". formattedMin, varName, formattedMax);
    break;
case NumericConstraintState.NonZero:
    formattedConstraint = string.Format("{0} != 0\", varName);
    break;
default:
    throw new Exception("Un handled state " + State);
}   
if (formattedConstraint != string.Empty)
{   
    if (isAssert)
    {
        return Contract.Requires(" + formattedConstraint + ")" + Environment.NewLine;
    }
    else{
        return Contract.Ensures(" + formattedConstraint + ")" + Environment.NewLine;
    }
}
return formattedConstraint;

public static string GetFormattedMin(double x)
{
    if (x == short.MinValue)
    {
        return "short.MinValue";
    } else if (x == int.MinValue)
    {
        return "int.MinValue";
    } else if (x == long.MinValue)
    {
        return "long.MinValue";
    } else if (x == float.MinValue)
    {
        return "float.MinValue";
    } else if (x == double.MinValue)
    {
        return "double.MinValue";
    }
    return x.ToString();
}

public static string GetFormattedMax(double x)
{
    if (x == short.MaxValue)
    {
        return "short.MaxValue";
    } else if (x == int.MaxValue)
    {
        return "int.MaxValue";
    } else if (x == long.MaxValue)
    {
        return "long.MaxValue";
    } else if (x == float.MaxValue)
    {
        return "float.MaxValue";
    } else if (x == double.MaxValue)
    {
        return "double.MaxValue";
    }
```csharp
} else if (x == double.MaxValue)
{
    return "doubleValue";
}
return x.ToString();

public void AddDataPoint(object p_val)
{
    double val = Convert.ToDouble(p_val);
    if (val == 0)
    {
        State = NumericConstraintState.NoConstraints;
        return;
    }
    switch (State)
    {
        case NumericConstraintState.NoData:
            State = NumericConstraintState.RangeLimit;
            Range = new Tuple<double, double>(val, val);
            break;
        case NumericConstraintState.RangeLimit:
            Range = new Tuple<double, double>(Math.Min(Range.Item1, val), Math.Max(Range.Item2, val));
            if (Range.Item1 <= short.MinValue && Range.Item2 >= 0)
            {
                State = NumericConstraintState.NoConstraints;
            } else if (Range.Item1 <= short.MinValue && Range.Item2 < 0)
            {
                State = NumericConstraintState.NonZero;
            } else if (Range.Item2 >= short.MaxValue && Range.Item1 <= 0)
            {
                State = NumericConstraintState.NoConstraints;
            } else if (Range.Item2 >= short.MaxValue && Range.Item1 > 0)
            {
                State = NumericConstraintState.NonZero;
            }
            break;
        case NumericConstraintState.NonZero:
            if (val == 0)
            {
                State = NumericConstraintState.NoConstraints;
            }
            break;
        default:
        break;
    }
}

using System;
using System.Collections.Generic;
using System.Diagnostics.Contracts;

namespace ContractInference
{
    /* implements constraints and constraint relaxation for
     * invariants over any variable:
     * - constant value: x = a indicates the variable is a constant
     * - non-null: x != null indicates the variable is never null
     */
```
− small value set: \( x \in \{a, b, c\} \) indicates the variable takes on only a small number of different values

*/

```csharp
public class ParamContractConstraint : IContractConstraint
{
    public List<object> AllowedValues { get; set; }
    public ConstraintState State;

    public const int MAX_NUM_ALLOWED_VALUES = 5;

    public enum ConstraintState
    {
        NoData,
        ValueSet,
        NonNull,
        NoConstraints
    }

    public ParamContractConstraint ()
    {
        AllowedValues = new List<object>();
        State = ConstraintState.NoData;
    }

    public string PrintPreCondition(string varName)
    {
        return PrintConstraint(varName, true);
    }

    public string PrintPostCondition(Type returnType)
    {
        return PrintConstraint(string.Format("Contract.Result<{0}>()", returnType.FullName), false);
    }

    private string PrintConstraint(string varName, bool isAssert)
    {
        Contract.Requires(!string.IsNullOrEmptyOrWhiteSpace(varName));
        string formattedConstraint = string.Empty;
        switch (State)
        {
            case ConstraintState.NoData:
                formattedConstraint = string.Empty;
                break;
            case ConstraintState.ValueSet:
                foreach (object val in AllowedValues)
                {
                    if (val.GetType() == typeof(bool))
                    {
                        formattedConstraint += string.Format("{0} == {1} || ", varName, val);
                    }
                    else if (val.GetType() == typeof(string))
                    {
                        formattedConstraint += string.Format("{0} == \"{1}\" || ", varName, val);
                    }
                    else
                    {
```
formattedConstraint += string.Format("{0}.GetHashCode() == {1} || ", varName, 
val.GetHashCode());
}

formattedConstraint = formattedConstraint.TrimEnd().TrimEnd(new char[] {'}'));
break;
case ConstraintState.NotNull:
    formattedConstraint = string.Format("{0} != null", varName);
    break;
case ConstraintState.NoConstraints:
    break;
default:
    throw new Exception("Unhandled case: " + State);
}
if (formattedConstraint != string.Empty)
{
    if (isAssert)
    {
        return "Contract.Requires(" + formattedConstraint + ");" + Environment.NewLine;
    } else {
        return "Contract.Ensure(" + formattedConstraint + ");" + Environment.NewLine;
    }
}
return formattedConstraint;

public void AddDataPoint(object val)
{
if (val == null)
{
    AllowedValues = new List<object>();
    State = ConstraintState.NoConstraints;
    return;
}
switch (State){
case ConstraintState.NoData:
    State = ConstraintState.ValueSet;
    AllowedValues = new List<object>();
    break;
case ConstraintState.ValueSet:
    if (AllowedValues.Contains(val)) {
        // nothing to do.
    } else if (AllowedValues.Count < MAX_NUM_ALLOWED_VALUES)
    { AllowedValues.Add(val);
    } else {
        // AllowedValues.Count >= MAX_NUM_ALLOWED_VALUES
        AllowedValues = new List<object>();
        State = ConstraintState.NonNull;
    }
    break;
case ConstraintState.NonNull:
    // nothing to do, since val != null.
    break;
case ConstraintState.NoConstraints:
    // nothing to do, since already no constraints.
    break;
default:
    System.Diagnostics.Debug.Assert(false, "Unknown ConstraintState");
    break;
}
namespace ContractInference
{
    public class SequenceContractConstraint : IContractConstraint
    {
        // invariants over a single sequence variable:
        // range: minimum and maximum sequence values, ordered lexicographically
        // For instance, this can indicate the range of string or array values.
        // element ordering: whether the elements of each sequence are non-decreasing,
        // non-increasing, or equal.
        // In the latter case, each sequence contains (multiple instances of) a single value,
        // though that value may differ from sequence to sequence.
        // NOT_IMPLEMENTED_FOR_MS_PROJECT invariants over all sequence elements (treated as
        // a single large collection):
        // for example, in Figure 2.3 (page 11), all elements of array B are at least −100.
        // The sum invariants of Figure 2.3 do not appear here because sum(B) is a derived
        // variable, which is described in Section 4.3, page 32.
        
        public enum SequenceConstraintState {
            NoData,
            RangeOrOrdered,
            NoConstraints
        }

        public enum RangeConstraintState {
            NoData,
            Ranged,
            NoConstraints
        }

        public enum OrderingConstraintState {
            NoData,
            NonDecreasing,
            NonIncreasing,
            Equal,
            NoConstraints
        }

        public Tuple<double, double> Range { get; set; }
        public SequenceConstraintState State { get; set; }
        public OrderingConstraintState OrderingState { get; set; }
        public RangeConstraintState RangeState { get; set; }
        public bool IsArray { get; set; }

        private List<List<double>> values = new List<List<double>>();

        public SequenceContractConstraint()
        {
            State = SequenceConstraintState.NoData;
        }
    }
}
```csharp
OrderingState = OrderingConstraintState.NoData;
RangeState = RangeConstraintState.NoData;

public string PrintPreCondition(string varName)
{
    Console.WriteLine("Printing SequenceConstraint");
    Contract.Requires(!string.IsNullOrEmpty(varName));

    string formattedConstraint = string.Empty;
    switch (State)
    {
    case SequenceConstraintState.NoData:
        break;
    case SequenceConstraintState.NoConstraints:
        formattedConstraint = string.Empty;
        break;
    case SequenceConstraintState.RangeOrOrdered:
        formattedConstraint = GetFormattedRangedOrOrderedConstraint(varName);
        break;
    default:
        throw new Exception("Unhandled case: " + State);
    }
    return formattedConstraint;
}

public string PrintPostCondition(Type returnType)
{
    // TODO
    return string.Empty;
}

private string GetFormattedRangedOrOrderedConstraint(string varName)
{
    Contract.Requires(State == SequenceConstraintState.RangeOrOrdered);
    string orderingContract = string.Empty;
    string count = varName;
    count += IsArray ? ".Length" : ".Count()";
    switch (OrderingState)
    {
    case OrderingConstraintState.NoData:
        break;
    case OrderingConstraintState.NoConstraints:
        break;
    case OrderingConstraintState.Equal:
        orderingContract = string.Format("Contract.ForAll(0, {0}, i => {1}[i] == {1}[0]);", count, varName);
        break;
    case OrderingConstraintState.NonDecreasing:
        orderingContract = string.Format("Contract.ForAll(1, {0}, i => {1}[i] >= {1}[i - 1]);", count, varName);
        break;
    case OrderingConstraintState.NonIncreasing:
        orderingContract = string.Format("Contract.ForAll(1, {0}, i => {1}[i] <= {1}[i - 1]);", count, varName);
        break;
    }

    string rangeContract = string.Empty;
    switch (RangeState)
    {
    case RangeConstraintState.NoConstraints:
```
break;
case RangeConstraintState.NoData:
break;
case RangeContract = string.Format("Contract. ForAll(0, \{1\}, i => \{0\} <= \{2\}[i] && \{2\}[i] <= (3)\}. NumericContractConstraint.GetFormattedMin(Range.Item1), count, varName, NumericContractConstraint.GetFormattedMax(Range.Item2));
break;
default:
throw new Exception("Unhandled case: " + RangeState);
}
return orderingContract +
rangeContract +
(rangeContract != string.Empty ? Environment.NewLine : string.Empty);
}

public void AddDataPoint(object p_val)
{
Contract.Requires(p_val != null);
List<double> val = new List<double>();
if (p_val is short[])
{
IsArray = true;
foreach (var v in (short[])p_val)
{
val.Add(v);
}
}
else if (p_val is int[]) {
IsArray = true;
foreach (var v in (int[])p_val)
{
val.Add(v);
}
}
else if (p_val is long[]) {
IsArray = true;
foreach (var v in (long[])p_val)
{
val.Add(v);
}
}
else if (p_val is float[]) {
IsArray = true;
foreach (var v in (float[])p_val)
{
val.Add(v);
}
}
else if (p_val is double[]) {
IsArray = true;
foreach (var v in (double[])p_val)
{
val.Add(v);
}
}
else if (p_val is List<short>) {
foreach (var v in (List<short>)p_val)
{
val.Add(v);
}
}
else if (p_val is List<int>) {
foreach (var v in (List<int>)p_val)
{
val.Add(v);
}
}
else if (p_val is List<long>) {
    foreach (var v in (List<long>)p_val)
    {
        val.Add(v);
    }
}
else if (p_val is List<double>) {
    foreach (var v in (List<double>)p_val)
    {
        val.Add(v);
    }
} else {
    throw new Exception("Unhandled sequence type "+p_val.GetType().FullName);
}

if (!values.Contains(val)) values.Add(val);
Console.WriteLine("Begin SequenceConstraint: State {0}", State);
Console.WriteLine("Begin SequenceConstraint: OrderingState {0}", OrderingState);
Console.WriteLine("Begin SequenceConstraint: RangeState {0}", RangeState);
switch (State)
{
    case SequenceConstraintState.NoData:
        OrderingState = GetOrderingState(OrderingConstraintState.NoData, val);
        if (val.Count > 0)
        {
            RangeState = RangeConstraintState.Ranged;
            Range = new Tuple<double, double>(val.Min(), val.Max());
        } else {
            RangeState = RangeConstraintState.NoData;
        } break;
    case SequenceConstraintState.RangeOrOrdered:
        switch (RangeState){
            case RangeConstraintState.NoData:
                OrderingState = GetOrderingState(OrderingConstraintState.NoData, val);
                break;
            case RangeConstraintState.Ranged:
                Range = new Tuple<double, double>(Math.Min(Range.Item1, val.Min()), Math.Max(Range.Item2, val.Max()));
                if (Range.Item1 == double.MinValue && Range.Item2 == double.MaxValue)
                {
                    RangeState = RangeConstraintState.NoConstraints;
                } break;
            default:
                // nothing to do.
                break;
        } break;
    case SequenceConstraintState.NoConstraints:
        break;
    default:
    OrderingState = GetOrderingState(OrderingState, val);
    if (RangeState == RangeConstraintState.NoConstraints && OrderingState == OrderingConstraintState.NoConstraints)
    {
        State = SequenceConstraintState.NoConstraints;
    } break;
    case SequenceConstraintState.NoConstraints:
    break;
    default:
break;
}
Console.WriteLine("End SequenceConstraint: State {0}" , State);
Console.WriteLine("End SequenceConstraint: OrderingState {0}" , OrderingState);
Console.WriteLine("End SequenceConstraint: RangeState {0}" , RangeState);

private static OrderingConstraintState GetOrderingState(OrderingConstraintState currentState , List<double> val)
{
switch (currentState){
case OrderingConstraintState.NoData:
    return GetOrdering(val);
case OrderingConstraintState.Equal:
    return GetOrdering(val);
case OrderingConstraintState.NonDecreasing:
    var newOrdering = GetOrdering(val);
    if (newOrdering == currentState || newOrdering == OrderingConstraintState.Equal){
        return currentState;
    }
    return OrderingConstraintState.NoConstraints;
case OrderingConstraintState.NonIncreasing:
    var newOrdering2 = GetOrdering(val);
    if (newOrdering2 == currentState || newOrdering2 == OrderingConstraintState.Equal){
        return currentState;
    }
    return OrderingConstraintState.NoConstraints;
default:
    // nothing to do
    return currentState;
}

public static OrderingConstraintState GetOrdering(List<double> val)
{
    bool equal = true , nonDecreasing = true , nonIncreasing = true;
for (int i = 1; i < val.Count; i++){
    equal &= val[i] == val[i - 1];
    nonDecreasing &= val[i] >= val[i - 1];
    nonIncreasing &= val[i] <= val[i - 1];
}
if (equal){
    return OrderingConstraintState.Equal;
} else if (nonDecreasing){
    return OrderingConstraintState.NonDecreasing;
} else if (nonIncreasing){
    return OrderingConstraintState.NonIncreasing;
}
return OrderingConstraintState.NoConstraints;
}
References


