Counterexample Execution

Jürg Billeter

Master Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://pm.inf.ethz.ch/

August 2008

Supervised by:
Joseph N. Ruskiewicz
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Spec# programming system enables automatic program verification. However, if the verification fails, it is often impossible to understand the cause of the error from the returned error message. This thesis uses the counterexample model to produce a matching runtime state, which can be debugged similar to a runtime error occurring in a test case. This allows the programmer to understand the error using the familiar debugger interface to inspect variables and control the execution of the program.

Contents

Chapter 1

Introduction

Automatic program verification is used by an increasing number of programmers. Static analysis tools report program errors by the click of a button or even as the programmer types.

This can be very helpful as programmers do make mistakes. However, programmers do not only need to know that there is an error in the program, they also need to understand the mistake, which can be a lot more complex.

When a program fails at runtime, programmers have the possibility to use a debugger to assist them in finding the reason for the error. The debugger allows them to look at relevant fields and variables, hide information they do not need, step through the program, and look at the backtrace. The debugger is a well-known tool and programmers know how to use it to find the bug in their program.

In static verification we do not have a backtrace, and we do not have a debugger. We just know that the verification failed. However, modern verifiers have the ability to return a counterexample model. The model describes the state of the program where the verification condition does not hold.

The information in the model can be very useful, however, it can be very large even for small programs. The model may also refer to values that only exist in the intermediate representation of the program used by the verifier, that is, it may refer to internal variable names that do not exist in the original program.

The counterexample model can be compared to a state dump you may get for a runtime error. The information in the state dump can be very useful, however, it also contains many pieces that are not relevant. This is why programmers use a debugger instead of just reading the state dump.

For example, let us look at the following Spec# code, a simple method with a postcondition, that is, a condition that must hold when returning from the method:

```
public class Simple
{
    public int Test (int value)
    ensures result >= 10;
    {
        return value;
    }
}
```

Compiling and verifying it using ‘ssc /verify’ leads to the following error message:

```
Error: Method Simple.Test(int value), unsatisfied postcondition: result >= 10
```

This means that the verifier is unable to prove that the postcondition always holds, however, it does not mention a specific input value for which the condition fails. The verifier has an option to print the model of the corresponding counterexample, which shows the state of the program at the point of the failure.

Short excerpt from the corresponding counterexample model

```
partitions:
  [...] (50 irrelevant lines)
  *53: 0 {$result value return.value SS$Display.Return.Local}
  [...] (10 irrelevant lines)
  *64: 9 {value$in}
  [...] (20 irrelevant lines)
function interpretations:
  [...] (50 irrelevant lines)
```

As can be seen in the above example, the counterexample model is usually very long, even for small programs. However, it is not only long, it also uses internal identifiers using non-obvious prefixes and suffixes. $result, return.value, and SS$Display.Return.Local all refer to the return value of the method, and value$in refers to the input value of the parameter. These mappings are not straight forward if you compare them to the original Spec# code in the example, and they slow down reading the counterexample.

This thesis uses the counterexample model to produce a matching runtime state, which can be debugged similar to a runtime error occurring in a test case. This allows the programmer to use the familiar debugger interface to inspect fields and variables and control the execution of the program.

We reach the runtime state by inserting and replacing instructions in the compiled program in such a way that local variables and method call results exactly match the values specified in the model of the counterexample. Mock objects take the place of real objects to reproduce the behavior from the counterexample, and we generate a new entry point for the modified program. The only code in the entry point method initializes the parameters of the method that we are debugging and calls the method. We never call any other method, except for methods in mock objects.

There are various aspects that need to be taken into account when performing this transformation. Method calls, loops, and field assignments all have to follow the counterexample, and runtime checks for the various types of verification failures need to be inserted to make it possible to reproduce errors found by the verifier at runtime.

We discuss the implementation of our tool that is able to automatically transform code according to a counterexample and our experience in testing it.

Chapter 2

Background

2.1 Spec# language

The Spec# programming language is an extension of C# to include non-null types, method contracts, and object and loop invariants. Together with the Spec# compiler and Boogie[2], the Spec# program verifier, it forms the Spec# programming system[4].

Spec# supports the specification of method pre- and postconditions, which need to hold when entering and leaving the method, respectively. The caller of the method is responsible for satisfying the preconditions, and the called method is responsible for satisfying the postconditions.

To achieve modularity, Spec# defines an ownership system[14]. Each object in a Spec# program has either no or exactly one owner, and the owner cannot be changed during the object's lifetime. An object is a peer of another object if both share the same owner. By default, a method can only be called when the receiver object and all method parameters are peer consistent. An object is consistent when it has been fully constructed, the object is not exposed, and it has no owner or the owner is exposed. An object is peer consistent if the object and all of its peers are consistent.

A method is pure[8, 1] if it does not modify the state of any object. Method calls in a specification must always be calls to pure methods. A pure method does not have the requirement that the receiver object and all parameters need to be peer consistent. Instead, the objects need to be peer fully valid. An object is fully valid when it has been constructed and it is not exposed. An object is peer fully valid if the object and all of its peers are fully valid.

Object fields that are used in invariants may only be modified when the object is exposed. Object invariants[3] may be broken while an object is exposed, however, they need to hold again at the end of the expose statement.

Frame conditions limit what fields and objects a method may modify. By default, a method may only modify the fields of the 'this' instance. Deviating frame conditions may be specified using a *modifies* clause. A method may also modify objects that it owns without explicitly specifying that in the modifies clause. The limit also covers calls to other methods. That is, a method cannot call another method that has a less restrictive frame condition.

Spec# allows you to use quantifiers and comprehensions in verification conditions, as long as the compiler can generate code to compute them. This enables the programmer to apply stronger pre- and postconditions, a requirement for successful verification.

2.2 Spec# compiler

The Spec# compiler generates .NET assemblies from Spec# source code. In addition to normal program code generation, it emits runtime checks for the specified contracts and also stores the contracts in attributes. The assemblies can be executed by a standard .NET VM; the assembly format has not been extended.

The Spec# compiler generates runtime checks for method pre- and postconditions, loop invariants, and object invariants. This means that you get a runtime exception when violating the specified verification condition.

However, the compiler does not generate as many runtime checks as Boogie does during static verification. Expensive runtime checks as, for example, checking frame conditions are omitted to avoid performance issues when running the program.

2.3 Boogie

Boogie[2] is a static program verifier that attempts to automatically prove that specified verification conditions in Spec# programs hold.

Boogie does not attempt to verify a program as a whole; it follows a modular approach verifying a program method by method. This requires strong contracts that define the interfaces. Object ownership rules and expose statements are necessary to make the modularity possible.

If Boogie is unable to verify a program, it will return a verification failure. This does not necessarily mean that there is an error in the program code as it might also be a limitation of the verifier.

BoogiePL

BoogiePL[10] is an intermediate programming language used by the Spec# programming system. It is not used to compile and execute programs but to verify them. Before the actual verification, Boogie translates .NET programs and libraries into BoogiePL and applies a number of transformations. These transformations modify the program in such a way that Boogie can generate efficient verification conditions.

In the following sections we present a short introduction to BoogiePL to the extent necessary to understand this thesis.

Uninterpreted functions

An uninterpreted function is a function that has a name and a type but no actual function definition. Axioms define the constraints that the function has to satisfy. Boogie uses uninterpreted functions and axioms in the background theory to describe Spec# concepts such as the type system and the ownership model. The counterexample of a failed verification may contain function interpretations, that is, concrete results for uninterpreted functions with specific arguments.

Local variables

As the value of a local variable may change during execution, Boogie in general assigns multiple identifiers to a local variable. The different incarnations of the same variable can be distinguished by a sequence number that Boogie appends to the variable name.

Example identifiers for the local variable i
i@0, i@1, i@2

Parameters

The input values of parameters of the failing method are annotated with a $in suffix. If the parameter value gets changed during the execution of the method, the parameter variable is treated like any other local variable.

Example identifiers for the parameter foo
foo$in, foo@0, foo@1

Heaps

The heap is the area of memory where the state of dynamically allocated objects is stored. Boogie models the heap as one big array, that is, the fields of all objects are stored as an element in the heap array.

As the content of the heap may change during execution, multiple identifiers refer to the different incarnations of the heap.

Example identifiers for the heap
$Heap, $Heap@0, $Heap@1

Let us look at an example how values on the heap are stored as elements in the two-dimensional heap array.

```
$Heap[stack50000o, Foo.field] := 23
```

The first index denotes the object reference and the second index references a field in that object. $stack50000o$ is an intermediate stack variable. The above assignments stores 23 in $Foo.field$ in the object referenced by the specified stack variable.

Method calls

When verifying a method that contains method calls, Boogie does not take the code in the called methods into account for proving verification conditions. Boogie reduces the method call to an assertion of the preconditions and an assumption of the postconditions. This means that the caller may never assume anything else than what has been specified in the method contracts.

Return values of method calls that occur in the method body are represented as special identifiers in the counterexample model. The generated identifier contains a unique number identifying the specific method call.

Example identifier describing a method call result
call2848formal@$result@0

Pure methods, used for method calls in contracts, are represented as uninterpreted functions. As pure methods always return the same value for the same input, it is not necessary to distinguish between multiple method calls with the same arguments.

Example function describing the result of a pure method call
#TestClass.Test($Heap, obj$in)

Background theory

The background theory of Boogie describes a formal encoding of the type system, the ownership model, and the basic data types of the Spec# programming system. It is defined in a file called 'prelude.bpl' that contains numerous definitions of constants, uninterpreted functions, and axioms.

Data types

Values and objects of different types need to be encoded in various ways in the verification conditions, depending on the data type.

Integers

No special encoding is required for integer literals.

Booleans

Boogie uses two special identifiers to represent true and false, the boolean values.

Identifiers used for booleans

```
@true, @false
```

Strings

Boogie encodes the length of strings using uninterpreted functions, however, Boogie does not keep track of the actual string contents.

Uninterpreted function for the string length

```
function $StringLength (ref) returns (int);
```

Example function describing the length of a string

```
$StringLength($stringLiteral)
```

Arrays

Boogie encodes the rank, the length, and the contents of Spec# arrays using uninterpreted functions.

Uninterpreted functions for arrays

```
function $Rank (ref) returns (int);
function $Length (ref) returns (int);
function ValueArrayGet (elements, int) returns (any);
function IntArrayGet (elements, int) returns (int);
function RefArrayGet (elements, int) returns (ref);
```

Example values using array functions

```
$Rank(initialElements$in)
$Length(initialElements$in)
IntArrayGet($Heap[initialElements$in, $elements]), 0)
```

Objects

A variable containing an object reference is like any other variable except that you need a specific heap instance in addition to the object reference to retrieve a specific state of the object as described before.

Example describing the state of an object field

```
$Heap[this, Bag.count]
```

Structs

Values of struct fields are encoded using uninterpreted function, similar to object fields except that no heap instance is involved as structs have value-type semantics.

Struct function declaration

```
function $StructGet (struct, name) returns (any);
```

Example describing a struct field

```
$StructGet(bar, Struct.foo)
```

Null references

Boogie uses the special identifier *nullObject* to represent null references, that is, invalid object references.

Sample translation

Let us look at a simple example in Spec# and how that translates to BoogiePL:

Example Spec# Code

```
public class Test
{
    public int bar;

    public void UpdateField()
    {
        this.bar = 42;
    }
}
```

To keep the example short, we only include the translation of the above assignment statement in the following BoogiePL code snippet:

Translated to BoogiePL

```
1     // ----- load constant 42 ----- Test.ssc(7,3)
2     stack0i := 42;
3     // ----- store field ----- Test.ssc(7,3)
4     assert this != null;
5     assert $Heap[this, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[this,
          $ownerRef], $inv] <: $Heap[this, $ownerFrame]) || $Heap[$Heap[this, $ownerRef],
          $localinv] == $BaseClass($Heap[this, $ownerFrame]);
6     havoc temp0;
7     $Heap[this, $exposeVersion] := temp0;
8     $Heap[this, Test.bar] := stack0i;
9     assume IsHeap($Heap);
```

On line 2 we see the assignment of the integer literal 42 to the intermediate stack variable called *stack*0*i*. The check on line 4 ensures that the object reference is not null to avoid a potential null reference exception. Line 5 checks ownership and expose state to ensure that we are allowed to modify the field. Line 6 ensures that we cannot assume anything about the value of *temp*0, and line 7 sets the internal field that keeps track of the expose state to this unknown value. The actual assignment happens on line 8, that is, it sets the corresponding element in the heap array to *stack*0*i*, the intermediate stack variable.

2.4 Z3

Z3[9] is an automatic SMT[1] theorem prover. Boogie translates the BoogiePL program into a formula and tries to prove it using Z3. If it cannot prove the formula, it returns a counterexample

[1]Satisfiability Modulo Theories

model. That model represents the state of variables and objects from the beginning of the method to the point where the verification fails.

The counterexample model is split into partitions. Each partition represents a single value or object reference and may be associated with one or more identifiers. The identifiers represent specific incarnations of local variables, return values of method calls, or internal stack variables.

Example of model partitions

```
partitions:
  *0: True
  *1: False
  *2: @true
  *3: @false
  *60: 23  {foo$in}
  *64: 0  {return.value $result}
  *70: 42  {call2848formal@$result@0}
  *71: 0  {i@0}
  *73: 1  {i@1}
  *80: 123
  *100: nullObject
 [...]
```

For example, if we want to read out the value of the method parameter ‘foo’, we look up the model partition for the input value called ‘foo$in’. This is partition number 60 in the above example, which has an associated concrete value: 23. Therefore, the parameter ‘foo’ has the value 23.

In addition to model partitions, the counterexample also contains function interpretations of pure methods and internal functions. That is, it specifies the output of uninterpreted functions for certain input values that are relevant to the counterexample.

Example of function interpretations

```
1  function interpretations:
2  #TestClass.Test($Heap, obj$in) = 42
3
4  $typeof(this) = *77 (SELF)
5
6  $StringLength($stringLiteral) = 3
7
8  $Rank(initialElements$in) = 1
9  $Length(initialElements$in) = 1
10 IntArrayGet(select2($Heap, initialElements$in, $elements), 0) = 9
11
12 $StructGet(*63, Struct.foo) = 42
13
14 select2($Heap@0, stack50000o@0, Foo.field) = 123
15 select2($Heap, this, Bag.count) = 0
16 [...]
```

On lines 8-10 we can see that the method parameter ‘initialElements’ is a one-dimensional array of length 1. The first element of that array is the integer 9 in the initial heap state.

2.5 Verification failures

As program verification can fail at various locations, we describe the different types of verification failures in the following paragraphs.

Assertion failures

An assertion statement is the simplest form of a verification condition. It means that the preceding code has to ensure that the condition holds, and the following code can assume that the condition holds. You can insert it at any place in a code block. If Boogie cannot verify that the assertion always holds, it will return an assertion failure.

Example code resulting in an assertion failure

```
1  public class Assertion
2  {
3      public void Test()
4      {
5          int i = 20;
6          assert (i < 10);
7      }
8  }
```

The assertion statement on line 6 will fail in this example.

Precondition failures

A method precondition is a condition that must be true when entering the method. The calling method is responsible to ensure that the condition holds. The code in the called method assumes that the condition holds and should not be executed otherwise. If Boogie cannot verify that the precondition holds for a specific call, it will return a precondition failure on the call-site.

Example code resulting in a precondition failure

```
 1  public class Precondition
 2  {
 3      public void UseInteger (int i)
 4      requires i < 10;
 5      {
 6           [...]
 7      }
 8
 9      public void Test ()
10      {
11          UseInteger (20);
12      }
13  }
```

In the above example the precondition on line 4 will fail when calling the method on line 11.

Postcondition failures

A method postcondition is a condition that must be true when leaving the method. The called method is responsible to ensure that the condition holds. The code following the method call in the calling method may assume that the condition holds and should not be executed otherwise. If Boogie cannot verify that the postcondition holds at the end of the method, it will return a postcondition failure.

A specialty of postconditions is that a postcondition may refer to the old state of a variable or expression, that is, the value it had when entering the method.

Example code resulting in a postcondition failure

```
1  public class Postcondition
```

```
 2 {
 3     public int Test ()
 4     ensures result < 10;
 5     {
 6         return 20;
 7     }
 8 }
```

In this example the postcondition on line 4 will fail at the end of the method, that is, on line 7.

Object invariant failures

An object invariant is part of the class contract. The condition must hold for all instances of that class, essentially for the whole lifetime of the object. Constructors are responsible to establish the object invariants. An object invariant may be temporarily broken when the object has been exposed. If Boogie cannot verify that the object invariant always holds outside of expose statements, it will return an object invariant failure.

Example code resulting in an object invariant failure

```
 1 public class ObjectInvariant
 2 {
 3     public int Foo;
 4
 5     invariant Foo < 10;
 6
 7     public void Test ()
 8     {
 9         this.Foo = 20;
10     }
11 }
```

As the above example does not use an expose statement, the object invariant on line 5 must always hold. However, the assignment on line 9 breaks the invariant.

Loop invariant failures

A loop invariant is a condition that must be true before and after each loop iteration. The code preceding the loop - this includes loop initialization in a for-loop - has to ensure that the loop invariant holds before entering the loop. The code in the loop body may then assume that the condition holds at the beginning of the iteration, however, it also has to ensure that the condition holds at the end of the iteration.

We distinguish between the following two types of loop invariant failures.

Loop pre-state invariant failures

If Boogie cannot verify that the loop invariant holds when entering the loop, it will return a loop pre-state invariant failure.

Example code resulting in a loop pre-state invariant failure

```
 1 public class LoopPreState
 2 {
 3     public void Test()
 4     {
 5         for (int i = 0; i < 10; i++)
 6          invariant i > 0 && i <= 10;
 7          {
 8              [...]
 9          }
10      }
11  }
```

The loop invariant on line 6 does not hold when entering the loop, that is, right after initializing i on line 5.

Loop post-state invariant failures

If Boogie cannot verify that the loop invariant holds at the end of a loop iteration, it will return a loop post-state invariant failure. The iterator expressions in a for statement are considered part of the loop body, this means that the loop invariant needs to hold after evaluating the iterator expressions.

Example code resulting in a loop post-state invariant failure

```
 1  public class LoopPostState
 2  {
 3      public void Test()
 4      {
 5          for (int i = 0; i < 10; i++)
 6          invariant i >= 0 && i < 10;
 7          {
 8              [...]
 9          }
10      }
11  }
```

In this example, the loop invariant on line 6 fails after the 10th loop iteration, that is, right after incrementing i on line 5.

Frame condition failures

Frame conditions limit the set of fields a method may modify. By default, a method may only modify the fields of the 'this' instance. Frame conditions can be violated either by directly modifying a field that does not lie within the frame condition or by calling a method that modifies fields you are not allowed to modify.

Example code resulting in a frame condition failure

```
 1  public class FrameCondition
 2  {
 3      int foo;
 4      int bar;
 5
 6      public void Test()
 7      modifies this.foo;
 8      {
 9          this.bar = 42;
10      }
11  }
```

The assignment on line 9 breaks the frame condition on line 7 in the above example.

2.6 Debugger

As our goal is to execute counterexamples in the debugger, we have to understand what information the debugger needs to function. First and foremost, the debugger needs a normal executable assembly containing the code you want to debug. However, the assembly does not contain any information about the original source code. This is why debuggers use an additional file that contains debug symbol information. In the case of Spec# and Visual Studio, the corresponding file format is called PDB - Program Database. This file format is specific to the Visual Studio debugger. The internal structure is not public, however, there is a COM API to read and write PDB files.

This file contains information like the name of local variables, the name of the source file, and the line number for each statement. An interesting point to note here is that there are no restrictions on how to associate line numbers with CIL instructions. The line numbers do not have to be sequential, and the code on the source lines does not have to match the CIL instructions.

Using these two files, the assembly and the debug symbol file, the debugger can provide the well-known debugging possibilities such as stepping through programs and displaying the value of local variables.

Chapter 3

General approach

Before going into the technical details, we describe here the general approach we have taken in this thesis.

Let us assume we have a code part that fails verification. This code can be part of a library or an application that expects user input, that is, in general it is not a stand-alone test case. This means that we cannot assume that it can be executed without extra setup code.

Example code failing verification

```
public class Simple
{
    public int Test (int value)
    ensures result >= 10;
    {
        return value;
    }
}
```

Boogie reports the condition that failed verification. It also returns a counterexample that contains value partitions describing how to reproduce or comprehend that specific verification failure.

Error message for above example code

```
Error: Method Simple.Test(int value), unsatisfied postcondition: result >= 10
```

In general, the error message that Boogie returns when program verification fails does not contain enough information to understand the verification failure. The counterexample model contains the missing information, however, the text version of the model can be very long, even for small programs. The model also references certain values with internal identifiers without any indication to which part of the original code it relates to. This has the effect that it is often impossible to understand the verification failure.

Short excerpt from the corresponding counterexample model

```
partitions :
  *53: 0  {$result  value  return.value SS$Display.Return.Local}
  *64: 9  {value$in}
```

This tells us that the verification failed for the case where the method Simple.Test gets called with the integer 9 as argument.

This thesis improves the situation significantly by enabling developers to use the familiar debugger user interface to follow the counterexample and understand the reason why a program fails verification.

We execute the failing method while following the model of the counterexample and without requiring manual setup or test code. Therefore, we transform the original program code into a form so that it is executable without further input, only qexecutes the failing method, and uses the values from the counterexample to be able to reproduce the verification failure at runtime.

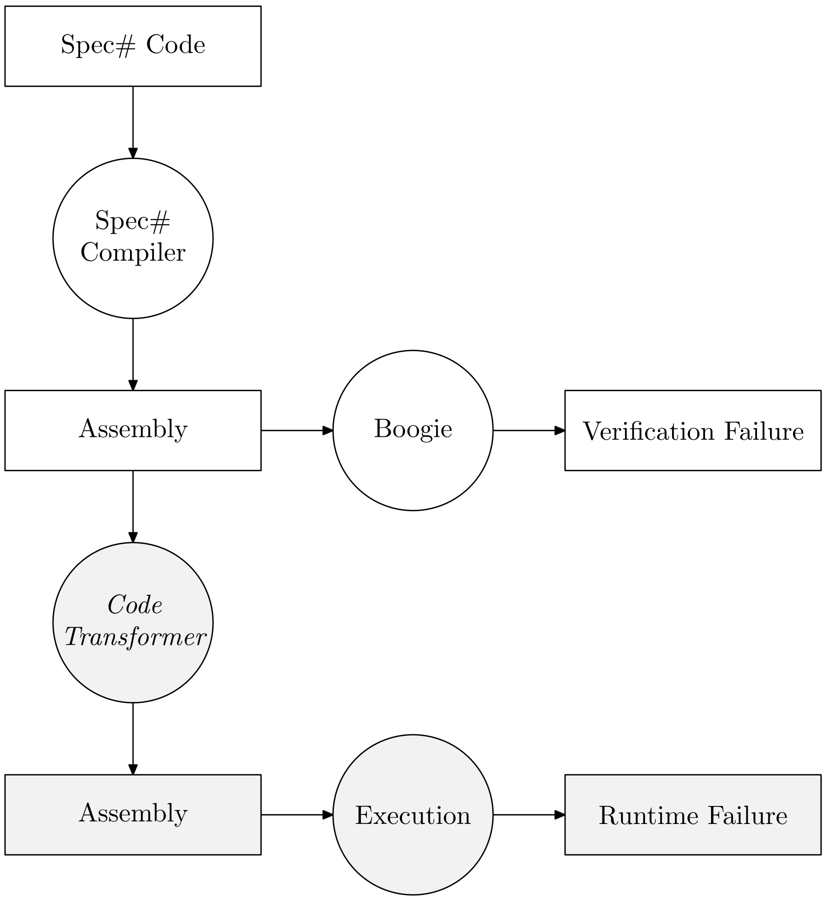

Figure 3.1: Components involved in code transformation

Figure 3.1 shows the various components involved in the compilation, verification, and transformation of the code and the following execution. The gray boxes represent the new components contributed by this thesis, that is, the code transformation and the execution in the debugger including runtime checks.

We rewrite the CIL[1] code in the assembly using mock objects to create a runtime state that corresponds to the counterexample. The code for method calls and loops is modified to retain the behavior specified in the counterexample when stepping through the program. The rewritten assembly is then executed in the debugger. Runtime checks ensure that the programmer is immediately aware of failing verification conditions. We have implemented all this in a tool called Cee C.

[1]Common Intermediate Language

Chapter 4

Mock objects

4.1 Introduction

As stated in in the previous chapter, we create a runtime state from the model of a counterexample. Mock objects enable us to achieve the desired runtime behavior.

A mock object is an object that has the same interface as a real object and behaves similar to a real object, however, it acts in a controlled way. Mock objects deliver deterministic results without, for example, interacting with external data sources. A method call on a mock object usually returns a predefined result after checking the validity of the arguments.

We use mock objects to create a runtime state that matches the counterexample as closely as possible. Using mock objects allows us to run the method step-by-step with a standard debugger user interface.

There are different possibilities to generate mock objects. Generating mock proxy classes via reflection is possible where interfaces or virtual methods are used. More flexible mock objects require rewriting CIL code, either on the fly or before execution. The following section provides an overview of the available frameworks and what we found to be their limitations in respect to our goals.

4.2 Mock frameworks

Rhino.Mocks

Rhino.Mocks[11] is an open source mock object framework to dynamically create mock implementations with a specific behavior. It uses the reflection API to create proxy classes at runtime. The library is easy to use and provides a rich API to specify the expected behavior.

Let us look at an example how to create mock objects using Rhino.Mocks:

```
// create mock object
MockRepository mocks = new MockRepository ();
ISample mockSample = mocks.CreateMock<ISample> ();

// specify desired behavior
Expect.Call (mockSample.Test ()).Return (true);
mocks.ReplayAll ();

// method will return true as specified above
bool result = mockSample.Test ();
```

We have implemented a proof of concept tool that can execute simple methods according to a counterexample using Rhino.Mocks to create mock objects.

However, the behavior of non-virtual methods cannot be changed without rewriting CIL code on the fly using the .NET Profiling API. Also, simple mock objects are not suitable for non-pure methods, as they might need to return something different on each call. These limitations render Rhino.Mocks unsuitable for our purposes.

.NET Profiling API

The Microsoft .NET Framework provides a profiling API that supports various possibilities for interception of code execution at runtime. Among other things, this makes it possible to rewrite CIL code on-the-fly before the JIT compiles it to native code.

However, the .NET Profiling API is only accessible from unmanaged C++ applications. It is not recommended to call into the virtual machine from C++ when using the .NET Profiling API to avoid side effects on the application being profiled.

We access the counterexample model using classes provided by Boogie in a managed assembly. We need to be able to access the counterexample model while rewriting CIL code, which is not possible as we cannot access a managed library when using the .NET Profiling API. This limitation makes the .NET Profiling API unsuitable for our purposes.

TypeMock

TypeMock[17] is a commercial mock framework that uses the .NET profiling API to support mocking non-virtual methods and sealed classes.

While it supports various possibilities to create mock objects and specify the behavior, it is not possible to just do specific changes to an existing method body. This makes TypeMock unsuitable for the more advanced parts of code transformation as, for example, handling loops or intercepting object construction.

4.3 Rewriting assemblies

As the current frameworks are insufficient for our project, we have decided to rewrite assemblies. Modifying CIL code by writing new assemblies is an alternative to creating mock objects at runtime. The advantage is that it allows modification of any aspect of the application, however, rewriting assemblies is considered more complex than using mock frameworks.

The idea of rewriting CIL code is to disassemble the compiled assembly, modify the code to match the behavior specified in the counterexample, and reassemble the code into a new executable assembly.

The possibilities of this approach are extensive, however, it requires a library that makes it possible to disassemble, modify, and reassemble CIL code while keeping the associated debug information.

Debug symbols

As we run the modified CIL code in the debugger and want to display the corresponding source code, we ensure that we have debug symbols that match the new code. In the case of the Visual Studio Debugger, this means that we also rewrite the PDB file. Otherwise, the debugger cannot associate the source code with the code being executed.

Cecil

Cecil[12] is an open source library, part of the Mono project, to disassemble, analyze, and generate programs and libraries in the ECMA CIL format. It also supports rewriting an assembly, that is, you can disassemble a program, manipulate the abstract syntax tree, and write back the modified program. It includes read and write support for debug symbol files, that is, it is possible to preserve debug information in the PDB file when rewriting an assembly.

We have performed some tests, which have shown that it is quite comfortable to work with Cecil to rewrite assemblies. It operates on a pretty low level, but the assembly nodes and the metadata are wrapped in nicely structured classes.

Common Compiler Infrastructure

CCI, the Common Compiler Infrastructure, is a framework for writing .NET compilers and other language tools. It allows reading, manipulating, and writing assemblies using both, common high-level language constructs and low-level CIL operations. CCI also supports reading and writing debug symbol information in PDB files. The Spec# compiler and the Boogie program verifier both use this framework to write and read assemblies, respectively.

As CCI supports reading and writing assemblies including debug information, it is a suitable candidate for our code transformation. An important advantage - compared to using Cecil - is that CCI is aware of contracts as the framework has been extended by the Spec# team. This allows us to access, for example, the contract deserializer, which is able to generate runtime expressions out of serialized contract information stored in an assembly.

The good integration with the existing Spec# tools was the reason why we have decided to use the CCI framework for our project.

Example

The following example demonstrates a bit more in detail how code rewriting works. We have a very simple method that takes an integer and returns an integer. The postcondition guarantees that the result is always greater than or equal to 10. The Spec# code looks as follows:

Example code in Spec#

```
public class Simple
{
    public int Test (int value)
    ensures result >= 10;
    {
        return value;
    }
}
```

Building and transforming this example using the Spec# compiler and Cee generates an assembly that allows execution in the debugger. To make the code more pleasant to read, we show the C# equivalent of the CIL code in the assembly:

C# version of the rewritten assembly

```
public class Simple
{
    public int Test (int value)
    {
        Debugger.Launch ();
        int result = value;
        if (result < 10)
        {
            throw new EnsuresException ("Postcondition 'result >= 10' violated from method '
                Simple.Test(System.Int32)'");
        }
        return result;
    }

14      static void Main ()
15      {
16          new Simple ().Test (9);
17          throw new ContractException ("Unable to reproduce verification failure");
18      }
19  }
```

Execution starts on line 16 where we create a new instance of the class *Simple* and call the *Test* method with 9 as argument. The method *Test* is called as that is the failing method, and the argument value has been determined using the counterexample. Right after entering the test method we call *Debugger.Launch* on line 5 to attach the debugger to the program, pause execution, and allow the user to inspect variables and step through the code. The return statement in Spec# has been transformed to an assignment to the *result* variable on line 6 as the runtime checks are performed before actually returning from the method. On line 7 we check whether the postcondition failed and throw an appropriate exception. This is the case in our example as we have called the method with 9 as argument.

If it was not possible to reproduce the verification failure at runtime, we would return the result on line 11, and the statement on line 17 would throw an exception explaining that we were unable to reproduce the verification failure.

4.4 Evaluating counterexamples

Now that we have a framework available that allows us to rewrite assemblies, we make sure to correctly read the counterexample model, so that we can rewrite the CIL code accordingly.

The model partitions of the counterexample are stored in an instance of the ErrorModel class. The Boogie API does not currently provide access to the ErrorModel object, given a Counterexample object. We have added an ErrorModel field to the Counterexample class with appropriate initialization code to make it possible for our tool to access the underlying model.

Heaps

When constructing the runtime state of an object from the counterexample, we have to make sure that we follow the appropriate heap when retrieving state information from the model.

For parameter values, we only care about the initial heap, as that is the heap at the beginning of the method. The other instances of the heap may be reached during execution of the counterexample. Mock methods may be called when the current heap is not the initial heap anymore. We distinguish between three cases when we call mock methods.

Results for method calls in code blocks that are related to the current counterexample are explicitly available in the model of the counterexample, we match these method calls using CIL offsets.

Method calls used in runtime checks of pre- or postconditions are required to be calls to pure methods, and corresponding call results are available as function interpretations in the model of the counterexample. For preconditions and postconditions we need to use the initial heap and the last heap, respectively.

The remaining method calls are in code blocks that are unreachable according to the counterexample. We make sure to not execute code blocks that should not be reached by terminating execution when we cannot reproduce the verification failure at runtime.

Constructing objects

To execute a method that takes object instances as arguments, we need a way to construct these objects. Constructing a new object in the CLR[1] involves calling a constructor of the corresponding

[1]Common Language Runtime

class. Some classes may not have a default constructor, that is, a constructor without parameters, which means that there is no clear way to create a new instance. Also, we do not execute existing constructor code at all, as we are only interested in creating mock objects.

Therefore, we create a new class deriving from the class that we want to instantiate. To this new class we add a default constructor that does not contain any code. Compilers of languages like C# require you to call the base constructor, however, the .NET Framework allows execution of constructors that do not chain up to the base constructor.

This scheme obviously does not work with sealed classes. If it is impossible to unseal the class, as, for example, it is in a different assembly, it is necessary to redirect all references to the class to a duplicate type with the same interface.

Abstract classes and interfaces

As Boogie can verify against abstract classes and interfaces, we extend them to be able to create mock objects. We add mock implementations for all abstract methods to the generated class. These mock implementations do not contain any useful code and only exist so that the rewritten code passes the assembly verification of the virtual machine. As we generate a separate mock method for each method call, the original abstract methods are never called and the mock implementations for abstract methods are never executed.

Duplicate types

It is possible to define two types with the exact same name and namespace in two different assemblies that are loaded in the same application. This makes it possible to duplicate types, for example, sealed classes of the core libraries, in the user assembly without using a different type name which could lead to confusion while debugging.

Null references

The counterexample model has a special partition called ‘nullObject’ that represents a null reference. This makes it possible to accurately represent null references, for example, to reproduce null reference exceptions.

Value types

As the various values and objects in the counterexample model can be of different types, creating a runtime state of values in the counterexample can differ significantly depending on the type. The following paragraphs describe how to handle each value type during code transformation.

Integers

The counterexample model contains concrete values for integers. In addition to the value we also need to know the exact CLR integer type to generate the corresponding runtime value. We infer the type information from the context, for example, we can use the formal parameter or return type of a method.

Booleans

There are two special partitions in the counterexample model that represent true and false, respectively. This allows a direct mapping to the CLR bool type.

Strings

Boogie reports the length of a string in the returned error model. However, there is no information about the actual content in the counterexample. We construct runtime version of strings that have the correct length but just consist of 'a' characters.

Arrays

The counterexample model contains the rank, the length, and also the contents of an array, as far as relevant. We use this information to create an appropriate mock instance of the array.

Structs

Structs do not have the construction issues that classes have, calling the struct initializer method is optional. We initialize struct fields to the values as specified in the counterexample to achieve the intended behavior.

4.5 Executing rewritten code

To make it possible to execute the failing method in the rewritten assembly, we generate an entry point method that calls the method after initializing the arguments. The values for the method call arguments are taken from the counterexample we receive from the program verifier.

End of execution

When we reach the point after the assertion failure reported by Boogie, we stop the execution and display an appropriate message. The counterexample does not specify execution beyond this point, and it also would not fulfill any purpose as it was apparently a spurious counterexample - however, not necessarily a spurious error.

The exact location of the point after the assertion failure depends on the type of the verification failure, we clarify this in the relevant sections.

Chapter 5

Method calls

5.1 Introduction

The normal model of execution differs from the model that Boogie uses to verify programs. Method calls are reduced to assertions and assumptions that conform to the method contract: preconditions, postconditions, and frame conditions. The actual code in the callee is not relevant to the verification of the calling method.

One possible way to execute method calls is to transform the counterexample back to normal execution. However, there is no direct mapping from Boogie's model back to the normal execution model. If we called methods as in normal execution, the return value and possible side-effects of the method call might not match the behavior specified in the counterexample. This means that we often could not reproduce the verification failure at runtime as we would not follow the counterexample. If we cannot reach the point of failure at runtime, stepping through the preceding statements is meaningless when trying to understand the verification failure.

As transformation back to normal execution is not suitable for our purposes, we have decided to follow Boogie's model and the counterexample as closely as possible. This means that, for example, we do not execute the body of the callee when encountering a method call; we only check the preconditions and return the value specified in the counterexample.

This may initially be confusing, however, it should not be hard to understand for developers that know how to specify code using Spec#. There is no value in trying to map the counterexample back to normal execution if the two paradigms do not match as our goal is to help developers specifying code, not to help people who do not know how to specify code.

If a failing method contains calls to other methods in its body, we replace these method calls by calls to mock methods to be able to follow the counterexample as returned by Boogie.

5.2 Return values

The model of the counterexample contains the return values for all method calls that are relevant for the counterexample. It does not contain return values for method calls in unreachable code parts, however, this is not of concern to us as we follow the counterexample.

For each method call we first generate a mock method that returns the value as specified in the model, and then we redirect the original method call to that mock method. We do not modify the originally called method as that method may be in a different assembly, or the method may be called multiple times and return different values.

Counterexample model

Boogie uses internal statement node identifiers to encode method call results in the counterexample model. As statement nodes are eliminated during verification, we have to store a mapping from the node identifiers to the original method calls before starting the actual verification process.

The expected results of method calls are provided in the counterexample model, in the form of an AssumeCmd in the trace, for example, call2848formal@$result@0. These AssumeCmd objects are created from CallCmd objects, which represent the actual method calls, however, it is impossible to access the corresponding CallCmd objects after verification. The number part of the identifier in the AssumeCmd, 2848 in this example, is the unique id of the corresponding CallCmd.

The System.Compiler project, part of CCI, can optionally be built with the compile-time option ILOFFSETS, which adds an ILOffset field to Compiler.Expression and Compiler.Statement nodes to associate code nodes with a byte offset in the CIL code. We have also extended the Boogie.CallCmd class with an ILOffset field and set it to the same value as the ILOffset field of the corresponding Compiler.MethodCall node. This makes it possible to associate disassembled method calls with CallCmd objects.

Before starting the actual verification, Cee iterates over all CallCmd objects and stores the mapping from CIL offset to the unique id of the corresponding CallCmd in a hash table. It needs to be executed before Boogie.Program.Typecheck() as the desugaring of Typecheck() already removes the necessary CallCmd nodes. Using the hash table we can now generate the model identifier for the return value from the CIL offset of a method call.

5.3 Method preconditions

When we execute a counterexample for a method that calls another method, we do not want to execute the actual body of the other method. We only call a mock implementation of that method, which returns the value specified in the counterexample as described in the previous section.

However, in addition to returning the appropriate value, we also want to execute runtime checks for the method preconditions because we want to be able to reproduce precondition failures at runtime, that is, we want to see the thrown RequiresException in the debugger in case that the precondition has been violated.

Let us look at a simple example of a method precondition:

Example for a precondition violation

```
1 public abstract class PreconditionTest
2 {
3     public abstract void UseInteger (int i);
4     requires i < 10;
5
6     public void TestObjectFunction (int i) {
7         UseInteger (i);
8     }
9 }
```

When executing the above example, we expect an exception to be thrown when calling the method on line 7 if the precondition on line 4 does not hold.

When debugging a verification failure, there is a corresponding runtime check that should fail according to the counterexample. If we pass that runtime check, we stop execution by throwing an exception. This means that we have found a spurious counterexample as we are unable to reproduce the verification failure at runtime. We discuss handling of spurious errors in a later section.

Handling of out-of-band contracts

The Spec# compiler also inserts runtime checks for preconditions at the beginning of the method body, however, this obviously only works if you write your code in Spec#. This is not the case for existing libraries such as the libraries part of the .NET Framework. Spec# supports out-of-band contracts, contracts specified in a separate file; this also works for libraries written in, for example, C#. However, as the manual runtime checks in existing libraries do not always cover all contracts, existing libraries may miss some runtime checks. Let us look at the following example:

Example code

```
int[] a = new int [0];
int[] b = new int [0];
a.CopyTo (b, 0);
```

The out-of-band contracts of Array.CopyTo contain the following precondition:

```
public void CopyTo (Array! array, int index)
    requires index < array.Length;
```

In the method call in our example code *index* and *array.Length* are both 0, and as 0 is not less than 0, the call violates the precondition. However, if we just compile and run it, we do not get any exception at runtime as mscorlib is actually less restrictive than the out-of-band contracts. If we debug the code using Cee we get the runtime exception as the runtime checks we insert into the mock methods strictly follow the contracts, they are independent of actual checks performed in the library. Ideally, the VM would be aware of the preconditions and ensure that they are always executed.

5.4 Object creation expressions

We treat object creation expressions similar to method calls, that is, if the failing method constructs new objects, we generate mock constructors and redirect the instructions to use them. The generated mock constructors are added to a subclass of the original class as it is not always possible to modify the original class. This follows the same concept as when creating mock objects as described in the previous chapter.

The counterexample may define field values for the constructed object. Cee uses that information to insert field assignments into the mock constructor. That way, the object will be able to satisfy the object invariants, and the programmer can inspect the fields in the debugger.

Chapter 6

Loops and recursion

6.1 Introduction

The normal execution of loops differs from the way how Boogie analyzes and verifies them. Before actually verifying a program, Boogie applies a number of transformations on the program code. Loops are transformed into a single iteration and a statement that abstracts the remaining iterations. Boogie does not attempt to prove that the loop terminates; it only proves that the loop invariants hold if the loop terminates, that is, it only verifies partial correctness.

Instead of analyzing multiple iterations of a loop, Boogie analyzes a single, abstract iteration of the loop. It first asserts the loop invariants in the loop pre-state. Boogie then havocs the heap and local variables, that is, it assumes that the heap and local variables have been modified arbitrarily - within limits inferred from the loop body. The havoc represents an arbitrary number of loop iterations. Then Boogie executes the loop body once and asserts the loop invariants again. If they hold, they will hold for any loop iteration. Figure 6.1 illustrates the difference between normal execution and Boogie's model.

Our goal is to execute the loop in a way that conforms to the counterexample and still remains usable when executing the code with a debugger.

Mapping the counterexample back to normal execution

One possible way to execute loops is to transform the counterexample back to normal execution. However, there is no direct mapping from Boogie's model to the normal execution model. This results in some issues when we try to execute the loop. The counterexample contains the incarnation of all variables before and after the havoc. However, there is no information about results of method calls that would happen in normal execution during the havoc. If the loop body contains method calls, the model of the counterexample will only contain the result of one such method call, even though the loop might run multiple times when executed normally. That means that we either cannot run the loop more than once, or we have to find a way to retrieve meaningful results for all calls.

If we executed loops as in normal execution, we could end up with infinite loops and side-effects that do not match the behavior specified in the counterexample. This means that we often could not reproduce the verification failure at runtime as we would not follow the counterexample. However, if we cannot reach the point of failure at runtime, stepping through the preceding statements is meaningless when trying to understand the verification failure.

Loop unrolling

A possible alternative is to get the counterexample closer to the actual execution by using loop unrolling. Boogie provides integrated support for unrolling loops by specifying a maximum number of iterations. When loop unrolling is enabled using the /loopUnroll:n commandline option, the

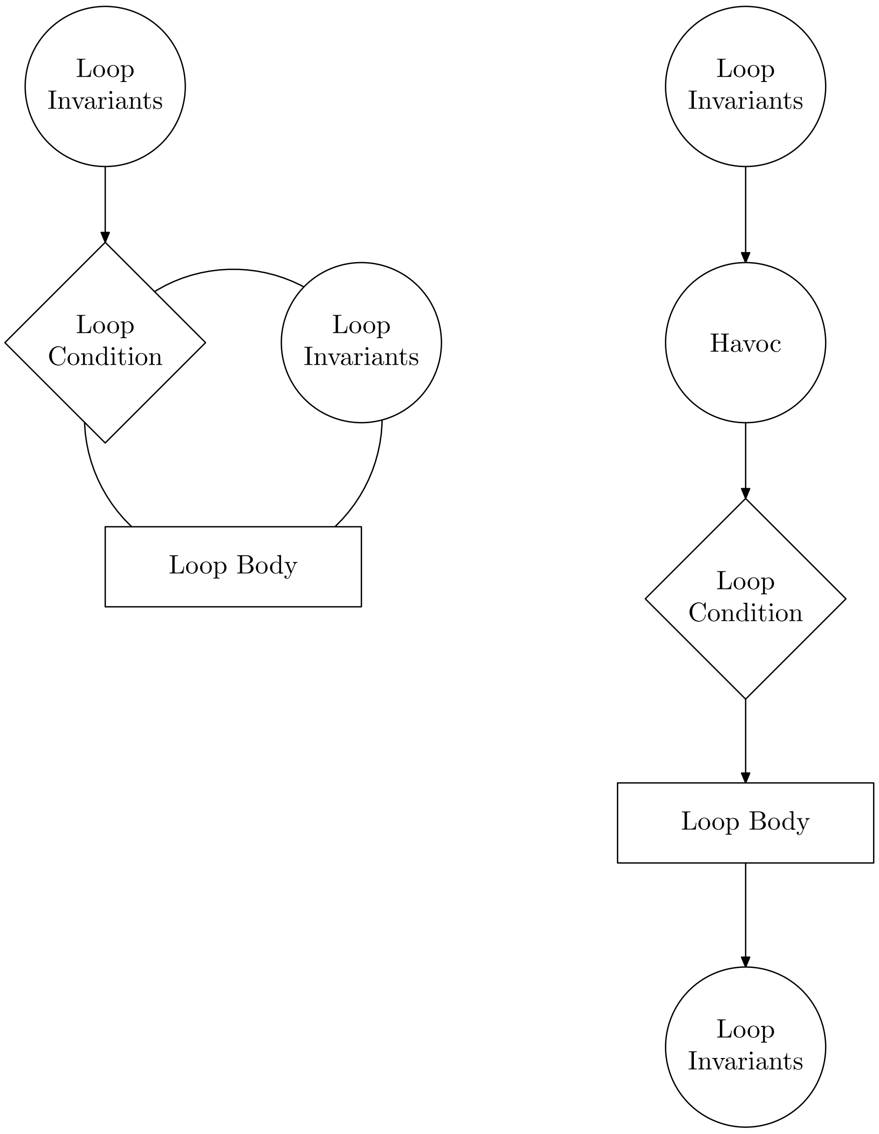

Figure 6.1: Normal execution versus Boogie's model

counterexample model contains concrete values for each loop iteration. That is, it adds separate method call results for each loop iteration to the model of the counterexample in the form of call1234formal@$result@i identifiers where i denotes the iteration number. With this additional information it is possible to execute multiple iterations of a loop with method calls to mock methods that return appropriate values for each iteration.

However, there are some issues with this approach. Loop unrolling requires specifying a maximum number of iterations. The number of necessary loop iterations is usually unknown and in general impossible or at least not practical to find out. The unrolling is always applied to all loops in the program, there is no way to unroll only a specific loop. Furthermore, it assumes that the loop terminates after the specified number of iterations, which means that Boogie will return invalid verification results if a loop needs more iterations than specified.

These issues render loop unrolling unsuitable for our purposes of executing counterexamples.

Following Boogie's model

As executing all loop iterations with or without loop unrolling does not work as desired, we have decided to closely follow Boogie's model for counterexample execution and step over the havoc in the debugger. This means that we never execute more than one iteration of any loop. This may initially be confusing, however, it should still allow the programmer to understand the verification failure.

In the case that we are debugging the counterexample for a failed loop post-state invariant, we start the debugging process right after the havoc, that is, we only step through the failing loop iteration. The programmer can inspect the state of local variables and objects right after starting the debugging. The state corresponds to the post-havoc values specified in the counterexample, and the loop invariants hold at this point. If the variables do seem to be incorrect, the loop invariants may not be strong enough.

6.2 Loop transformation

In this section we explain in detail how we transform loops in Cee in such a way that we can follow the counterexample at runtime.

Let us start with a code snippet that shows the structure of the compiled version of a simple while-loop, that is, a loop with a condition, a body, and loop invariants:

Original loop after compilation

```
start:
    assert loop invariants
    if (loop condition) {
        loop body
        goto start
    }
```

If we apply the loop transformation, which Cee performs on the CIL code when rewriting the assembly, to the pseudo-code, we obtain something like the following:

Loop after transformation

```
    assert loop invariants
    update local variables
    if (loop condition) {
        loop body
        assert loop invariants
    }
```

The update of the local variables and corresponding heap changes are performed by realizing an appropriate runtime state from the post-havoc incarnation map as provided in the counterexample.

An interesting point to note here is that the transformation is completely independent of the verification failure being debugged. This means that this transformation does not break the soundness of the runtime checks, it is just a necessary step to follow Boogie's model. Therefore, it is not only possible to use the same transformation to debug both, pre-state and post-state loop invariant failures, you can also debug verification failures that are not relevant to the loop at all using the same transformation.

Implementation

The Spec# compiler surrounds generated runtime checks with a special try/catch block with a ContractMarkerException to allow verification tools to distinguish between normal method code and runtime checks. Boogie ignores the runtime checks as they should not influence the verification.

We remove the runtime checks when rewriting the assembly to have full control over what runtime checks we generate in what locations. This also drops the runtime checks for loop invariants, of course. Regenerating and inserting them at the right location is not as straight forward as with method pre- and postconditions, though.

It is possible to detect runtime checks for loop invariants by looking for calls to the AssertionHelpers.LoopInvariant method and just not drop these runtime checks. However, this might make it impossible to mock method calls within loop invariants as we only have access to the normalized form of the conditions.

The Spec# compiler also serializes the loop invariants into strings and uses dummy calls to the AssertionHelpers.LoopInvariant method to store the serialized form of the conditions, similar to storing the pre- and postconditions in the Requires and Ensures method attributes. Using the contract deserializer makes it possible to regenerate runtime checks from these strings, however, we need to correct the associations of block variables after deserialization.

The main issue we have with matching method calls within loop invariants - whether we have the conditions in low-level or high-level form - is that it is not obvious which incarnation of the heap is the current heap when checking the invariants. However, the pre- and post-havoc incarnation maps also contain an entry for the current heap, which solves this issue.

Representation in Boogie

The following two examples show the object structure Boogie uses to store the information that we use for the loop transformation.

Loop start block in Boogie

```
Microsoft.Boogie.Block
    Label = "block1751$LoopPreheader"
    Statements = [
        LoopInitAssertCmd
            ErrorData = ErrorPair
                offendingNode = ExpressionStatement
                    Expression = MethodCall
                        ILOffset = 44
    ]
```

This shows the structure of the loop start block, the block that contains the loop pre-state invariant checks. The LoopInitAssertCmd contains the CIL offset - 44 bytes in this example -, which we can use to associate the serialized loop invariant in the assembly with loop information as returned by Boogie in the counterexample.

Starting from the loop start block, we can identify the corresponding loop end block in Boogie as shown below:

Loop end block in Boogie

```
Microsoft.Boogie.Block
    Label = "block1819_@2_block1751"
    Statements = [
        PredicateCmd
            Expr = LoopPredicate
                BlockName = "block1751"
                loopPredicateName = LoopPredicateName
                    PreHavocIncarnationMap # used for initial loop invariants
                    PostHavocIncarnationMap # used for simulating havoc command
        LoopInvMaintainedAssertCmd
            ErrorData = ErrorPair
                offendingNode = ExpressionStatement
```

```
                    Expression = MethodCall
                        ILOffset = 44
            IncarnationMap # used for after−state loop invariants
    ]
```

The loop end block helps us to retrieve the incarnation maps, that is, the model partitions that the heap and the variables have at the loop pre-state (PreHavocIncarnationMap), after the havoc (PostHavocIncarnationMap), and at the loop post-state.

6.3 Recursive methods

The normal execution of a recursive method is significantly different from the model Boogie uses for verification, similar to the difference in handling loops. While a recursive method is normally executed multiple times due to the recursive call, Boogie treats a recursive method call the same way as any other method call. This means that Boogie only takes the contract of the called method into account for verification; it transforms the method call into an assertion and an assumption statement according to the pre- and postcondition, respectively.

Normal execution

If we want to execute the counterexample of a recursive method using the normal execution model, we must not redirect the recursive method call to a mock method as usual but call the original method. This makes it possible to step into a recursive method call, however, it will not completely follow the counterexample anymore, and it is not always possible. If the recursive method contains loops or method calls to other methods, the counterexample does not contain enough information to execute the method recursively.

Execution according to Boogie's model

As our goal is to closely follow the counterexample to enable developers to understand the verification failures, we execute recursive methods according to Boogie's model. That is, we treat recursive method calls like any other method call and redirect them to mock methods. This behavior is in line with the transformation we follow with loops.

The McCarthy 91 function is a recursive function that always returns 91 as long as the input is less than 102. The following code snippet cannot be verified by Boogie. Using Cee you will recognize that the method misses a postcondition for the case that n is greater than or equal to 102.

```
public class F91Test {
    public static int F91(int n)
    ensures n < 102 ==> result == 91;
    {
        if (n > 100)
            return (n − 10);
        else
            return F91(F91(n + 11));
    }
}
```

Boogie reduces method calls to assertion and assume statements as we have seen in the previous chapter. Therefore, a recursive method is not considered recursive during verification.

If we apply this to the above example, we have to transform two method calls. The inner call is evaluated first and will result in the following code:

```
stack0i := n + 11;
havoc stack1i;
assume stack0i < 102 ==> stack1i == 91;
```

The second method call gets transformed to the following instructions:

```
havoc $result;
assume stack1i < 102 ==> $result == 91;
```

The postcondition check follows the two method calls:

```
assert n < 102 ==> $result == 91;
```

The counterexample states that n is 100. Let us see how this counterexample could be executed by following the transformed instructions. $stack01$ will be set to 111. As $stack0i < 102$ is false, we do not gain any information about the result of the inner method call in the following assumption. The counterexample assigns 102 to the return value, which is stored in $stack1i$. The assumption of the second method call does not provide any information, either, as $stack1i < 102$ is also false. The counterexample assigns 90 to the return value, which is stored in $\$result$. $n < 102$ is true, which means that the assertion does not hold for our $\$result$, and the corresponding runtime check will fail.

Chapter 7

Runtime checks

7.1 Introduction

The Spec# compiler generates runtime checks for method pre- and postconditions, loop invariants, and object invariants. We also want these checks in the rewritten assembly, however, we need to modify and sometimes also extend these checks. When rewriting the assembly, Cee first reads the assembly code that has previously been compiled by the Spec# compiler. The abstract syntax tree that we obtain that way does not contain all the information that the compiler had when writing the assembly, for example, information about old expressions is lost. For that reason, we remove all the previously generated runtime checks and then regenerate them with the necessary modifications.

The runtime checks are enclosed by a dummy try-catch block using the ContractMarkerException. This makes it possible to reliably remove them, just like Boogie's bytecode translator removes the runtime checks before verifying the program.

7.2 Execution order

If we want to debug a verification failure by executing the method, we have to make sure that the code does not fail for other reasons before the part that we want to debug. This means that we can always only debug the first verification failure in execution order, it is not possible to choose a specific failure to debug.

This requires careful sorting of the counterexamples that Boogie returns. We do this by ordering the counterexamples by the location of the failing assert node in the generated BoogiePL program and choosing the first one.

7.3 Pure methods

Contracts may also contain method calls, however, the called methods must be pure, that is, they may not have side-effects. While the model contains return values for each individual method call that occurs in the method body, there are no corresponding model entries for method calls within contracts.

To determine the return value of methods called from preconditions, postconditions, and invariants, we need to use the function interpretations from the counterexample model.

Example of a function interpretation in the counterexample model

```
#TestClass.Test($Heap, obj$in) = 42
```

As there may be multiple function interpretations for the same method, we need to match the arguments in the code with the arguments in the function interpretations to determine the right interpretation and therefore also the right return value.

Arguments in the code are in the form of subexpressions such as literals, method calls, or operations. Arguments in the function interpretations are partitions in the counterexample model. In the case of function interpretations, these partitions map to constants, fields, old values of fields, input parameters, or the return value. This means that we need to statically evaluate the subexpressions, so that we can compare them to the arguments in the function interpretations. These subexpressions may contain arbitrary Spec# expressions, which we need to handle before running the application.

Old expressions

Spec# allows you to refer to old versions of expressions in postconditions. Mapped to the model, this means, that we use the initial heap for evaluating method calls instead of the last heap. When we look at old expressions in the CIL code, we only see references to local helper variables that store the value of old expressions. However, we need the original expression for method call matching in the postconditions.

The Spec# compiler serializes the original expression, stores that into an EnsuresAttribute, and attaches the attribute to the method. We deserialize the postcondition from this attribute and regenerate the runtime checks for the pre- and postconditions. This makes it possible to redirect pure method calls to mock methods and therefore support old expressions.

7.4 Field assignments

Object invariants

Field assignments may violate object invariants, which means that we have to check object invariants after assignments to be able to reproduce object invariant violations at runtime.

However, the check is only necessary when the object is not exposed as invariants may be broken while an object is exposed. The Spec# compiler already inserts invariant checks at the end of the expose statement, and we use the same checks in Cee. These runtime checks use a generated helper method called ‘SpecSharp::CheckInvariant’.

Spec# also allows field modifications outside of expose statements. In that case, object invariants have to hold after each modification. This can make the checks very expensive as a single method may modify more than just one field.

The Spec# compiler currently does not insert runtime checks for object invariants after an assignment to a field of a non-exposed object for performance reasons. For example, the following program will not have any runtime checks, even though the object invariant may be violated.

Example code for field assignment without expose statement

```
class Invariant
{
    int n;
    invariant n < 10;

    public void Test()
    {
        this.n += 1;
        this.n += 2;
        this.n += 4;
    }
}
```

This means that we cannot be sure that we always have a runtime exception corresponding to a static verification failure.

To be able to check the object invariants outside expose statements, we have to know whether the we only have to check the invariants in the current class or also the invariants in subclasses. Object invariants can normally only refer to fields in the same class, that is, not to inherited fields. However, invariants may access so-called additive fields from a base class, that is, fields that have been annotated with the [Additive] attribute.

Additive fields can only be modified within an additive expose block and never outside of an expose block. This means that we never need to insert additional invariant checks after additive field updates as they are already covered by the checks at the end of the expose block.

However, fields that are not marked as additive cannot be referenced in invariants of subclasses. This means that we only have to check the invariants of the declaring class when assigning to a non-additive field.

We insert calls to the 'SpecSharp::CheckInvariant' method after field assignments to check the object invariants at runtime. We only run the invariant checks if the object is not currently exposed to allow invariant violations in expose blocks.

Frame condition failures

Frame conditions can be violated by method calls or by field assignments. In the case of a method call, a frame condition is a special precondition and in the case of a field assignment, it is a special assertion statement.

The Spec# compiler currently does not insert runtime checks to guard frame conditions for performance reasons as this would require keeping track of all field modifications. We have not implemented support for checking frame conditions in Cee.

7.5 Handling of spurious errors

When Boogie returns a verification failure, this normally means that there is an error in the program code or the contracts. However, this is not always the case. The verifier has limitations, it cannot prove everything even if it is correct.

When we execute a counterexample and we reach the point where a verification condition should fail but it does not, we have found a limitation, either in Cee or in Boogie. There is not much we can do in this situation as we do not know whether it is a spurious error, a spurious counterexample - that is, a real error with an incorrect counterexample -, or just a limitation of Cee. Therefore, we terminate execution by throwing an exception describing that we cannot reproduce the verification failure.

The following code is an example for a simple method that is correct but fails verification by Boogie due to limitations of the verifier or the background theory. Executing the code with Cee shows that the verification failure cannot be reproduced at runtime.

Spec# code

```
public class Test
{
    public uint TestMultiplication(uint u)
    ensures result >= u;
    {
        return 2*u;
    }
}
```

Verification using Boogie results in the following verification failure

```
Error: Method Test.TestMultiplication(uint u), unsatisfied postcondition: result >= u
```

Transforming the example using Cee generates an assembly that allows execution in the debugger. To make the code more pleasant to read, we show the C# equivalent of the CIL code in the assembly:

Transformed code as C#

```
 1 public class Test
 2 {
 3     static void Main()
 4     {
 5         new Test().TestMultiplication(50002);
 6         throw new ContractException("Unable to reproduce verification failure");
 7     }
 8
 9     public uint TestMultiplication(uint u)
10     {
11         Debugger.Launch();
12         uint result = 2 * u;
13         if (result < u)
14         {
15             throw new EnsuresException("Postcondition 'result >= u' violated from method '
                   Test.TestMultiplication(System.UInt32)'");
16         }
17         return result;
18     }
19 }
```

Running this example does not fail the postcondition as checked on line 13. The method 'TestMultiplication' will therefore return the result, and the main method will throw an exception describing that we cannot reproduce the verification failure as described further up in this section.

Chapter 8

Conclusions

We conclude this thesis by describing the limitations of our tool, related and further work, and our impressions using Cee.

8.1 Limitations

Our goal was to make it possible to construct a runtime state that reflects the counterexample as far as necessary to be able to reproduce the failed verification condition. We have shown that our tool works successfully for many test cases, however, there are still some limitations as described in the following.

Incorrect runtime state

We have encountered two situations where our code rewriting fails to produce a runtime state that matches the counterexample:

Comprehensions Boogie encodes comprehensions as a set of axioms using induction [13]. To avoid infinite loops in the Z3 theorem prover, Boogie uses triggers in the axioms. This has a side effect that the returned counterexample does not contain correct values when using comprehensions, and thus we cannot accurately reproduce the state at runtime. We have informed[6] the Z3 developers of this problem, and they are investigating the problem.

Arguments of method calls in contracts Returning the correct value for pure method calls in contracts requires static evaluation of all arguments as described in the previous chapter. Cee supports some common expression types, but the support is by no means considered complete. For example, we do not currently handle conditional expressions.

Missing runtime checks

We generate the same set of runtime checks as the Spec# compiler except that we have an additional check for field assignments outside of expose statements. Therefore, we also have the same limitations, that is, there are some verification conditions that are not checked at runtime and thus the developer will not be able to see an appropriate exception when debugging the counterexample.

An example is that we do not have any runtime checks for frame conditions. Implementing these should be possible but would presumably slow down program execution. While this may be an issue for normal execution of the program, we do not expect this to be problematic in a tool like Cee where we only execute a single method.

8.2 Related work

Check 'n' Crash

Check 'n' Crash[7] is a related project that uses static checking and automatic randomized testing to find possible runtime exceptions in Java, as for example invalid casts, null dereferences, and divisions by zero. Our goal was not to find errors but to make it possible to reproduce verification failures at runtime. To achieve that we use a counterexample returned by the verifier instead of randomized testing. Furthermore, we do not only analyze a fixed set of error types, we support a wide range of programmer written verification conditions such as method pre- and postconditions.

Pex

Pex[15] is a test generation tool of Microsoft Research that uses symbolic execution and parameterized unit tests to generate unit tests. It generates stand-alone executable unit tests from handwritten parameterized unit tests. Pex uses Z3 to search the execution paths to achieve maximal code coverage while keeping the amount of unit tests small.

It uses the .NET profiling API to rewrite CIL instructions on-the-fly in order to create mock objects with behavior[16]. While it uses a similar technique to recreate a static state at runtime, it has different goals. It simplifies writing test suites, it does not attempt to improve the handling of verification failures.

8.3 Possible extensions

Spurious errors

Integrating Cee into the Spec# compiler might help limiting the effect of spurious errors. Running the counterexample of each verification failure would give an indication whether it is a spurious error or not. If the failure can be reproduced at runtime, we know that it is not just a verifier limitation and can report it as a fatal error to the user, instead of warnings as reported by Spec#.

Feedback loop

If the execution of the counterexample does not fail, we have found a spurious counterexample, however, this is not necessarily a spurious error. It might be possible to rerun the verification with this additional piece of information to either get a different counterexample or a successful verification. A possible approach is to insert additional assume statements into the BoogiePL code stating that the variable does have a different value than what was in the spurious counterexample.

If we repeat this procedure multiple times, we reach one of three possible situations. One possibility is that we find a counterexample that is reproducible at runtime, that is, a real error. Another possibility is that the verification succeeds, which means that it is a spurious error, and we can avoid reporting it to the programmer. The third possibility is that all returned counterexamples do not fail at runtime, in this case we cannot decide whether the error is spurious.

Keeping the counterexample

When Boogie returns a verification failure and we are able to reproduce the counterexample at runtime, it might be interesting to keep the counterexample around in the form of a test case. A possible approach is to generate source code that corresponds to the counterexample. However, the test case might require many mock methods that need to be included in the generated source code. This test case could be a useful addition to a test suite to complement verification in the future.

Loop evolution

In some cases it might be useful to be able to debug more than one iteration of a loop. This will require a modification of the way Boogie generates verification conditions to encode loops. It will also be necessary to think of possibilities how the programmer can influence the number of iterations to be inspected using the debugger as it is certainly not always suitable to step through all loop iterations.

Visual Studio integration

A tighter integration into the Visual Studio debugger could improve the user experience significantly. For example, it might be useful if the exception dialogs for runtime checks contained the actual values of all relevant variables and method calls. It should also be made really simple to start the counterexample execution by clicking on the verification failure in the error list.

8.4 Impressions

While working on this thesis, we have tested our tool with various examples and often compared the information we found in the counterexample model with the information we were able to see in the user interface of the debugger. After some time we have found a work flow that is comfortable and efficient to use:

After launching the debugger we first take a look at the parameter values to see whether they should be allowed at all. If they should not, we strengthen the precondition and try the verification again. When the parameter values are ok, we step through the program code and check for all variables that change whether the new value is appropriate for the algorithm. If we find an issue, we check both, code and contracts, and correct errors as needed.

Using this process, we have been able to efficiently find errors in code and specification, certainly more efficient than when looking at the long counterexample model as returned by Boogie. While we have been skeptical about the practical use when starting the project, we are happy with what we have achieved in this thesis. Our approach has shown to be useful in our experience.

Acknowledgements I would like to thank my supervisors Joseph N. Ruskiewicz and Prof. Peter Müller for their support and a pleasant collaboration.

Appendix A

Examples

We show a couple of code examples to demonstrate the possibilities of counterexample execution using Cee. These do not represent our complete test suite, we have many more test cases that we have used for functionality and regression testing.

A.1 Precondition failure

The following example demonstrates a precondition failure:

Spec# code

```
public abstract class PreconditionTest
{
    public abstract void UseInteger(int i);
    requires i < 10;

    public void TestObjectFunction(int i)
    {
        UseInteger(i);
    }
}
```

Verification failure reported by Boogie

```
Error: Call of PreconditionTest.UseInteger(int i), unsatisfied precondition: i < 10
```

Transformed code as C#

```
public abstract class PreconditionTest
{
    public abstract void UseInteger(int i);

    public void TestObjectFunction(int i)
    {
        Debugger.Launch();
        UseIntegerCee1416(this, i);
    }

    static void UseIntegerCee1416(PreconditionTest self, int i)
    {
        if (self == null)
14          {
15              throw new NullReferenceException();
16          }
17          if (i < 10)
18          {
19              throw new ContractException(”Unable to reproduce verification failure”);
20          }
21          throw new RequiresException(”Precondition ’i < 10’ violated from method ’
                PreconditionTest.UseInteger(System.Int32 i)’”);
22      }
23
24      static void Main ()
25      {
26          new PreconditionTestCee().TestObjectFunction(10);
27          throw new ContractException(”Unable to reproduce verification failure”);
28      }
29  }
30
31  public class PreconditionTestCee : PreconditionTest
32  {
33      public override void UseInteger(int i)
34      {
35      }
36  }
```

Running the transformed code will start debugging at line 7 and throw a RequiresException at line 21.

A.2 Loop pre-state invariant failure

Spec# code

```
public class Foo
{
    public int Test()
    {
        int j = 0;
        for (int i = 0; i < 10; i++)
            invariant i > 0 && i < 10;
        {
        }
        return j;
    }
}
```

Verification failure reported by Boogie

```
Error: Loop invariant might not hold: i > 0
```

Transformed code as C#

```
public class Foo
{
    static void Main()
```

```
 4     {
 5         new Foo().Test();
 6         throw new ContractException("Unable to reproduce verification failure");
 7     }
 8
 9     public int Test()
10     {
11         int result;
12         Debugger.Launch();
13         int j = 0;
14         int i = 0;
15         if (i <= 0)
16         {
17             throw new LoopInvariantException("Loop invariant 'i > 0' violated from method '
                   Foo.Test'");
18         }
19         if (i >= 10)
20         {
21             throw new LoopInvariantException("Loop invariant 'i < 10' violated from method '
                   Foo.Test'");
22         }
23         j = 0;
24         throw new ContractException("Unable to reproduce verification failure");
25         if (i >= 10)
26         {
27             goto Label_0089;
28         }
29         i = i + 1;
30         if (i <= 0)
31         {
32             throw new LoopInvariantException("Loop invariant 'i > 0' violated from method '
                   Foo.Test'");
33         }
34         if (i >= 10)
35         {
36             throw new LoopInvariantException("Loop invariant 'i < 10' violated from method '
                   Foo.Test'");
37         }
38     Label_0089:
39         result = j;
40         return result;
41     }
42 }
```

Running the transformed code will start debugging at line 12 and throw a LoopInvariantException at line 17.

A.3 Loop post-state invariant failure

Spec# code

```
public class Foo
{
   public int Test()
   {
       int j = 0;
       for (int i = 0; i < 10; i++)
           invariant i == j;
           invariant j < 10;
       {
          j++;
       }
       return j;
   }
}
```

Verification failure reported by Boogie

```
Error: Loop invariant might not hold: j < 10
```

Transformed code as C#

```
public class Foo
{
    static void Main()
    {
        new Foo().Test();
        throw new ContractException("Unable to reproduce verification failure");
    }

    public int Test()
    {
        int j = 0;
        int i = 0;
        if (i != j)
        {
            throw new LoopInvariantException("Loop invariant 'i == j' violated from method '
                Foo.Test'");
        }
        if (j >= 10)
        {
            throw new LoopInvariantException("Loop invariant 'j < 10' violated from method '
                Foo.Test'");
        }
        j = 9;
        i = 9;
        Debugger.Launch();
        if (i >= 10)
        {
            int result = j;
            return result;
        }
        j = j + 1;
        i = i + 1;
        if (i != j)
        {
            throw new LoopInvariantException("Loop invariant 'i == j' violated from method '
                Foo.Test'");
34          }
35          if (j < 10)
36          {
37              throw new ContractException("Unable to reproduce verification failure");
38          }
39          throw new LoopInvariantException("Loop invariant 'j < 10' violated from method 'Foo.
                Test'");
40      }
41  }
```

Running the transformed code will start debugging at line 23, that is, after the havoc, and throw a LoopInvariantException at line 39.

Appendix B

Bugs found

While working on Cee, we have found and fixed a number of bugs in related software. We describe the issues in the following sections.

B.1 Cecil

Unfortunately, it seems to produce invalid code for some test cases. We have fixed a simple bug in the local variable handling that was responsible for issues in many of our samples.

While the generated CIL code seems to work correctly in most cases, we have found issues with the generated debug information. Cecil supports reading and writing debug information in PDB files, however, the support does not seem to be very robust. In some cases Visual Studio does not accept the new PDB file at all, while in many other cases, it just misses some local variables, especially variables in subscopes.

To solve these problems we wanted to use the methods GetDebugInfo and DefineLocalVariable2 of ISymUnmanagedWriter and ISymUnmanagedWriter2, respectively. However, these two methods are not exposed by the wrapper classes in System.Diagnostics.SymbolStore. Therefore we wrote our own wrapper classes and interfaces that only wrap the methods we need for PDB support in Cecil.

We have reimplemented the SymWriter class, which is responsible for writing out the debug symbol information, to expose more methods of the unmanaged COM symbol writer interface. This allows us to generate debug information for local variables in a more reliable way and fixes the issues we have been seeing with Visual Studio not recognizing some local variables.

We have also modified the method that patched the generated PDB to match the assembly. It now works the other way round, that is, the method now modifies the generated assembly to match the PDB. The reason for this is that we now can avoid the magic position calculation that depends on the internal structure of the PDB file format, which is neither public nor stable. We use the new GetDebugInfo method together with assembly information that Cecil already has to modify the assembly, that is also the recommended way to write debug information. Our tests have shown that Visual Studio now always accepts the generated PDB file.

We have submitted the two patches[5] upstream, however, the maintainer did not have time to review it so far.

B.2 Common Compiler Infrastructure

Normalizer

The normalizer is a visitor in the CCI framework that converts high-level AST nodes to a lower level suitable for writing CIL code. Cee uses the normalizer to convert contract expressions containing high-level AST nodes. However, the nodes we get when reading the compiled assembly are already normalized.

When normalizing these nodes again, invalid code might be emitted in some cases. Some subexpressions in the original Spec# code are interpreted as expression statements when reading the compiled assembly. The normalizer adds a *pop* instruction after expression statements with non-void expressions, which is wrong in the case of subexpressions and causes a stack underflow when trying to run the rewritten application.

We have worked around this issue by not normalizing expression statements to avoid the extra *pop* instructions.

Invalid branch instructions for runtime checks

We have found a bug in the Spec# compiler that results in throwing exceptions for not violated preconditions (and also vice versa), which can be a quite big problem. The following example has been tested using Spec# 1.0.20411.0:

Test1.ssc

```
public abstract class Foo {
    public abstract void Test(int i);
    requires 0 > i;
}
```

Test2.ssc

```
public class Bar : Foo
{
    public override void Test (int i) { }

    static void Main ()
    {
        Bar bar = new Bar ();
        bar.Test (−1);
    }
}
```

Build the source files to two separate assemblies with

```
$ ssc /t:library Test1.ssc
$ ssc /r:Test1.dll Test2.ssc
```

Test2.exe verifies without errors with Boogie, which is correct as 0 is greater than -1. However, if you run Test2.exe, you will get a bogus RequiresException. The example works fine when compiling everything into one assembly, and therefore avoiding contract deserialization.

The contract deserializer always uses UInt32 or UInt64 for non-negative integer literals. Therefore, the literal 0 in the precondition gets deserialized to a UInt32 value. The normalizer then uses an unsigned comparison when transforming the If statement into a Branch statement, as it only checks whether the first operand is unsigned. In CIL this results in a 'ble.un' instruction, which will not work correctly with negative values.

Using Int32 in the contract deserializer when the value is less than or equal to Int32.MaxValue fixes the problem. However, as the code comments do not describe the reason for always using UInt32 or UInt64, we cannot be completely sure that it will not cause issues at other places.

We have reported the issue to the Spec# mailing list, however, without any response.

Instance initializers in generated expressions

The Resolver class is responsible for resolving types and overloads in the CCI framework. We have noticed that it cannot handle object construction expressions that are already bound to a specific instance initializer, as is the case for generated object construction expressions. We have fixed this issue in the Resolver.

Object creation expressions

Object creation expressions are represented by Construct objects in the abstract syntax tree. The code flattener of CCI creates MethodCall objects from Construct objects, however, it does not propagate the ILOffset value, which prevents Cee from redirecting object creation expressions to mock constructors. We have fixed this issue.

Appendix C

Using Cee

C.1 Debugging counterexample

simple.ssc

```
public class Simple
{
    public int Test (int value)
    ensures result >= 10;
    {
        return value;
    }
}
```

When building the code we need to make sure to enable debug symbols as in the following commandline:

```
ssc /debug+ /target:library simple.ssc
```

This generates a library called simple.dll from the Spec# source code.

Running Cee

```
cee simple.dll
```

This command rewrites the assembly, stores the result in *simple.cee.exe*, and launches the debugger. After selecting a debugger in the appearing dialog box, the debugger window shows *simple.ssc*, the source code of the library. The execution has been paused at the beginning of the method, and usual debugger commands such as step and run can be used to continue execution.

Before leaving the method, the postcondition *result* >= 10 will be checked

C.2 Automatic execution

In addition to interactive debugging, Cee also supports automatic execution of a counterexample that executed code without user intervention. Using the above example, automatic execution of the counterexample can be used as follows:

Running Cee

```
cee --check simple.dll
```

Output

```
expected no exception
caught: Microsoft.Contracts.EnsuresException: Postcondition 'result >= 10' violated from
    method 'Simple.Test(System.Int32)'
```

The expected runtime exception can be specified using the $--$*expect* option to be able to use Cee in test suites.

Running Cee with expected exception

```
cee --check simple.dll --expect "Microsoft.Contracts.EnsuresException: Postcondition 'result
    >= 10' violated from method 'Simple.Test(System.Int32)'"
```

Test suite

The ‘tests’ subdirectory in the source tree of Cee contains more than 30 test cases to ensure that code rewriting and execution works as intended and further changes will not introduce any regressions.

Bibliography

[1] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] Mike Barnett, Robert Deline, Manuel Fähndrich, and K. Rustan M. Leino. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3:2004, 2004.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. pages 49–69. Springer, 2004.

[5] Jürg Billeter. Improve PDB support. http://article.gmane.org/gmane.comp.gnome.mono.cecil/371.

[6] Jürg Billeter. Triggers and counterexamples in Z3. Personal correspondance.

[7] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 422–431, New York, NY, USA, 2005. ACM.

[8] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[9] Leonardo de Moura and Nikolaj Bjrner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[10] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[11] Oren Eini. Rhino.Mocks. http://www.ayende.com/projects/rhino-mocks.aspx.

[12] Jb Evain. Cecil. http://www.mono-project.com/Cecil.

[13] K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2007.

[14] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts, 2003.

[15] Microsoft Research. Pex. http://research.microsoft.com/Pex/.

[16] Nikolai Tillmann and Wolfram Schulte. Mock-object generation with behavior. *ase*, 0:365–368, 2006.

[17] Typemock. Isolator. http://www.typemock.com/.