Master's Thesis

Improving Cee

and

Ownership-Based Verification

Christoph Studer

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

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

April 2009

Supervised by
Prof. Dr. Peter Müller
Arsenii Rudich
Joseph N. Ruskiewicz

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Spec# is a verification system that allows programmers to statically verify object-oriented programs. Cee was developed as Jürg Billeter's master's thesis project to help programmers better understand Spec# verification failures. It generates counterexample programs that exhibit failing execution traces and are debuggable using the well-known debugger user interface.

In this thesis, we enhanced Cee by contributing method side effect capturing, generation of runtime checks for frame conditions, and improved handling of method calls in contracts. Furthermore, our work enables integration of Cee with Visual Studio and other IDEs.

•

As a separate effort, we designed and analyzed heap models for ownership-based verification with the aim to improve verification performance. We evaluated these models within the Spec# system and consider none of them valid replacements for the current heap model. We discuss our findings in this report for future reference.

Acknowledgments

I would like to thank Prof. Dr. Peter Müller for the agreeable collaboration throughout my master's thesis. During part II of this thesis Arsenii Rudich supervised my work, while Joseph N. Ruskiewicz oversaw part I. Thank you for numerous hours of discussions, valuable inputs, and advice.

Furthermore, I would like to thank Yves Alter and Thomas Lenherr for their feedback on this report.

Contents

I Improving Cee **1**

1 Introduction **3**

2 Background **5**
2.1 The Spec# Programming System 5
2.2 Frame Conditions 6
2.3 Boogie 7
2.4 Z3 and Counterexamples 11
2.5 Cee - Counterexample Executor 12

3 Maintaining Object States **15**
3.1 Object Initialization 15
3.2 Object Updates 18

4 Runtime Checks for Frame Conditions **23**
4.1 Addressing Heap Locations 23
4.2 Adding Runtime Checks 24
4.3 Example 25
4.4 Limitations 26

5 Method Calls in Contracts **29**
5.1 Function Tagging 29
5.2 Example 30
5.3 Implementation 31
5.4 Limitations 33

6 Other Improvements to Cee **35**
6.1 Counterexample Selection 35
6.2 Visual Studio Integration 36
6.3 Integration with the *e TextEditor* 36
6.4 Runtime Checks 37
6.5 Debugging Options 37

7 Future Work **39**
7.1 Object Registry 39
7.2 Improving Side Effect Capturing 40
7.3 Improving Runtime Checks for Frame Conditions 40
7.4 Improving Function Tagging 41
7.5 Understanding Verifiable Programs 41

8 Conclusions **43**

II Multiple Heaps for Ownership-Based Verification 45

9 Introduction **47**

10 Background **49**
10.1 Spec# and Boogie 49
10.2 Current Heap Model 49
10.3 Ownership Type System 50
10.4 Problem 53

11 Investigated Heap Models **55**
11.1 Model: *Peer Heaps* 56
11.2 Model: *Paths* 60
11.3 Model: *Heap Variables* 62

12 Conclusions **67**
12.1 General Problems 67
12.2 Future Work 67

III Appendices 71

A Cee Usage **73**
A.1 Usage 73
A.2 Options 73

B Visual Studio Integration **75**
B.1 Cee AddIn 75

C Text Case Definitions **77**
C.1 Example 1: Frame Condition Violation 77
C.2 Example 2: Side Effect Capturing 78

Part I

Improving Cee

Chapter 1

Introduction

Although formal verification techniques have been available to program designers for more than 25 years, it was not until 5 years ago that it became feasible to use for programmers in modern object-oriented systems. Spec# effectively closed the gap between the abstract program specification and its implementation by allowing programmers to write them alongside each other and in the same language.

By employing Spec#, programmers can statically verify critical parts of their systems and guarantee correct functioning for all executions. However, if the verification fails, it is often not obvious to programmers where they made a mistake and how it can be corrected. The mistake might be located in the implementation or, leading to more subtle errors, in the specification.

The counterexample executor *Cee* was developed as Jürg Billeter's master's thesis project in order to help programmers better understand Spec# verification failures. From a counterexample, which is part of the Spec# verification output, it generates a debuggable program that exhibits one particular execution trace leading to the verification failure. By being able to inspect variables and control execution through the well-known debugger user interface, programmers can compare the failing execution to their expectations. They can thereby better understand the verification failure and find mistakes more easily. This thesis improves Cee by adding new functionality and removing limitations of existing features.

In this report, we first give an overview of the Spec# programming system and the general functioning of Cee. In chapter 3 we describe how Cee maintains object states throughout counterexample execution, what limitations we encountered and how we removed them. Chapter 4 presents the support of runtime checks for frame conditions we contributed to Cee, while chapter 5 shows a technique that improves Cee's handling of method calls in program specifications. A collection of smaller enhancements to Cee can be found in chapter 6, and finally, this part of the report is closed by an outlook at possible future work in chapter 7 and the conclusions in chapter 8.

Chapter 2

Background

2.1 The Spec# Programming System

Spec#[4] is a programming system for developing and statically verifying programs. It is developed at Microsoft Research with contributions from various universities and individuals. Spec# allows developers to formally verify whether a program is correct with respect to some specification. It largely consists of the following components:

1. Spec# programming language
2. Spec# compiler
3. Boogie static verifier

Figure 2.1 illustrates the compilation and verification process in the Spec# programming system described in the following:

The system takes Spec# programs (1) as input. The Spec# language is a super-set of C# extending the syntax with keywords and constructs for specifications, sometimes called *contracts*. For example, pre- and postconditions can be specified for a method using the `requires` and the `ensures` keyword, respectively. Refer to listing 2.1 for a simple Spec# program.

Input programs are compiled into .NET CIL assemblies by the Spec# compiler (2). In addition to the compiled program statements, the compiler inserts runtime checks for specifications and also encodes them using special instructions. The output assemblies can be read and executed by any standard .NET virtual machine. Runtime checks guarantee that a particular execution adheres to the specification.

Furthermore, the static verifier Boogie (3) provides the developer with the possibility to statically prove that all executions are correct with respect to the specification.

For the example program in listing 2.1, verification with Spec# yields:

```
Error: Method Vector.Invert(), unsatisfied postcondition:
    y == -old(y)
```

This is because the statement on line 10 wrongly assigns `-x` to `y`. After correcting it to `y = -y` the program is successfully verified.

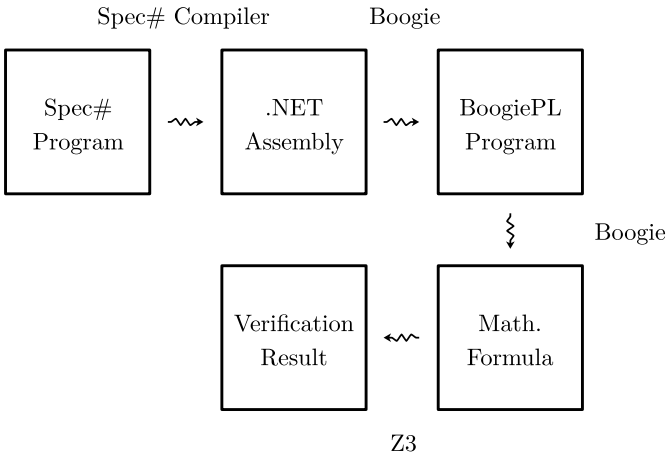

Figure 2.1: The Spec# programming system.

```
1  public class Vector
2  {
3    public int x, y;
4
5    public void Invert()
6      ensures x == -old(x);
7      ensures y == -old(y);
8    {
9      x = -x;
10     y = -x;
11   }
12 }
```

Listing 2.1: Example Spec# program.

2.2 Frame Conditions

Frame conditions define which parts of the object graph a method is allowed to modify. By default, methods in Spec# are only allowed to modify fields on the *this* object, but a programmer can specify a list of modifiable object fields for each method using the `modifies` keyword.

For verification of object-oriented programs, frame conditions are an essential technique to permit reasoning about the state of the heap after a method call. The verifier can assume the object graph regions outside the specified modifies clause to be unchanged after a call, while regions mentioned by the modifies clause might have changed arbitrarily or as specified by postconditions.

Note that frame conditions only cover observable modifications. Objects instantiated within the method are not observable from the outside and can hence be modified without reference in the modifies clause.

Spec# verifies that methods do not violate their frame conditions, neither directly in their body nor by calling methods with weaker frame conditions.

Pure Methods

Methods without observable side effects are called *pure*. In Spec#, only calls to pure methods are allowed from within contracts, so that determining whether a certain contract holds does not change program state. The absence of side effects allows pure methods to be modeled as mathematical functions[7], which facilitates reasoning about pure methods.

2.3 Boogie

Boogie[3] is the central part in the Spec# programming system as it translates Spec# programs into abstract mathematical programs and is therefore responsible for all employed abstractions and mathematical models.

The Spec# input program is first translated into BoogiePL[2], the internal representation of Boogie for abstract programs. After several rounds of abstractions, a formula representing all possible program executions is fed into a theorem prover and checked for validity. The program adheres to the specification if and only if the formula is valid.

Boogie supports different theorem prover back-ends, but best performance is currently achieved using Z3[5], which is developed as a separate project at Microsoft Research.

Consider the example program in listing 2.1. The remainder of this section describes how this program is first translated into BoogiePL, then iteratively processed and transformed into a formula that can be fed into Z3.

Program Translation into BoogiePL

Each method declaration and program instruction is individually translated into a corresponding BoogiePL declaration or instruction, respectively. The resulting initial BoogiePL program therefore closely resembles the input program.

We now present the translations of the Spec# constructs that are most important for the main part of this thesis. Note that some of the translations have been simplified in order to improve readability.

Heap

BoogiePL does not have a built-in concept of a heap. Instead, Boogie translates all heap accesses[1] in the Spec# program into operations on a 2-dimensional array called `$Heap`. For instance, the Spec# field access

```
this.x;
```

where `this` is of type `Vector`, is translated into the BoogiePL commands:

```
assert this != null;
$Heap[this, Vector.x];
```

[1] Heap accesses include field and array element accesses.

Similarly, a Spec# field write

```
this.x = 5;
```

is translated into the BoogiePL commands:

```
assert this != null;
$Heap[this, Vector.x] := 5;
```

The `assert` commands model the runtime behavior of a .NET virtual machine when a field access is performed on a null reference.

Method Declaration and Definition

For each method in the Spec# program, Boogie creates a procedure in BoogiePL that encodes the signature including contracts. The frame condition is translated into an additional postcondition that relates pre-state and post-state heaps. The two heaps are glued together such that all locations are unchanged, except for locations mentioned in the modifies clause.

The following shows and excerpt of the procedure generated for the `Invert` method:

```
procedure Vector.Invert(this: ref);
...
  // user-declared postconditions
  ensures $Heap[this, Vector.x] == 0 - old($Heap[this,
      Vector.x]);
  ensures $Heap[this, Vector.y] == 0 - old($Heap[this,
      Vector.y]);
```

Procedures are used by Boogie for look-up of the method signature and contracts when the corresponding method itself is verified or when it is called from a different method.

The actual implementation of each method (if available) is translated into an implementation in BoogiePL. The following is an excerpt of the implementation for the `Invert` method:

```
1  implementation Vector.Invert(this: ref)
2  {
3    var stack0i: int;
4
5    ...
6      // ----- load field -----
7      assert this != null;
8      stack0i := $Heap[this, Vector.x];
9      // ----- unary operator -----
10     stack0i := 0 - stack0i;
11     // ----- store field -----
12     assert this != null;
13     $Heap[this, Vector.x] := stack0i;
14   ...
15
16 }
```

The excerpt above shows the translated field load, negation, and field store on line 9 in the source Spec# program. Intermediate values for sub-expressions are assigned to the variable `stack0i`.

Method Call

How a Spec# method call is translated into BoogiePL depends on the location of the call. Method calls that are located within regular C# statements are translated into `call` commands in BoogiePL. Suppose the assignment `x = -x` on line 9 of the example Spec# program was replaced by `x = -GetX()`, where `GetX` is a method that simply returns `x`. The following would be the BoogiePL translation for the `GetX` call:

```
// ----- call -----
assert this != null;
call stack0i := Vector.GetX(this);
```

If a method call is located within a contract, however, it is translated into a mathematical function expression. This is necessary as contracts are preserved as Boolean expressions in BoogiePL, which cannot contain method calls. Calls from within contracts can be modeled as function expressions because the called methods are required to be pure.

`GetX` is such a pure method that can be used in a contract. Suppose the first postcondition `x == -old(x)` on line 6 in the Spec# program was replaced by `GetX() == -old(GetX())`. The BoogiePL translation of this contract would then be:

```
ensures #Vector.GetX($Heap, this) == 0 - old(#Vector.GetX
    ($Heap, this));
```

Boogie would also declare an uninterpreted function[2] in BoogiePL:

```
function #Vector.GetX(heap, ref) returns (int);
```

Only one uninterpreted function per pure method is created and is used for all calls from contracts. The postcondition of the method is encoded using an axiom which states that the return value is equal to `this.x`:

```
axiom (forall $h: heap, this: ref ::
    #Vector.GetX($h, o) == $h[this, Vector.x])
```

By applying this axiom to the modified postcondition, one can see that it is in fact equivalent to the original postcondition, the translation of which was shown above.

Translation from BoogiePL to a Formula

Having translated the Spec# program into BoogiePL, Boogie then verifies implementation by implementation. Z3 takes a mathematical formula as input, which means that further abstraction is required.

[2] An uninterpreted function is a functions with a name and type, but no actual function definition.

Amongst others, the following transformations are applied to the BoogiePL program before it can be fed into Z3:

1. Preconditions of the verified implementation are added as assumptions at the beginning, postconditions as assertions at the end.
2. Calls are replaced by contracts of the called procedure. Preconditions become assertions, postconditions become assumptions.
3. Commands are translated into passive commands.

Transformations (1) inject the contracts of the implementation under verification. When Boogie replaces method calls by their contracts during transformation (2) the precondition of the called method is asserted followed by a havoc[3] of the heap. Then, the postcondition of the method is assumed. As the frame condition is encoded in the postcondition, side effects are properly reflected on the heap.

Transformation (3) primarily transforms all assignments into assumptions. In order to keep the process mathematically correct, fresh variables have to be introduced when assuming new facts on them. This essentially translates the program into single static assignment (SSA) form and the introduced variable versions are called incarnations. Incarnations for the original variable `$var` are named `$var@0`, `$var@1`, and so on.

Consider the `Invert` implementation from before. The following excerpt shows how the implementation uses assume and assert statements only after having applied the 3 transformations explained above:

```
1  implementation Vector.Invert(this: ref)
2  {
3    var stack0i@0: int, stacki@1: int;
4    var $Heap: heap, $Heap@0: heap;
5
6    ...
7      // ----- load field -----
8      assert this != null;
9      assume stack0i@0 == $Heap[this, Vector.x];
10     // ----- unary operator -----
11     assume stack0i@1 == 0 - stack0i@0;
12     // ----- store field -----
13     assert this != null;
14     assume $Heap@0 == $Heap[this, Vector.x := stack0i@1];
15   ...
16
17 }
```

Note the expression on the right hand side on line 14. This is syntactic sugar for a universal quantification glueing `$Heap@0` and `$Heap` together by leaving all entries intact except for the entry `[this, Vector.x]`, on which a new value `stack0i@1` is assumed.

The resulting implementation in SSA form is translated into a mathematical formula and fed into Z3, which then tries to prove its validity.

[3] Havocing a variable removes all assumptions on it.

2.4 Z3 and Counterexamples

When a formula is checked for validity but found invalid, an unsatisfying assignment can be presented as proof. Z3 is able to produce such an unsatisfying assignment, called a *counterexample*. If the command line option `/printModel` is given to Boogie, it prints the respective Z3 counterexample along each verification failure.

Z3 was optimized for program verification. It delivers high performance as a back-end to Spec# and is able to provide partial models, that is counterexamples only containing relevant information about program execution up to a violated contract. Important theories such as integer arithmetics, uninterpreted functions, and quantifiers are directly supported in Z3.

Partitions

The abstraction for any value in the counterexample is called *partition*. Each partition represents an identity of an element in the universe of the counterexample. Elements can be anything that requires definition, for example BoogiePL variables, types, type fields, object instances, and so on. These elements all stem from variables and values in the input formula.

Counterexamples contain mappings from elements to their partition values. If two elements map to the same partition, they are in fact equal.

Consider the following excerpt from the counterexample of the example program in listing 2.1:

```
*0: True
*1: False
...
*6: -1  {stack0i@0}
...
*11: System.Object
...
*43: Vector.x
*45: Vector.y
*47: Vector
...
*58: 0
...
*82: $Heap@0
...
```

The left-hand-side denotes the partition while the right-hand-side lists elements that were assigned that partition. Partition 0 and 1 are used for Boolean values. Partition 6 maps to the integer literal `-1` as well as to the BoogiePL variable `$stack0i@0`, which means that `$stack0i@0` is equal to `-1`. Partitions 11, 43, 45, and 47 map to types and type fields.

Function Interpretations

Similarly to the fact that counterexamples contain actual values for variables in the input formula, uninterpreted functions are assigned function interpreta-

tions. For each function in BoogiePL, the counterexample contains a list of interpretations which assign results to function expressions with actual input parameters.

For example, for the pure function `GetX` introduced in the previous section 2.3 the counterexample contains the following interpretation:

```
#Vector.GetX($Heap, this) = 0
```

This means that the function expression in the postcondition was assigned a result of `0` by Z3. Note that the arguments and the function results are given as partition values. They are shown as resolved elements here for improved readability.

Heap Contents

In section 2.3 we explained how heap accesses are translated into accesses on a 2-dimensional array called `$Heap` in BoogiePL. As part of a further abstraction step, these array accesses are replaced by `select2` function expressions. For example, the field read `this.x` on line 9 of the example program in listing 2.1 is transformed into the following function expression:

```
select2($Heap, this, Vector.x)
```

Heap contents in the counterexample can therefore be found in the list of `select2` function interpretations:

```
select2($Heap, this, Vector.x) = -1
select2($Heap@0, this, Vector.x) = 1
...
select2($Heap, this, Vector.y) = 0
select2($Heap@1, this, Vector.y) = -1
```

Using these interpretations, object field values can be extracted for all heap incarnations.

2.5 Cee - Counterexample Executor

The counterexample executor *Cee*[1] was developed as Jürg Billeter's master's thesis project at the Chair of Programming Methodology, ETH Zurich. It is a tool that helps programmers better understand Spec# verification failures. Cee allows programmers to debug a method execution that exhibits a verification failure using the familiar debugger user interface. Variables can be inspected and the program execution can be controlled similar to a runtime error in a test case.

Consider the diagram in figure 2.2. Given a verification failure, Cee reads the original method from the assembly produced by the Spec# compiler. It then triggers and intercepts the verification process in order to obtain a counterexample with information about a failing method execution. Cee uses this information to rewrite the original program such that it can be run to reproduce the verification failure.

Rewritings performed on the original method include:

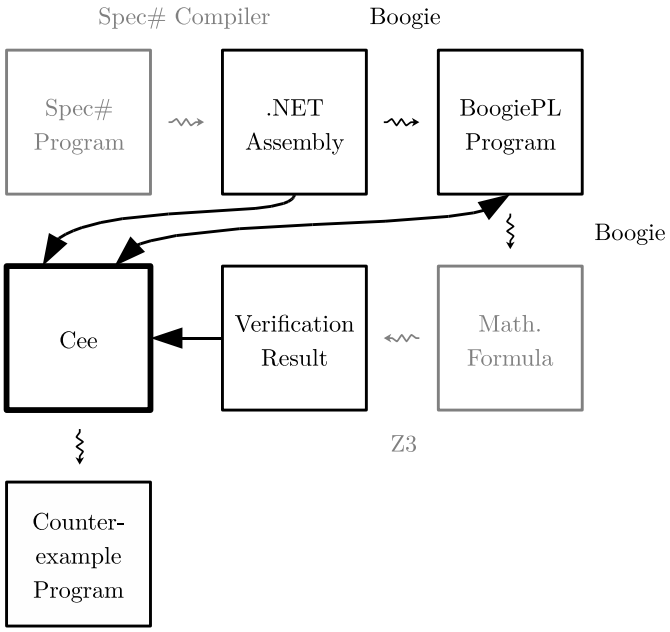

Figure 2.2: The counterexample executor *Cee*.

- Initializing parameters including *this*
- Replacing method calls with calls to mock methods that behave as specified in the counterexample
- Rewriting loops to match the counterexample trace
- Adding runtime checks at the location of the verification failure

By applying these rewritings, Cee effectively provides the method with the initial state and interface behaviors necessary to reproduce the verification failure.

Cee then outputs the rewritten program as a .NET assembly that calls the method when executed. Attaching a debugger allows the inspection of variables and control of program execution, since debugging information is preserved throughout the rewrite.

Consider the Spec# program in listing 2.2. Verification yields an assert violation on line 8, because the return value of `Decrement(10)` is `9`, and consequently `x != 8`. Giving this program and the verification failure to Cee results in the counterexample program in listing 2.3[4]. Note how each call to a method was replaced by a mock method call that behaves identically to the called method, even if there is no implementation available. The rewritten method `Foo` can be debugged and exhibits the assert violation at the appropriate location.

While Cee was already a useful tool prior to this thesis, it suffered from limitations that constricted the set of reproducible counterexamples. The goal of this part of the thesis was to eliminate some of these limitations, and the achievements of this undertaking are presented in the following chapters.

[4] A simplified version is shown here for improved readability.

```
1  public abstract class CeeExample
2  {
3    public void Foo()
4    {
5      if (Decrement(5) == 4)
6      {
7        int x = Decrement(10);
8        assert x == 8;
9      }
10   }
11
12   public abstract int Decrement(int input)
13     ensures result == input - 1;
14 }
```

Listing 2.2: Spec# program with assert violation on line 8.

```
1  public abstract class CeeExample
2  {
3    public void Foo()
4    {
5      if (DecrementCee975(this, 5) == 4)
6      {
7        int x = DecrementCee1026(this, 10);
8        if (x == 8)
9        {
10         throw new ContractException("Unable to reproduce
               verification failure");
11       }
12       throw new AssertException("Assertion 'x == 8'
             violated from method 'CeeExample.Foo'");
13     }
14   }
15
16   private static int DecrementCee1026(CeeExample self,
       int input)
17   {
18     return 9;
19   }
20
21   private static int DecrementCee975(CeeExample self, int
       input)
22   {
23     return 4;
24   }
25 }
```

Listing 2.3: Cee output for the assert violation in listing 2.2.

Chapter 3

Maintaining Object States

Cee needs to ensure that runtime states of objects reflect the counterexample, which means that object states must be correctly initialized and kept up-to-date during counterexample execution.

When calling a method with side effects, parts of the heap may change depending on the frame condition and postcondition. Prior to this thesis, Cee did not establish correct object states after method calls, and hence, side effects were lost. Generated counterexample programs where flawed for programs where the further execution depends on side effects of a method call.

We addressed this issue and added side effect capturing to Cee. During development of side effect capturing, we discovered and fixed some flaws and missing functionality in Cee's object initialization.

This chapter describes the implementation of object initialization and object updating, limitations we found and fixed as part of this thesis, as well as remaining limitations related to object state maintenance.

3.1 Object Initialization

Objects are created by Cee whenever they enter the method under verification. There are four interface points where this can happen:

1. Input parameters of the method under verification
2. Return values of called methods
3. Explicit constructor calls
4. As part of recursive initialization of object fields

By controlling these four interface points, Cee guarantees that new objects the program deals with resemble the information gathered from the counterexample.

A flaw in Cee when handling case (3) caused it to initialize the resulting objects incorrectly in the following case: When the result of an explicit constructor was assigned to a local variable that was previously assigned another object instance, Cee wrongly initialized the new object with values from the previous instance. By regarding an explicit constructor call as a regular method call with side effects and employing the newly added capturing of side effects

presented in the next section we were able to fix this. Case (3) will hence be discussed in section 3.2.

Object Types

In order to find initial values of object fields, Cee needs to determine the partition of the object. For cases (1) and (2) we can reconstruct the names of the unique BoogiePL identifier of the resulting object: (1) A parameter with name `p` in Spec# is named `p$in` in BoogiePL. (2) Return values of method calls are assigned to specially named variables in BoogiePL that we can reconstruct by using the unique ID of the Boogie method call. Therefore, in cases (1) and (2), we can perform a simple look-up by identifier in the counterexample to retrieve the partition of the object to initialize. As these identifiers are for one-time use we do not have to care about different incarnations and can simply resolve the plain identifier.

In case (4), Cee is recursively initializing a reference type field as part of another object's initialization. As the field to initialize was read from the counterexample, the partition value is readily available.

Now that the partition of the object to initialize is known, Cee can scan the heap information in the counterexample. Cee searches for `select2(h,o,f)=v` function interpretations where `o` equals the partition determined for the object to initialize. It then finds for each field `f` the entry with the first available heap incarnation. This entry contains the initial value partition `v` of the object field `o.f`. Cee has now a mapping from fields `o.f` to value partitions `v` that it can use to initialize the fields of object `o`.

Initialization of these fields are performed in a parameter-less mock type created by Cee. Prior to this thesis, Cee was limited to create only one mock type per actual type `T`. The same mock constructor was called for all instances of this type, and hence some instances where initialized with incorrect field values.

We fixed this flaw as part of this thesis. For each object initialization of type `T`, Cee creates a new subclass with the name `T` with a unique number of spaces appended[1] containing a parameter-less constructor and default implementations for possible abstract or unimplemented methods.

The mapping from fields `o.f` to value partitions `v` is turned into a list of assignment statements inside the constructor. If `f` is of a custom object type, array type or string, object initialization is performed recursively for that field, leading to case (4) described above. If `f` is a basic type, for instance `Int32` or `bool`, it is directly assigned a literal with the correct value.

Arrays and Strings

Arrays and strings require separate treatment as they are specially encoded in Boogie.

The length of an array is recorded using the `$Length(arr)=len` function. We extract this information from the counterexample to initialize an array object of correct length.

Prior to this thesis, Cee did not initialize the array elements of the instantiated array. We added support for array element initialization to Cee. First,

[1] `T␣+` is used so that object instances look as if they were of type `T` in a debugger.

```
1  public class VectorCalculator
2  {
3    public Vector! Add(Vector! a, Vector! b)
4    {
5      int x = a.x + b.x;
6      int y = a.y + b.y;
7      return CreateVector(x, y);
8    }
9
10   public Vector! CreateVector(int x, int y)
11     requires x >= 0 && y >= 0;
12     ensures result.x == x && result.y == y;
13   {
14     return new Vector(x, y);
15   }
16 }
```

Listing 3.1: Example Spec# program illustrating object initialization.

the partition of the pseudo field `$elements`, which represents all elements in the array, is extracted from the heap. Then, Cee scans the function interpretations of `RefArrayGet($elements,index)=v` or `IntArrayGet($elements,index)=v` for element values, depending on the type of the array. Array elements are then recursively initialized and stored into the array.

For strings, only the length is preserved using the `$StringLength(str)=v` function in Boogie. When initializing a string, Cee scans `$StringLength(str)` interpretations for an entry with the string to construct and then initializes a string with the extracted length. Because the actual content of strings is not encoded in Boogie, Cee initializes dummy strings containing only "a" characters.

Example

Consider the example Spec# program in listing 3.1. Verification of the `Add` method using Boogie yields a failure, as we cannot guarantee that `x >= 0 && y >= 0` holds true on line 7, and hence the precondition of `CreateVector` might be violated. When executing the counterexample using Cee, the two input parameters `a` and `b` require initialization. We are in case (1) described above and therefore Cee inspects the `select2` interpretations for information about `a$in` and `b$in`. The relevant interpretations found in the Z3 counterexample are:

```
select2($Heap, a$in, Vector.x) = -1
select2($Heap, b$in, Vector.x) =  0
select2($Heap, a$in, Vector.y) =  0
select2($Heap, b$in, Vector.y) = -1
```

Using this information, Cee is able to generate constructors for the mock types of `a` and `b`. The following listing shows the mock type created for `a`, where the fields are properly initialized in the constructor[2]:

```
public class Vector_ : Vector
{
    public Vector_()
    {
        base.x = -1;
        base.y = 0;
    }
}
```

Consequently, when running the generated counterexample program, the debugger shows the two vectors `a` and `b` properly initialized with values `(-1,0)` and `(0,-1)`, respectively:

Name	Value
⊞ this	{VectorCalculator }
⊟ a	{Vector }
⊞ [Vector]	{Vector }
x	-1
y	0
⊟ b	{Vector }
⊞ [Vector]	{Vector }
x	0
y	-1

3.2 Object Updates

There are two cases where fields of objects can change during execution of a method: Fields are updated explicitly in the function body by assigning new values or fields are updated due to side effects of a called method. As Cee leaves intact field assignment instructions from the original program, no extra work is required to cover these updates. Method calls, however, are replaced by calls to mock methods and therefore Cee needs to ensure they exhibit the side effects defined in the counterexample.

Constructors with field initialization can be regarded as special cases of regular methods with side effects. We therefore discuss case (3) of the previous section about object initialization here, together with regular methods calls, under the term "method calls with side effects".

Side effect capturing was added as part of this thesis, and in the following we describe the functioning of this feature as well as its limitations.

Side Effects and Frame Conditions

Method side effects are closely related to frame conditions, as Spec# guarantees that no method modifies observable heap locations that are not specified in its

[2] "_" characters in the class name represent space characters.

frame condition. Hence any side effect must adhere to the frame condition specified by the programmer. As frame conditions are encoded as postconditions, possible side effects are properly reflected on the heap after a method call.

Side Effect Capturing

In order to capture side effects of method calls, the appropriate post-state heap incarnations have to be determined. Since there is no mapping associating heap incarnations with locations in the program, we developed a technique that allows Cee to determine the valid heap incarnation after method calls.

We achieve this by modifying the Boogie translation of the input program. After each method call with possible side effects, that is after each `call` command in the BoogiePL program, Cee inserts a heap update to a dummy field called `$ceeLastCalledMethod` with the ID of the method call as the new value. These field updates are reflected in the counterexample as function interpretations of `select2`. Cee is then able to look up the valid heap incarnation after a certain method call by filtering `select2` interpretations for `$ceeLastCalledMethod` entries with a value equal to the method's ID. As the dummy heap update introduces a new heap incarnation, the preceding incarnation contains the method call's potential side effects.

Cee scans the determined heap incarnation for field updates on objects that can be addressed from the counterexample program. Only objects where the partition and their C# expression in the counterexample program are both known can be considered for side effects. This is not the case for all objects, as Cee does not keep track of created object instances. By inspecting the parameters of the `CallCmd` Boogie node, however, Cee is able to infer their partition in the counterexample and the C# expression is readily available from the read assembly. In addition to the parameters, Cee knows how to address the *this* object and its partition can be easily extracted from the counterexample, too. Therefore, Cee is able to reflect side effects on the current *this* object and on parameters passed to the called method[3]. Fields of these objects are updated using object initialization as described in the previous section from within the mock methods or the mock constructors, respectively.

Heap Simplification

In some cases, the counterexample contains the same value for object fields in consecutive heap incarnations. In order to prevent Cee from doing unnecessary field updates, we first simplify the `select2` interpretations to only represent new field values. Interpretations in consequent heaps in `select2` with equal values for a certain field can simply be removed.

We do this by scanning the `select2` interpretations by increasing heap incarnations, during which we store the last seen value for each object field `o.f`. When encountering an interpretation with a new value for `o.f`, we update the last seen value accordingly and keep the interpretation. If the value is equal to the last seen value, however, we remove the interpretation and continue the scan. After this procedure has been applied to the counterexample, the `select2` interpretations contain heap updates only.

[3] For simplicity, we only implemented updates on the `this` object and on the first parameter, which corresponds to the method call target.

```
1  public class Vector
2  {
3    public int x, y;
4
5    public void Add(Vector! other)
6      requires other.x > 0;
7      requires other.y > 0;
8      ensures y > 0 && x > 0;
9    {
10     Add(other.x, other.y);
11   }
12
13   public void Add(int x, int y)
14     ensures this.x == old(this.x) + x;
15     ensures this.y == old(this.y) + y;
16   {
17     this.x += x;
18     this.y += y;
19   }
20 }
```

Listing 3.2: Example Spec# program illustrating method side effects.

Example

Similarly to the example in the previous section, we illustrate method updates with a vector addition example in listing 3.2. This time, the addition is directly performed on a vector. Verification with Boogie returns a failure for the `Add(Vector!)` method, as it cannot guarantee that `y > 0 && x > 0` holds true in its post-state. The actual update of the fields `x` and `y` is performed in the method call of `Add(int,int)` on line 10, and therefore capturing the side effects of that call is crucial for generating the counterexample program.

The first step is to extract the correct heap incarnation valid after the method call. The following is an excerpt of the generated BoogiePL program:

```
1 // ----- call -----
2 assert this != null;
3 call Vector.Add$System.Int32$System.Int32(this, stack0i,
    stack1i);
4 $Heap[cee, $ceeLastCalledMethod] := 5407;
```

Note the dummy update inserted by Cee on line 5, which is reflected in the `select2` interpretations of the counterexample:

```
select2($Heap@1, cee, $ceeLastCalledMethod) = 5407
```

Now that the heap incarnation valid after the dummy field update is known to be `$Heap@1`, the side effects can be extracted from the preceding incarnation `$Heap@0`:

```
select2($Heap@0, this, Vector.y) = 0
...
select2($Heap@0, this, Vector.x) = 1
```

Cee is able to deduce the side effects of the method, which are updates of `this.x` to 1 and `this.y` to 0. These field updates are then included in the corresponding mock method in the counterexample program:

```
private static void AddCee5412(Vector self, int x, int y,
    Vector caller)
{
  if (self == null)
  {
    throw new NullReferenceException();
  }
  caller.y = 0;
  caller.x = 1;
}
```

Limitations

As discussed above, this technique of capturing method side effects only works for objects we can address in C#, which are currently the called method's parameters as well as the *this* object. For other objects, Cee is able to extract side effects from the counterexample, but reflecting them using field updates is impossible since these objects cannot be addressed.

We think that capturing side effects on parameters and *this* is sufficient for the most common cases. It is trivial, however, to find verification failures that cannot be correctly reproduced due to this limitation. Consider, for example, the following Spec# program:

```
1  public abstract class UpdateLimitation
2  {
3    public UpdateLimitation! next;
4    public int x;
5
6    public void Foo()
7      requires next.x == 0;
8    {
9      Update();
10     assert next.x == 0;
11   }
12
13   public abstract void UpdateNextX()
14     modifies next.x;
15     ensures next.x == 1;
16 }
```

The method `Update` has a side effect that does not directly affect *this*, but `this.next`. When reading the side effect from the counterexample, Cee cannot address the corresponding object. The update of `this.next.x` to `1` is not captured and the assert failure on line 9 cannot be reproduced, as the value of `this.next.x` remains 0 in the counterexample program.

Section 7.2 presents an idea that could enable the capturing of side effects on arbitrary heap locations.

Chapter 4

Runtime Checks for Frame Conditions

To our knowledge, there is currently no programming system that supports runtime checks for frame conditions. This is because generic runtime checks essentially require taking a snapshot of the heap at a method's pre-state and comparing it to the heap contents at the end of the method. Potential modifications are then matched against the method's frame condition specification. If a heap modification is not legitimate according to the frame condition, an exception is thrown.

This procedure is impractical for various reasons: Access to the heap as a whole is usually not granted to programs. Therefore, snapshotting at the beginning of the method requires recursive cloning of all reachable objects, an operation that is too resource intensive for complex systems. Finding differences between object graphs and matching them against specified frame conditions is also too costly to perform in large systems.

However, we were able to extend Cee with the capability to generate runtime checks for frame condition violations by exploiting information from the counterexample. When building a counterexample program in Cee, we have to consider only one particular instance of a frame condition violation. This situation is different from generating generic runtime checks, since the Z3 counterexample contains information about the exact heap location of the violation.

Having the counterexample available, we are able to generate runtime checks for a frame condition violation, observing only the heap location necessary to reproduce the verification failure. The remainder of this chapter discusses how heap locations reported by Z3 are translated into corresponding C# expressions, and how runtime checks are generated and inserted into the method under verification.

4.1 Addressing Heap Locations

From the counterexample, we get a BoogiePL identifier and an object field that specify a heap location which is different in the pre- and post-states of the method under verification. If we want to observe that heap location from our generated counterexample program, it has to be translated into a C# expression

that is valid at the beginning of the method where we observe the pre-state. The identifier reported in the counterexample might however be invalid or uninitialized at the beginning of the method or it might have no corresponding expression in the original Spec# program at all.

In order to be able to construct a valid C# expression, it is important to understand the following fundamental property about frame condition violations: Frame condition violations can only occur if the heap modification is observable from the caller. The only common handles into the heap, shared by both the caller and the callee, are the parameters (possibly including *this*). Therefore, frame condition violations can only occur at locations that are reachable via the method's parameters.

Given this property, we know that any heap location reported as frame condition is reachable by a series of field or array element accesses, starting at some parameter. As the counterexample contains information about object fields and array elements, Cee can perform a depth-first search starting with the partition of the reported identifier. It then traverses object field and array element information in the counterexample to recursively trace the modified heap location to a parameter. If an identifier can be successfully traced to a parameter, the yielded trace can then be translated into a C# expression valid at the beginning of the method.

4.2 Adding Runtime Checks

In order to be able to compare pre- and post-states of the heap location reported in the counterexample, Cee needs to store the initial value at the beginning of the method, then compare it to the current value immediately before returning from the method.

By constructing a C# expression as explained in the previous section, the modified object can be addressed. It is stored into a local variable called `$ceeFrameConditionObject` at the beginning of the method. Then, the pre-state is preserved as follows: If the modified object is an array, a shallow clone is created and stored into a local variable called `$ceeFrameConditionOldValues`, otherwise the value of the modified field reported by the counterexample is stored into a local variable called `$ceeFrameConditionOldValue`.

At the end of the method, that is before each `Return` node in the AST, the runtime check is inserted. In the case of a regular object, the current value of the modified field of `$ceeFrameConditionObject` is compared to the preserved value in `$ceeFrameConditionOldValue`. In the case of an array object, a loop is inserted that compares every element in `$ceeFrameConditionOldValues` to its current value accessed through `$ceeFrameConditionObject`.
A `Microsoft.Contracts.ModifiesException` is thrown upon differences in pre- and post-states, containing a human readable description of the modified heap location[1].

```
1  public class VectorCalculator
2  {
3    public void InvertNext(Vector! vector)
4    {
5      Vector! local = vector.next;
6      local.x = -local.x;
7      local.y = -local.y;
8    }
9  }
10
11 public class Vector
12 {
13   [Peer]
14   public Vector! next;
15   public int x, y;
16 }
```

Listing 4.1: Example Spec# program with a frame condition violation.

4.3 Example

Consider the example Spec# program in listing 4.1. Verification with Boogie yields a frame condition violation of method `InvertNext`. The default frame condition of this method encompasses all fields of the *this* object and hence modifications on fields of the `vector.next` object are disallowed. For frame condition violations, extended information is returned by Z3 that describes the heap location of the modification:

```
1  Error: Method VectorCalculator.InvertNext(Vector! vector)
       , unsatisfied frame condition
2    ...
3    (internal state dump): $f == Vector.y
4    (internal state dump): $o == local@0
5    (internal state dump): $o.$f == -1
```

From the internal state dump on line 3 and 4 Cee can extract the field and the object violating the frame condition. In order to construct the runtime check, the BoogiePL identifier `local@0` needs to be translated into a C# expression. Cee follows field accesses and array element accesses as described above. The following is the relevant `select2` interpretation that allows Cee to deduce that `local@0` can be addressed in the counterexample program using `vector.next`:

```
select2($Heap, vector$in, Vector.next) = local@0
```

Given this expression and the field extracted from the state dump, Cee generates the necessary runtime checks reproducing the frame condition violation. Listing 4.2 shows the rewritten `InvertNext` method in the generated counterexample program.

[1] For instance “this.refArray[0].x” or “param1.x.y”.

```
1  public void InvertNext(Vector modopt(NonNullType) vector)
2  {
3    // ...
4    Vector $ceeFrameConditionObject = vector.next;
5    int $ceeFrameConditionOldValue =
         $ceeFrameConditionObject.y;
6    Vector local = vector.next;
7    local.x = -local.x;
8    local.y = -local.y;
9    if ($ceeFrameConditionOldValue !=
         $ceeFrameConditionObject.y)
10   {
11     throw new ModifiesException("Modifies clause"
12       + " violated from method"
13       + " 'VectorCalculator.InvertNext(... Vector)'."
14       + " Value of 'vector.next.y' differs in" +
15       + " pre- and post-state.");
16   }
17   throw new ContractException("Unable to reproduce" +
18     + " verification failure");
19 }
```

Listing 4.2: Rewritten method generated by Cee for listing 4.1.

4.4 Limitations

Under some circumstances, the reported heap location violating the frame condition has an object identifier that does not occur in any other parts of the counterexample. In other cases, only the modified field without the object is reported. Cee is currently unable to generate frame conditions without an exact heap location and stops with an error message.

An example of a case where the object of the frame condition violation is unknown is shown in the following:

```
public abstract class FcLimitation
{
  public int x;
  public FcLimitation! b;

  public void Foo()
    modifies x;
  {
    Update();
  }

  public abstract void Update()
    modifies b.x;
    ensures b.x == 5;
}
```

Even though it seems clear that the call to `Update` modifies `this.b`, which violates the frame condition of `Foo`, the counterexample does not provide Cee with the object that was modified:

```
Error: Method FcLimitation.Foo(), unsatisfied frame
    condition
        ...
        (internal state dump): $f == FcLimitation.x
        (internal state dump): $o.$f == 5
```

In contrast to the example presented in the previous section, `$o` cannot be extracted from the internal state dump. Note that this limitation is not related to the use of abstract classes or methods. The same counterexample is produced after changing `Update` to a concrete method.

Cee is currently unable to generate frame condition runtime checks for this example. In section 7.3, we discuss how counterexamples without valid heap locations could be handled.

Chapter 5

Method Calls in Contracts

As seen in section 2.3, method calls outside of contracts are translated into `call` commands in BoogiePL. These `call` commands provide enough context to be uniquely associated with originating method calls in the input program as well as with return values in the counterexample. By exploiting these unique mappings, extraction of return values can be performed using simple identifier look-ups in the counterexample.

Extracting return values for calls within contracts is more complex. Since these calls are translated into function expressions, no `call` commands are generated. In order to extract return values from the counterexample, the corresponding function interpretation must be found without having similar context available as provided by `call` commands. Hence, more complex techniques than an identifier look-up are required.

Prior to this thesis, Cee employed static parameter matching in order to find the corresponding function interpretation. Parameters in the Spec# method call were statically evaluated to actual values and the function interpretations where searched for entries with matching parameters.

This technique did not work in many cases, since static evaluation of parameters requires explicit support for each type of expression. For example, Cee was able to statically determine the value of integer additions such as `5 + 5`, while divisions such as `10 / 2` were not supported. When unsupported expressions were used in parameters, Cee was unable to find the corresponding function interpretation and extract the return value.

While we could have added support for more expression types to the static parameter evaluator of Cee, we chose an entirely different approach that scales better and furthermore allows to map method calls uniquely to their corresponding interpretations. The following sections describe this technique we call *function tagging* and discuss how it works and its limitations.

5.1 Function Tagging

In order to be able to identify which function interpretation in the counterexample corresponds to a certain method call from a Spec# contract, function expressions are tagged by a unique identifier (UID) assigned to its originating Spec# method call.

```
1  public abstract class Test
2  {
3    public void Foo(int i)
4      requires IsEven(i);
5    {
6      assert IsEven(ReturnInput(i));
7      assert IsEven(ReturnInput(i) + 1);
8    }
9
10   [Pure]
11   public abstract bool IsEven(int input)
12     ensures result == (input % 2 == 0);
13
14   [Pure]
15   public abstract int ReturnInput(int input)
16     ensures result == input;
17 }
```

Listing 5.1: Spec# program which requires function tagging.

A new wrapper function is declared in BoogiePL for each pure method that is called from a contract. The wrapper function introduces a new integer parameter for the UID tag and is tied to the original function using an axiom that drops said parameter. This axiom guarantees that the semantics remains the same, as the wrapper function just passes through all parameters to the original function and all axioms about this original function remain in effect.

By passing the UID to the wrapper function, Cee is later on able to identify the function interpretation in the counterexample corresponding to a certain contract method call. It can simply search the interpretations for the entry where the last parameter is equal to the method call UID and extract the return value.

5.2 Example

Consider the Spec# program in listing 5.1. When verifying method `Foo`, Boogie reports a possible assertion violation on line 7, as `ReturnInput(i)+1` is odd. In order to generate the counterexample program, Cee has to create a runtime check for the failed assertion and extract the return value of the `IsEven` call from the counterexample. Without method tagging, Cee was unable to find the corresponding interpretation, as the value of `ReturnInput(i)+1` cannot be statically determined.

With function tagging, however, the counterexample contains the following tagged function interpretations:

```
tagged#Test.IsEven$...($Heap, this, -2, -1889192024)=@true
tagged#Test.IsEven$...($Heap, this, -2, -1732590203)=@true
tagged#Test.IsEven$...($Heap, this, -1, -1732524654)=*88
```

We can see that there are three separate interpretations, two of which evaluating to *true* and one without explicit value (partition `88` with no element), in which case *false* can be assumed. The two first interpretations are for the contract method call in the requires clause of `Foo` and for the first assertion on line 6. The third interpretation corresponds to the call in the failing assertion on line 7. This interpretation originates from the translated assertion statement in BoogiePL:

```
// ----- serialized AssertStatement -----
assert tagged#Test.IsEven$System.Int32($Heap, this,
    tagged#Test.ReturnInput... + 1, -1732524654);
```

Knowing the UID of the method call on line 7, `-1732524654`, Cee is able to correctly look up the third interpretation with a value of *false*. This falsifies the assertion, such that the verification failure is correctly reproduced.

5.3 Implementation

Tagging in Boogie

Tagging has been implemented by patching Boogie. When deserializing contract expressions into C# expressions from the read Spec# program, UIDs are assigned to the generated calls and stored into the `ILOffset` fields of their corresponding AST call node. These `ILOffsets` are then turned into UID parameters when translating the program into BoogiePL.

In order to be able to pass this additional UID parameter, a new function is introduced in BoogiePL as briefly mentioned in section 5.1. The following excerpt shows the original function definition (line 2), the newly introduced tagged function (line 3), and the glueing axiom (lines 6-9) for the `IsEven` method in listing 5.1:

```
1 function        #Test.IsEven...(heap, ref, int) returns (
    bool);
2 function tagged#Test.IsEven...(heap, ref, int, int)
    returns (bool);
3 ...
4 // tagged pure methods connect to untagged version
5 axiom (forall $h: heap, o: ref, input$in: int, t: int ::
    { ... }
6     tagged#Test.IsEven...($h, o, input$in, t)
7       <==>
8           #Test.IsEven...($h, o, input$in));
```

An example for a generated tagged BoogiePL call was presented in the previous section.

Extraction in Cee

Identically to Boogie, when Cee deserializes contract expressions into C# expressions, UIDs are assigned to the generated calls and stored into the `ILOffset` fields of their corresponding AST call node. Cee uses the same component as

Method call	Line	uidStr
IsEven(i)	4	Test.Foo(System.Int32).req.0.0
IsEven(...)	6	Test.Foo(System.Int32).body.47.0
ReturnInput(i)	6	Test.Foo(System.Int32).body.47.1
IsEven(...)	7	Test.Foo(System.Int32).body.96.0
ReturnInput(i)	7	Test.Foo(System.Int32).body.96.1

Table 5.1: Method UIDs for the program in listing 5.1.

Boogie to achieve this, enabling code sharing and making the system immune to UID scheme changes. When replacing the contract call nodes with mock calls, the UID is read from the `ILOffset` field and Cee determines the return value by searching for the tagged function interpretation.

UID Scheme

Any UID scheme for contract method call tagging needs to satisfy two conditions:

- Each contract method call needs to be assigned a unique ID
- The UID assignment algorithm needs to be deterministic such that the calculation is repeatable

We implemented a UID scheme that works as follows:

1. For each contract, determine a unique base string:
   `base`
2. For the n-th method call in the contract, calculate the unique method call string:
   `uidStr = base.Append(‘.’).Append(n)`
3. Calculate the method's integer UID:
   `uid = -1 * Math.Abs(uidStr.GetHashCode())`

The `base` string uniquely identifies the contract within the program, while `uidStr` and `uid` uniquely identify the contract method call. `uid` values are chosen to be negative such that the misuse of the `ILOffset` field in the AST call nodes is likely to lead to early errors should any part of the system rely on these values to be real offsets.

Table 5.1 shows the `uidStr` values for the contract method calls in listing 5.1.

Consider the `IsEven` call on line 6. Its `uidStr` consists of the `base` string `Test.Foo(System.Int32).body.47`, which uniquely identifies the first `assert` statement in the body of the method, and the suffix `.0`, which identifies the call uniquely within that expression. The `ReturnInput` call in the same contract starts with the same `base` but has a different suffix of `.1`, and is consequently assigned a different UID.

5.4 Limitations

The presented function tagging technique only works when method calls are executed at most once[1]. However, there are two cases when methods calls in contracts can be executed multiple times:

1. Method calls in a *forall* expressions
2. Method calls in loop invariants

In case (1), the generated runtime check contains a loop that repeatedly calls the method. If the parameter depends on a quantified variable, the actual return value might be different for some iterations. However, the counterexample contains only one interpretation, namely the one for the failing iteration. Cee wrongly picks that interpretation for all iterations and therefore always return the same value, possibly yielding an incorrect runtime check.

(2) Method calls in loop invariants are cloned into two instances as Boogie creates one check of the loop-invariant for the pre-loop state and one for the post-loop state. Therefore, the counterexample will contain two interpretations with the same UID. When Cee extracts the return values for the two corresponding mock methods it will pick the same interpretation for both of them, as it distinguishes interpretations solely by UID. This can potentially lead to incorrect runtime checks, as one of the mock methods might return a value that is inconsistent with the counterexample.

Possible solutions to these problems are discussed in section 7.4.

[1] Function tagging also works when methods are always called with the same parameters.

Chapter 6

Other Improvements to Cee

In addition to the major improvements and features discussed in previous chapters, we contributed numerous minor enhancements to Cee. In this chapter we present some of them and briefly explain their purpose, functioning, and limitations.

6.1 Counterexample Selection

In many cases, an assembly contains more than one verification failure. Before this thesis, Cee executed the first counterexample in the first method of a program.

We added command line options to Cee that allow selection of the counterexample to execute.

Selection by Source Code Location

Using the `--sourceLocation` option, a source code location of the format `line:column` can be passed to Cee. If the location corresponds to a verification failure location, Cee executes the respective counterexample. The verification failure location encompasses the Spec# AST node that contains the broken contract. If the given source code location does not correspond to a verification failure, Cee aborts and prints an error message.

This command line option was added such that we could integrate Cee with IDEs. We can now offer a programmer to directly execute a certain verification failure reported by Boogie from within IDEs.

Selection by Method Name

The `--method` option can be used in order to instruct Cee to execute a counterexample in the method with the given full name. If there are multiple counterexamples in the specified method, Cee picks the first one by default. Optionally, the n-th counterexample in the specified method can be executed by using the `--nThCounterexample` option. Counterexamples are ordered by the verification failure location in BoogiePL.

These options for selecting counterexamples were developed for facilitating the creation of automatic test cases.

6.2 Visual Studio Integration

Counterexample Execution

As part of this thesis, we developed a Visual Studio AddIn that allows the execution of counterexamples from within the Visual Studio IDE.

When the Spec# plug-in for Visual Studio reports verification failures, the user can right click into the editor at the location of such a failure (marked by green squiggly lines) and choose "Run counterexample" from the context menu. This starts Cee in the background, passes the source code location using the `--sourceLocation` option and automatically attaches the Visual Studio debugger. See appendix B.1 for a screenshot of the AddIn.

A new option `--waitForDebugger` has been added causing Cee to generate a counterexample program that waits until a debugger has been attached before calling `Debugger.Break()`. This prevents the system from showing a unnecessary debugging dialog box before the Visual Studio debugger attaches to the counterexample program.

The Visual Studio AddIn greatly improves the development cycle of programmers, as they can now perform all development related activities within one IDE.

Unfortunately, the AddIn still has some missing functionality. One major missing piece is better integration with project properties. The AddIn compiles the currently opened file separately, without compiling any other source files in the project or linking referenced assemblies. This heavily limits the type of programs which counterexamples can be executed from within Visual Studio.

Testing Framework

The existing testing framework based on *make*[1] has been replaced by a testing system that is integrated into Visual Studio. Additionally, a feature has been added to the testing system that allows embedding of Cee command line options and expected output directly into the Spec# test case files. See appendix C.2 for an example test case definition.

6.3 Integration with the *e TextEditor*

We created a Spec# bundle for the *e TextEditor*[2] program. The bundle includes a language syntax definition for Spec# as well as commands to build and verify Spec# programs.

Additionally, commands have been created to generate and run counterexample programs. These commands support the same Cee command line options embedding as the testing framework presented in the previous section.

[1] Information about the GNU make software can be found on the Internet under `http://www.gnu.org/software/make/`

[2] The *e TextEditor* is a *TextMate* clone for the Windows platform. More information can be found on the Internet under `http://www.e-texteditor.com/`.

6.4 Runtime Checks

Prior to this thesis, Cee generated essentially the same runtime checks as the Spec# compiler. We modified Cee to only emit runtime checks that are required to reproduce the verification failure. The failing contract node is determined from the counterexample, and all other contracts are stripped from the program. This increases performance of both, counterexample program generation and execution. Additionally, it facilitates reading and understanding of the generated counterexample program for humans.

If the option `--keepAllRuntimeChecks` is given, Cee generates all runtime checks.

6.5 Debugging Options

In order to facilitate debugging of the Cee program we added the following to options: `--debugCee` causes Cee to break at the beginning of its execution and `--printBpl` emits the Boogie program in BoogiePL, including all Cee rewritings, to the specified file.

Chapter 7

Future Work

After having improved Cee as discussed in the previous chapters, there are still some limitations that restrict the set of reproducible counterexamples. The first three sections of this chapter discuss how some of these limitations in Cee could be eliminated.

The fourth section presents an idea that enables Cee to broaden its scope, which is currently restricted to unverifiable programs. It introduces a technique that allows programmers to use Cee for understanding verifiable programs, too.

7.1 Object Registry

There is a major flaw in Cee that prevents a great number of counterexamples from being executed correctly. Cee currently does not track object creations and therefore, when the same object enters the method under verification at different locations, multiple instances are initialized. Potential interface points where objects (re-)enter the method under verification are listed in section 3.1.

Consider the following example Spec# program:

```
1  public abstract class Test
2  {
3    public int x;
4
5    public void Foo(Test! p)
6      requires p.x == 0;
7    {
8      p.x = 100;
9      Test! q = ReturnInput(p);
10     assert q.x == 0;
11   }
12
13   public abstract Test! ReturnInput(Test! input)
14     ensures result == input;
15 }
```

The counterexample program generated by Cee initializes two instances for the same object passed in as parameter `p`. First, an instance is created for the

input parameter `p`. The second instance is created in the mock method for the return value of `ReturnInput` on line 9, even though the counterexample correctly specifies the same partition as for the input parameter. Besides not being reference-equal, these spurious instances are all initialized with the initial field values for that object. In the example above, this means that after line 9, `q.x` is 0 instead of 100, and consequently the runtime check for the assert statement wrongly passes. Creating spurious instances clearly introduces deviations from the counterexample and might prevent the counterexample program from reproducing the verification failure.

Using a global object registry in the counterexample program could solve this problem. Cee would create a central method in the counterexample program that is exclusively responsible for creating new instances. This method would keep an object registry of object instances, indexed by their partition. When an instance of a certain partition is requested, it would check the registry and if an entry for the partition is found, return the already created instance. Otherwise it would initialize a new instance as described in section 3.1 and add it to the registry. This would effectively solve the problem described above.

7.2 Improving Side Effect Capturing

Side effect capturing in Cee is currently limited to (immediate) fields of the method call parameters. Cee requires a mapping from objects to C# expressions in order to access and update modified objects, and for method call parameters this mapping is readily available. The following shows how side effect capturing on arbitrary heap locations could work.

As counterexamples contain observable side effects only, it would be sufficient to obtain expressions for objects that are reachable from the method call's parameters through a series of field or array element accesses. The technique employed by Cee to address heap locations for frame condition runtime checks, which we presented in section 4.1, could be extended for this purpose. While it is currently able to produce expressions rooted in the parameters of the method under verification only, it could be adjusted to generate expressions rooted in an arbitrary set of objects. By passing in the set of parameters of the method call with side effects, anything reachable from these parameters can be addressed and, consequently, updated.

Alternatively, the object registry presented in the previous section could be used to address arbitrary heap locations. However, we think that the technique presented in this section could produce better readable expressions in the counterexample program.

7.3 Improving Runtime Checks for Frame Conditions

We suspect that in cases where the counterexample does not report an explicit heap location violating the frame conditions, Z3 introduced identifiers for universal quantifications which cannot be easily connected to an element from the input program. When inspecting such counterexamples, we found object identifiers such as `$o.sk.4.0` and type fields such as `$f.sk.4.1` that are related to

the frame condition violation. It might be possible to extract enough information from the counterexample by scanning the `select2` interpretations for these elements in order to deduce the heap location violating the frame condition.

If the heap location cannot be deduced from the counterexample, falling back to the naïve approach of frame condition runtime checks might be required. Operations such as snapshotting and comparison of heaps could be implemented on the basis of the object registry we introduced previously. Whether one can implement feasible algorithms performing these operations remains to be explored.

7.4 Improving Function Tagging

As mentioned in section 5.4, there are two cases for which function tagging results in potentially flawed counterexamples: *forall* expression and loop invariants.

forall Expression

Fixing the issue with method calls in *forall* expressions could be tackled by reducing the runtime check to the one failing iteration. Z3's concept of providing only the necessary information for an unsuccessful execution holds true in this case, too. Hence, only one corresponding interpretation of a call in a *forall* expression is present in the counterexample and this interpretation yields the return value for the failing iteration. Cee can therefore unambiguously extract the return value for each method call in the failing iteration.

Unfortunately, the extraction of return values might not be enough to set up the runtime check for the failing iteration, as the expression within the *forall* quantification might depend on quantified variables outside of method calls[1]. This requires that the quantified variables are set up correctly. Whether extracting the actual value of a quantified variable for the failing iteration is possible remains to be explored. If quantified variables are passed to a method, the corresponding interpretation could be inspected for their actual values.

Loop Invariant

Method calls in loop invariants result in two actual calls in the Boogie translation: One for the pre- and one for the post-loop invariant assertion. When the call is cloned into these two calls, Boogie could assign new UIDs that distinguish the two instances. A similar scheme for UID generation as presented in section 5.3 could be employed, distinguishing the two call instances with `.invpre` and `.invpost` suffixes, for example.

7.5 Understanding Verifiable Programs

Cee is a tool that helps programmers better understand verification failures by showing an execution of the program that leads to the contract violation. In

[1] For example `forall { int i in (0:8); 2*i == ReturnInput(i) }`.

the following, we present a technique that would broaden Cee's scope and allow its usage for understanding verifiable programs, too.

Even if programs come with contracts and are verifiable, programmers might still have problems to understand their functioning. Control and data flow are especially hard to understand in methods with excessive branching and looping. By leveraging the execution generation facility of Cee backed by Spec#, we can demonstrate to programmers how a certain statement in a program can be reached:

Given the program statement to reach as an input[2], Cee would insert an `assert false` statement immediately after it. Verification using Boogie would obviously return a failure at that location and Cee would then create the corresponding counterexample program. Running this program in a debugger reveals all input parameters, field values, and method call return values required to reach the statement. In addition, stepping through the method until the statement is reached helps understanding the behavior of control flow constructs.

[2]For example as source code location of the form `line:column`, such that users could pick a program statement in a text editor.

Chapter 8

Conclusions

In this thesis, we have enhanced Cee by removing limitations of existing features as well as adding new functionality.

We improved maintenance of object states by adding method side effect capturing and improving object initialization. Furthermore, we developed a technique to generate runtime checks for frame condition violations by exploiting information from the counterexample. Generation of frame condition runtime checks is a unique feature of Cee that, to our knowledge, was never implemented before. In order to improve return value extraction for method calls from within contracts, we devised a technique named *function tagging*, which we realized in Cee. We also improved integration of Cee with Visual Studio in form of an AddIn that allows execution of verification failures from within the Visual Studio IDE and contributed smaller improvements to Cee such as the bundle for the *e TextEditor* and new debugging options.

In the last chapter of this report, we presented ideas that could be applied to remove some of the remaining limitations of Cee. A global object registry could be employed to track object instances in the counterexample program and thereby prevent redundant object instantiations. Additionally, the object registry could be used to improve runtime checks for frame conditions. We also showed how side effect capturing could be extended to arbitrary heap locations by employing the heap addressing technique currently used for frame condition runtime checks. And finally, we presented a technique that can broaden the scope of Cee. While Cee is currently restricted to reproducing verification failures, this technique would allow the usage of Cee for gaining better understanding of verifiable programs, too.

We believe that our contributions greatly increased the usefulness of Cee, as a larger set of verification failures can be successfully reproduced, and integration with IDEs eases the usage of Cee as integral part of the development cycle. The removal of some remaining limitations by implementing the presented ideas for future work would furthermore improve Cee significantly.

Part II

Multiple Heaps for Ownership-Based Verification

Chapter 9

Introduction

Originally, the topic of this master's thesis was to improve the heap model for ownership-based verification at the example of Spec#. The goal was to find heap encodings which transfer structural information from the static ownership annotations into the abstract model, such that ownership related proof obligations would be simpler to solve by the verifier.

None of the approaches that were developed as part of this research yielded a heap model that is promising to improve verification performance. Therefore, the idea was abandoned and "Improving Cee" became the new main topic of this thesis.

This part of the thesis describes the developed heap models and encountered problems in order to preserve this research for future reference.

Chapter 10

Background

10.1 Spec# and Boogie

Please refer to sections 2.1 and 2.3 in part I of this report for background information on Spec# and Boogie.

10.2 Current Heap Model

When Boogie translates a Spec# program into BoogiePL for verification, heap accesses have to be translated in a way that models the computer's actual heap. Every access into the heap in Spec# is either a field access on an object or an element access on an array, where the latter can be regarded as a special case of the former (details omitted). The current Boogie implementation models the heap as 2-dimensional array, indexed by an object and a field.

For example, consider the Spec# program:

```
1  public class LinkedListElement
2  {
3    public LinkedListElement next;
4
5    public void CopyNext(LinkedListElement! from)
6      requires from.next != null;
7    {
8      this.next = from.next;
9    }
10 }
```

Boogie translates the field accesses on line 8 into:

```
tmp := $Heap[from$in, LinkedList.next];
$Heap[this, LinkedListElement.next] := tmp;
```

Formalization

In order to be able to discuss new heap models, a mathematical formalization of 2-dimensional array heaps is helpful.

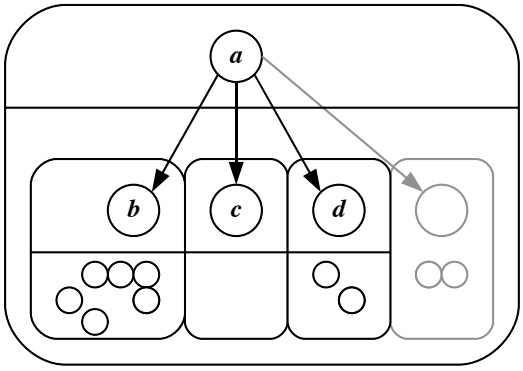

Figure 10.1: Ownership hierarchy where a owns b, c, and d.

Be NAME the set of class fields defined in a Spec# program and ADDR the set of object addresses.

Then we can define HEAP as the set of heaps, which are modeled as 2-dimensional arrays mapping an object address and a class field to an object address. Reading from and writing to heaps are defined as follows:

$$\begin{gathered} ReadHeap : \text{HEAP} \times \text{ADDR} \times \text{NAME} \rightarrow \text{ADDR} \\ WriteHeap : \text{HEAP} \times \text{ADDR} \times \text{NAME} \times \text{ADDR} \rightarrow \text{HEAP} \\ \text{s.t. } \forall h, h' \in \text{HEAP}, o, v, o' \in \text{ADDR}, f, f' \in \text{NAME} \bullet \\ h' = WriteHeap(h, o, f, v) \Rightarrow (ReadHeap(h', o, f) = v \\ \wedge((o' \neq o) \vee (f \neq f') \Rightarrow ReadHeap(h, o, f) = ReadHeap(h', o', f')) \end{gathered}$$

The formula expresses that after applying $WriteHeap(h, o, f, v)$ to a heap, the stored value for $o.f$ is v, and all other values stored in the heap remain constant.

For the sake of brevity, we introduce the notations

$$h[o, f] :\Leftrightarrow ReadHeap(h, o, f)$$

and

$$h[o, f] := v :\Leftrightarrow WriteHeap(h, o, f, v)$$

10.3 Ownership Type System

Ownership type systems provide the means to impose structure on an object graph. Every object is *owned* by at most one other object, which leads to a tree shaped ownership hierarchy.

10.3.1 Type System Rules

On top of a conventional type system, the following type modifiers are introduced in ownership based systems. These modifiers denote the ownership relation between the object o of the current context and an object v.

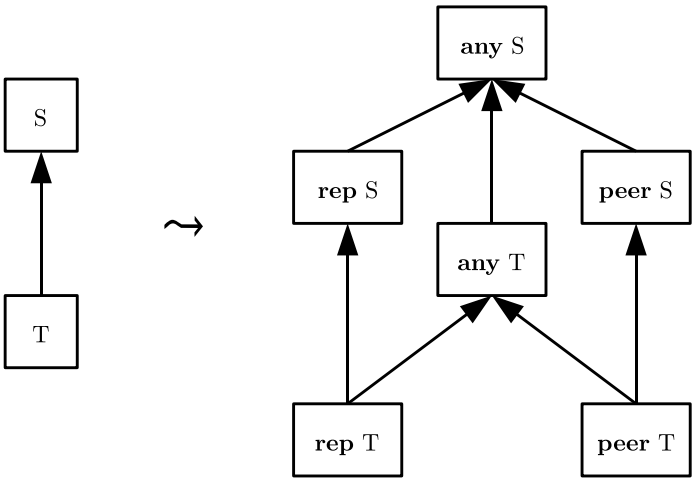

Figure 10.2: Type compatibility and introduction of ownership types. T → S: Expression of type T can be assigned to target of type S.

rep T denotes a type of an object v with $Type(v) = T$ and $Owner(v) = o$.

peer T denotes a type of an object v with $Type(v) = T$ and $Owner(v) = Owner(o)$.

any T denotes a type of an object v which is not restricted in its location in the ownership hierarchy, i.e. $Owner(v)$ is unrestricted. *any* is sometimes called *read-only* for reasons that should become apparent in the next section.

See figure 10.1 for an illustration of how an ownership hierarchy could look like. From the context of b objects c and d are *peer*, its 6 owned objects are *rep* and the 2 objects owned by d are neither *peer* nor *rep* (but *any*). Note that in the context of a, the 6 objects owned by b are not *rep* (but *any*) as they are not directly owned by a.

In order to enforce the ownership type system, each conventional type T is extruded into 3 types: ***any*** T replacing T, ***rep*** T, and ***peer*** T, with the type compatibility hierarchy illustrated in figure 10.2. This means, for example, that a variable of type ***rep*** *Object* can be assigned to a variable of type ***any*** *Object*, but two variables of type ***rep*** *Object* and ***peer*** *Object* are incompatible.

10.3.2 Ownership Hierarchy and Method Side Effects

The ownership hierarchy is used in Spec# to restrict what parts of the object graph might be changed by a method call and which objects remain unchanged. The modification rules based on the ownership hierarchy can be summarized as follows:

> Within a method of an object o, the state (fields) of an object v can be directly changed and methods with side effects can be called on v iff either $Owner(v) = o$ or $Owner(v) = Owner(o)$.

By restricting object state changes by this rule, side effects of methods of o are confined within the (transitive) ownership cone of $Owner(o)$. This object graph structuring enables verification for object oriented programming as it allows a verifier to assume stability in certain regions of the object graph (namely everywhere outside the ownership cone of $Owner(o)$). To further narrow that region of possible state changes, so called *frame conditions* can be specified for each method. In Spec#, frame conditions are defined using *modifies clauses*, which essentially list all heap locations (object fields) the method is allowed to modify. The default modifies clause in Spec# lists all fields of the *this* object.

Note that Spec# has more such rules to enable, for example, object invariant checking. These rules are not discussed here, as they are not relevant for the further discussion.

10.3.3 Implementations

An ownership type system can be implemented using a static type system that extrudes conventional types as explained in section 10.3.1 and checks type compatibility at compile time, or by dynamically proving the necessary proof obligations over the ownership hierarchy and thereby off-loading the proving of type system constraints to the verifier.

In Spec#, the ownership type modifiers *rep* and *peer* are represented by the attributes `Microsoft.Contracts.Rep` and `Microsoft.Contracts.Peer`, respectively. Types without annotations are treated as *any*. Ownership constraints are proven dynamically during verification, and the modification rule mentioned in the previous section 10.3.2 is translated into a complicated postcondition.

The ownership hierarchy is encoded in a special *$ownerRef* field pointing to the owner of the object. This field is only visible during verification, but custom ownership constraints can be formalized by the programmer via the method `Microsoft.Contracts.AssertHelpers.OwnerIs(object,object!)`. For example, in order to assert that the owner of object v is o, a programmer could write the following line in Spec#:

```
assert AssertHelpers.OwnerIs(o, v);
```

This Spec# line would translate to the following BoogiePL code:

```
assert $Heap[v,$ownerRef] == o;
```

```
1  public class Geometry
2  {
3    [Peer] Rectangle r;
4    [Rep] Circle c;
5
6    public void Bar()
7    {
8      r = new [Rep] Rectangle(0, 0, 5, 10);
9      c = new [Peer] Circle(0, 0, 5);
10     assert c.radius == 5;
11     r.Translate(5, 5);
12     assert c.radius == 5;
13   }
14 }
```

Listing 10.1: Prototypical Spec# program.

10.4 Problem

As the current heap model does not take any ownership information into account, proof obligations related to the ownership hierarchy are generally complex.

Consider the prototypical Spec# program in listing 10.1. In order to prove that the assertion on line 12 holds, the verifier has to reason about the post-state of the heap after calling `r.Translate(...)`. Because the heap is simply a 2-dimensional array, the formalization of the side effects is a complex expression involving universal quantification. The theorem prover then has to determine whether `c.radius` is affected by this complex expression or not, a procedure that is rather costly.

Chapter 11

Investigated Heap Models

Given that Spec# programs generate many complicated proof obligations related to the ownership hierarchy as explained in section 10.3.3, it looks like a promising idea to model the heap in a way that resembles the static ownership information of a program. Three such heap models were developed as part of this thesis. They were developed and analyzed under the following considerations:

1. The heap model must be sound and complete. This means that there cannot be any contradictions within the model definition, and all operations on heaps in Spec# must be expressible using the model.

2. The heap model must be realizable in Boogie PL, the input language of Boogie.

3. The heap model must guarantee modularity, i.e. method implementations must be verifiable separately, in their own context, without inspecting implementations of other methods or modifying their specifications.

4. The heap model must show promising performance characteristics when applied to certain types of programs, while the average case performance should not be degraded.

Condition (4) is rather hard to measure and we will elaborate on it more when discussing the actual models. Generally, we try to improve verification performance of the prototypical program introduced in listing 10.1.

The following sections describe the three models *Peer Heaps*, *Paths*, and *Heap Variables*. All of them have in common that they are based on the current heap model, but also try to encode some of the ownership structure in order to facilitate verification. This structure on the heap would then allow the verifier to reason about disjoint heaps. In the case of the prototypical program in listing 10.1, the models try to provide the verifier with disjoint heaps for c and r, such that the assertion would be trivial to prove, as the side effect of r could be expressed by havocing a separate heap.

None of the investigated models actually proved to be feasible for Spec#. Hence, each section will also present the problems found during the analysis of the respective model.

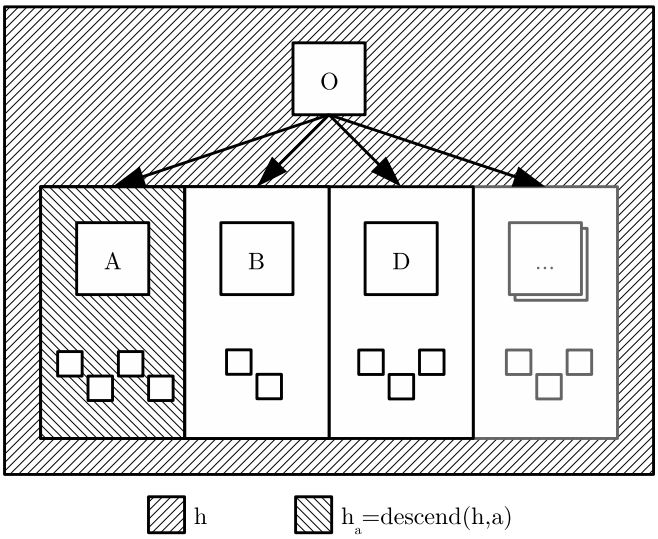

Figure 11.1: Illustration of the *descend* function in the *Peer Heaps* model.

11.1 Model: *Peer Heaps*

The *Peer Heaps* model imposes a tree structure on heaps representing the ownership cones of the program. The goal is that the verification of the example program in listing 10.1 profits from the segmentation of the heap into smaller parts, such that the side effects of the method call can be described in simpler terms by havocing the specific ownership cones of r.

In addition to the tree structure, some local perspective is added to the model: A collection of peer objects of *this* is available and there is a function that returns the heap for each peer object. Unfortunately, for some objects we cannot deduce anything about their location in the ownership hierarchy and therefore a global heap as in the current model is still required.

The idea was that assigning each ownership cone its own heap and the addition of a local perspective would simplify proofs for certain ownership related proof obligations. The following subsections describe the formalization of this model, how existing Spec# programs would be encoded, and why the idea behind it is flawed.

11.1.1 Tree Structure

The heap tree is defined by the following function:

$$descend : \text{HEAP} \times \text{ADDR} \rightarrow \text{HEAP}$$

$descend(h, o)$ returns the sub-heap of o, given the parent heap h. For better readability, we define:

$$h_o := descend(h, o)$$

See figure **11.1** for an illustration of the *descend* function.

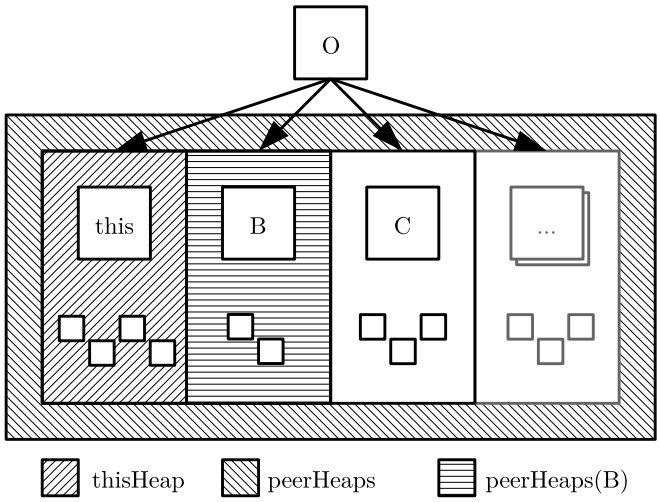

Figure 11.2: Illustration of the *peerHeaps* function in the *Peer Heaps* model.

Variable/Function	**Description**
$this \in$ ADDR	The current object
$thisHeap \in$ HEAP	Contains *this* and transitively owned objects.
$peers \subseteq$ ADDR	Peer objects of *this*
$peerHeaps : peers \rightarrow$ HEAP	Mapping from $peers$ to their heaps

Table 11.1: Context variables for the *Peer Heaps* model.

11.1.2 Local Context

As the ownership type system works relative to a current context, we introduce the context variables in table 11.1 defining the basis for the recursive formalization presented above. Note that $this \in peers$ and $thisHeap = peerHeaps(this)$. See figure 11.2 for a graphical illustration of the *peerHeaps* function.

Spec# verifies each program method separately, and hence these context variables are passed into methods using input parameters analogously to the conventional *this* parameter.

11.1.3 Program Translation

In the following we show how heap related statements in Spec# are translated into BoogiePL. The arrow symbol "$\rightsquigarrow$" should be read as "roughly translates to", as more complex translations could be required in some cases.

Field Access

Each field access $x.f$ is translated according to the following rules:

$$\begin{aligned} \textbf{\textit{peer}}\ T\ x &\quad\leadsto\quad peerHeaps(x)[x, T.f] \\ \textbf{\textit{rep}}\ T\ x &\quad\leadsto\quad thisHeap_x[x, T.f] \\ \textbf{\textit{any}}\ T\ x &\quad\leadsto\quad heap[x, T.f] \end{aligned}$$

Method Call

$$\begin{array}{lcl} x.m(...) & \leadsto & \textbf{\textit{assert}}\ P'(A) \\ \text{where} & & \textbf{\textit{havoc}}\ W \\ \textbf{\textit{procedure}}\ T.m(A) & & \textbf{\textit{assume}}\ Q'(A) \\ \quad \textbf{\textit{requires}}\ P(A) & & \\ \quad \textbf{\textit{ensures}}\ Q(A) & & \end{array}$$

Reasoning about precondition P and postcondition Q has to be done in the current context, they were written in the context of the called method, however. P' and Q' are obtained by first translating the expressions according to the translation rules defined above, then applying the variable substitutions listed in table 11.2.

	Substitution
$\textbf{\textit{peer}}\ T\ x$	$[x/this, peerHeaps(x)/thisHeap]$
$\textbf{\textit{rep}}\ T\ x$	$[x/this, thisHeap_x/thisHeap, Addrs(thisHeap)/peers,$ $Heaps(thisHeap)/peerHeaps]$
$\textbf{\textit{any}}\ T\ x$	$[x/this, heap/thisHeap, peers' \subseteq \text{ADDR}/peers,$ $peerHeaps' \in \text{HEAPS}/peerHeaps]$

Table 11.2: Context variable substitutions for procedure calls.

By adding *pure* and $confined$ method modifiers, we can more precisely describe what parts of the memory the procedure is allowed to read and write. The *pure* modifier states that the procedure cannot perform any externally visible changes to any heap. $confined$ restricts a procedure to perform changes only to $thisHeap'$, i.e. only change fields of the current object or (transitively) owned objects. And finally, $confined pure$ means that the procedure only reads from $thisHeap'$ and does not perform any externally visible changes to any heap. See table 11.3 for the exact definitions of read set R and write set W given a procedure modifier. From the write set W can be inferred which parts of the tree heap structure have to be invalidated after a call of a method with the respective modifier. The read set R defines which heap variables and functions the method takes as arguments.

Synchronize Heaps

After applying the two translation rules above and after having desugared assignments into assumptions, each heap expression on a specific heap (i.e. involving $peerHeaps$ or $thisHeap_y$) in an `assume` statement needs to be replicated on

	R	W
No modifier	$\{heap, thisHeap', peers', peerHeaps'\}$	$\{heap, thisHeap', peers', peerHeaps'\}$
pure	$\{heap, thisHeap', peers', peerHeaps'\}$	$\{\}$
confined	$\{thisHeap'\}$	$\{heap, thisHeap', peers', peerHeaps'\}$
confined pure	$\{thisHeap'\}$	$\{\}$

Table 11.3: Procedure modifiers and their effect on R and W.

the global heap as well. This step is required in order to keep the model sound and reflect the same information in the global heap as well as in the specific heaps.

For example, the statement

```
assume peerHeaps(this)[this, T.f] == 5;
```

needs to be rewritten to

```
assume heap[this, T.f] == 5 &&
    peerHeaps(this)[this, T.f] == 5;
```

because we might later on access the field $T.f$ of *this* through an *any* reference.

11.1.4 Flaws

The *Peer Heaps* model unfortunately is not feasible in reality. It has some fundamental flaws making it impossible to build and use for the Spec# programming system.

Keeping Heaps in Sync

Keeping heaps in sync is expensive: For each new fact on a *rep* or *peer* reference, we have to assume that fact in both, the specific heap of that object and the global heap. Imagine a program where the same object is referenced in one *rep* and one *any* variable. Failing to assume a new fact on the global heap would render the model unsound, as the field of the same object would have different values depending on which reference we use to access it.

For each new fact on an *any* reference, it is even worse. As it is impossible to statically determine the most specific heap for an *any* reference[1], a complicated BoogiePL expression involving universal quantifiers is required to keep the model sound and reflecting the fact in the specific heaps.

Obligations involving quantifiers are generally slow to prove with theorem provers, which means that keeping heaps in sync would most likely lead to a great performance decrease during verification.

[1] If this was possible, we would always use the most specific heap only and would not have to keep the global heap at all.

Improper Usage of Mathematical Functions

The model defined as above is not sound. Modeling $peerHeaps$ as a mathematical function is not correct because the return values for the same input might change as the sub-heaps change over time. Fixing the model requires replacing mathematical functions such as $peerHeaps$ by 1-dimensional arrays, the elements of which can change over time.

Failing to model these functions as arrays leads to contradictions as soon as a heap changes twice, rendering the model unusable.

The *Peer Heaps* model was abandoned and not corrected because the heap synchronization problem was already considered grave enough to prevent it from being used as a successful model for verification purposes. Correctly modeling $peerHeaps$ and other functions would introduce even greater complexity such that we felt it was not worth to investigate further in this direction.

11.2 Model: *Paths*

The *Paths* model was developed in order to remove the synchronization problems from the heap model. Similarly to the previous model *Peer Heaps*, the heap is split into smaller heaps that are structured in a tree. Each object and its heap are addressed by a path describing their locations in the tree starting at the root.

The advantage of this model is that we always know where fields of an object are stored, namely in the heap with the path of that object. If the path of an object is unknown, we simply do not make any assumptions and leave it to the theorem prover to reason about it. Thereby it removes the need to statically address a heap or perform expensive heap synchronization in the program translation.

The remainder of this section presents the formalization of the model as well as program translation rules. The last sub-section discusses why this model is not a good candidate for replacing the current heap model.

11.2.1 Formalization

Be PATH the set of paths. The model introduces the following constants and functions to build paths, determine whether one path is a prefix of another, and address heaps:

$Path$: REF $\rightarrow$ PATH	Function mapping from objects to their paths
$Append$: PATH $\times$ REF $\rightarrow$ PATH	Extends a path
$IsPrefix$: PATH $\times$ PATH $\rightarrow \mathbb{B}$	Whether path 1 is a prefix of (or equal to) path 2
$Heaps$[PATH] $\rightarrow$ HEAP	Array mapping a path to the heap at the respective position in the heap tree

The model defines axioms that describe properties of these functions. The following is an excerpt that should illustrate the meaning of each function:

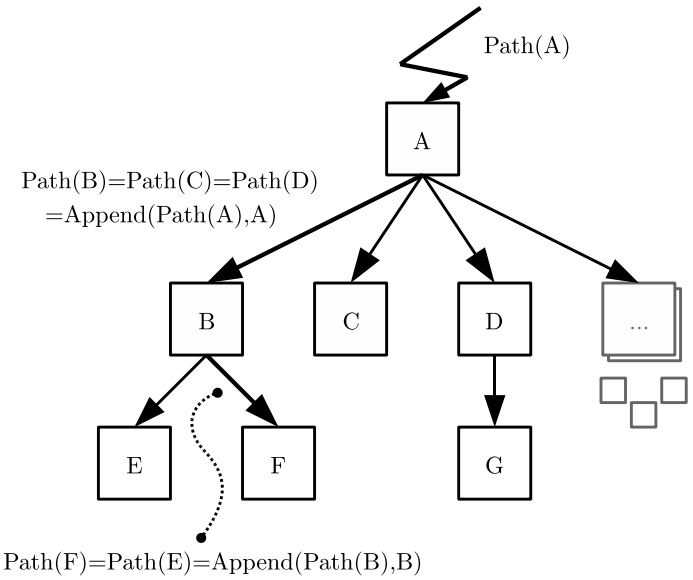

Figure 11.3: Illustration of the *Path* function in the *Paths* model.

Ax1 $\forall p \in \text{PATH}, r \in \text{REF} \bullet Append(p, r) \neq p$
Ax2 $\forall p \in \text{PATH}, r1 \in \text{REF}, r2 \in \text{REF} \bullet$
$r1 \neq r2 \Rightarrow Append(p, r1) \neq Append(p, r2)$
Ax3 $\forall p \in \text{PATH} \bullet IsPrefix(p, p)$
Ax4 $\forall p1 \in \text{PATH}, p2 \in \text{PATH}, r \in \text{REF} \bullet$
$IsPrefix(p1, p2) \Rightarrow IsPrefix(p1, Append(p2, r))$

Ax1 states that *Append* returns a path different from the path an element is appended to. Ax2 states that appending different elements to the same path yields different paths. Ax3 states that each path is a prefix of itself. Ax4 states that if a path $p1$ is a prefix of another path $p2$, $p2$ can be extended and $p1$ will still be a prefix.

See figure 11.3 for an illustration of the *Path* function.

11.2.2 Program Translation

Field Access

Each field access of the form $x.f$ is translated according to the following rules:

$$\textbf{\textit{peer}}\ T\ x \quad \rightsquigarrow \quad Heaps[Path(this)][x, T.f]$$

$$\textbf{\textit{rep}}\ T\ x \quad \rightsquigarrow \quad Heaps[Append(Path(this), this)][x, T.f]$$

$$\textbf{\textit{any}}\ T\ x \quad \rightsquigarrow \quad Heaps[Path(x)][x, T.f]$$

Method Call

$$
\begin{array}{lcl}
x.m(...) & \rightsquigarrow & \textbf{\textit{assert}}\ P'(A) \\
\text{where} & & \textbf{\textit{havoc}}\ W \\
\textbf{\textit{procedure}}\ T.m(A) & & \textbf{\textit{assume}}\ Q'(A) \\
\quad \textbf{\textit{requires}}\ P(A) & & \\
\quad \textbf{\textit{ensures}}\ Q(A) & &
\end{array}
$$

We omit defining the read and write sets explicitly here. They are very similar to the sets described for the *Peer Heaps* model. Havocing a certain sub-tree rooted at path p can be expressed using the following statements:

$$
\begin{array}{l}
\textbf{\textit{havoc}}\ heaps' \\
\forall p' \in \text{PATH} \bullet \neg IsPrefix(p, p') \Rightarrow heaps'[p'] = heaps[p']
\end{array}
$$

This states that heaps outside of the sub-tree rooted at p remain the same, while we cannot assume anything about heaps within that sub-tree.

11.2.3 Flaws

Equivalence to Current Model

When regarding the *Paths* model from a distance one realizes that it is essentially equivalent to the current heap model. While in the current model, the owner is stored in a special *$ownerRef* field for each object, this information would now be encoded in paths. While the theorem prover had to reason about transitive ownership following a series of *$ownerRef* field accesses, it would now do so by building and comparing paths. Using the *Paths* model, there would be no performance gain for verifying ownership related proof obligations, and we therefore abandoned this model.

Having recognized that encoding ownership information from a global perspective prevents us from helping the theorem prover with ownership related proof obligations, we were led to build a model that goes into the other direction and tries to model the heap from a very local perspective.

11.3 Model: *Heap Variables*

The *Heap Variables* model is fundamentally different than the two models presented above in that it does not structure heaps in a tree. Instead, each variable is assigned its own heap variable and all accesses to fields of that variable are made through its heap. Note that a particular variable might have different objects assigned during program execution. It is therefore necessary to (conceptually) use a heap for each variable spanning all possibly assigned objects.

In order to keep the model sound when objects are accessed through different variables, pairwise set relations between all heaps have to be statically determined. These relations are then used to perform heap synchronization when necessary.

Using separate heaps per variable removes the necessity of a global heap as well as disburdens the theorem prover from having to reason about (transitive) ownership hierarchies. Instead, some ownership hierarchy analysis is performed

statically during program translation. The hope was that moving the ownership hierarchy analysis from verification to program translation phase would result in better verification performance.

This model has not been devised in full detail, meaning that this section merely presents the general ideas behind it. The reasons why this model was dropped are discussed ath the end of this section.

11.3.1 Program Translation

A separate local heap variable is created for each local variable with reference type in the method under verification. Parameters are a special case of local variables and consequently a heap variable per parameter is created and additionally expected to be passed to the method.

Consider the method `Foo` in the following Spec# program:

```
public class Test
{
  public void Foo(Test o, int x)
  {
    Test p = new Test();
    int y = 12;
    // ...
  }
}
```

In the program translation phase, the signature of `Foo` will be modified to require passing a heap for each reference type parameter, and a local heap variable is created for each local reference type variable. The following pseudo code should illustrate how heap variables are introduced:

```
procedure Test.Foo(Test this, Heap thisHeap, Test o, Heap
    oHeap, int x)
{
  var Test p, Heap pHeap;
  var int y;
  // ...
}
```

All field accesses are then performed on the respective heap variable of the field access receiver. Chained field accesses[2] could either be performed on the heap of the first field access or intermediary heaps could be introduced for each subsequent field access.

11.3.2 Heap Relations

Possible relations we considered between a pair of heaps are:

- **Equality**
  Two heaps are considered *equal* if the set of objects that are possibly accessed through them are determined to be equal. Heaps determined to be

[2] A Spec# expression such as `this.o.f.x`.

equal can be accessed through one shared heap variable. While coalescing two heaps into one heap variable offers the advantage of removing the need for heap synchronization, it also brings the disadvantage of forming bigger heaps. Whether identifying two heaps as equal is advantageous depends on the actual program and is generally hard to determine.

- **Disjointness**
  Two heaps are *disjoint* if we can statically guarantee that no object is accessed through both heaps. Disjointness of heaps is desirable as disjoint heaps do not require to be synchronized.

- **Overlap**
  If none of the two relations above, *equality* or *disjointness*, can be asserted, it is possible that at least one object is accessed through both heap variables during program verification, and we therefore say that the heaps *overlap*. Hence these heaps need to stay synchronized in order to guarantee that information about each object is consistently reflected, regardless of the heap it is accessed through.

It is important to note that equality of heaps should only be determined if the two heaps share a lot of accesses on the same objects. If too many heaps are coalesced into one heap variable, we eventually end up with the current model, i.e. one global heap only.

11.3.3 Statically Determining Heap Relations

We considered two techniques that can be employed to statically determine the relations *equality*, *disjointness*, and *overlap* between heaps.

Annotations

The first technique, *annotations*, requires the programmer to specify the relations manually:

Pairwise specifications such as `EqualHeaps(o,p)` (`DisjointHeaps(o,p)`) would indicate to the program translator that variables o and p should be accessed through equal (disjoint) heaps. If there is no annotation for reference type variables o and p, the program translator would assume that the heaps for o and p overlap.

Additionally, the program translator needs to guarantee that each method call conforms to the annotations given by the programmer, i.e. that the relations given for the actual parameters conform to the annotations of the called method.

It is the responsibility of the programmer, however, that the annotations are sound and reflect all possible program executions. If, for example, two heaps are annotated as being disjoint although an object might be accessed through both of them, contradictions might be introduced at verification time.

As for some programs, the ownership type system already poses an unpleasing burden to the programmer, we do not think that the additional burden of annotating pairwise heap relations would be feasible for real world software engineering.

Data Flow Analysis

By employing data flow analysis and exploiting the ownership type specifications of a program, the *equality*, *disjointness*, and *overlap* relations can be deduced.

In the general case, data flow analyses are bound to be imprecise as they usually work on a high abstraction level. In order to improve precision, specifications such as pre- and postconditions could be taken into account in addition to the ownership type specifications.

Consider the following Spec# program:

```
public class Test
{
  public int x;

  public void Foo(Test! other)
    requires other != this;
  {
    other.x = 5;
  }
}
```

A data flow analysis that takes preconditions into account could deduce that the heap for `this` is disjoint from the heap of `other`.

Any automatic technique that deduces relations between heaps needs to be pessimistic at determining disjointness. Marking two heaps as disjoint although they are not disjoint in reality might introduce contradictions at verification time and would thereby render the whole process unsound. For the equality relation, the algorithm needs to weigh the advantage of removing heap synchronization against the disadvantage of bigger heaps, as explained in section 11.3.2.

11.3.4 Flaws

Infeasibility of Static Analysis

We consider both techniques for statically determining relations between heaps infeasible. Requiring the programmer to manually annotate relations for each pair of heaps seems too big of a burden. Automatically performing data flow analysis would most likely be too imprecise in the general case.

Disjointness Hard to Determine

As any automatic technique is required to be pessimistic regarding disjointness, the result would be excessive heap synchronization (overlap) or degeneration of the model towards the current model with just one global heap (equality).

Chapter 12

Conclusions

12.1 General Problems

We have seen that – independent of whether the static ownership information is encoded from a global or from a local perspective – all the investigated models exhibit major flaws that render them infeasible for real world usage in our opinion.

The general problems encountered during analysis of the models can be summarized as follows:

- When partitioning the heap into disjoint sub-heaps, not every heap access can be statically resolved to the most specific heap. It is therefore necessary to either carry along a global heap or to synchronize a number of heap pairs. *(Peer Heaps, Heap Variables)*
- When trying to avoid heap synchronization, resolving heap access to the most specific heap must be offloaded to the verifier, which basically yields the current model. *(Paths)*

Statically modeling the heap resembling the ownership hierarchy would also prevent support for ownership transfer, which might be added to Spec# in the future.

12.2 Future Work

Given the experience with the analyzed heap models presented above, we do not believe that a reasonably complex model resembling the ownership hierarchy can be built for Spec#. It is well possible, however, that other static program verifiers provide features that would enable the creation of such heap models.

Instead of focusing on the ownership hierarchy, future work for Spec# aiming to increase performance of verification could be based on separation logic[6] or similar techniques. That would imply the addition of a major new concept for the programmer as well as for Spec#, however.

Bibliography

[1] Jürg Billeter. Counterexample execution. Master's thesis, ETH Zurich, 2008.

[2] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An overview, 2004.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[6] John C. Reynolds. Separation logic: A logic for shared mutable data structures, 2002.

[7] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. 2008.

Part III

Appendices

Appendix A

Cee Usage

A.1 Usage

```
Cee [--debug|--check|--generate]
    [--expect EXCEPTION]
    [--sourceLocation LINE:COL]
    [--method METHOD_FULL_NAME]
    [--nThCounterexample N_TH_COUNTEREXAMPLE]
    [--keepAllRuntimeChecks]
    [--waitForDebugger]

    [--debugCee]
    [--printBpl BPL_PRINT_FILENAME]
    [--verbose]
    [--wait]
    ASSEMBLY
```

A.2 Options

Option	Description
`--debug`	Start counterexample program in debug mode
`--check`	Check counterexample output against `--expect`
`--generate`	Generate counterexample program only
`--expect EXCEPTION`	Exception counterexample is expected to throw
`--sourceLocation L:C`	Counterexample selection by source code location
`--method METHOD_NAME`	Counterexample selection by method's full name
`--nThCounterexample N`	Select n-th counterexample in specified method
`--keepAllRuntimeChecks`	Generate all runtime checks
`--waitForDebugger`	Generated program will wait for debugger to attach
`--debugCee`	Pause at the beginning of the execution of Cee
`--printBpl FILENAME`	Output generated Boogie program to specified file
`--verbose`	Output more information to console
`--wait`	Waits for key press before exiting Cee
`ASSEMBLY`	Assembly containing the input Spec# program

Appendix B

Visual Studio Integration

B.1 Cee AddIn

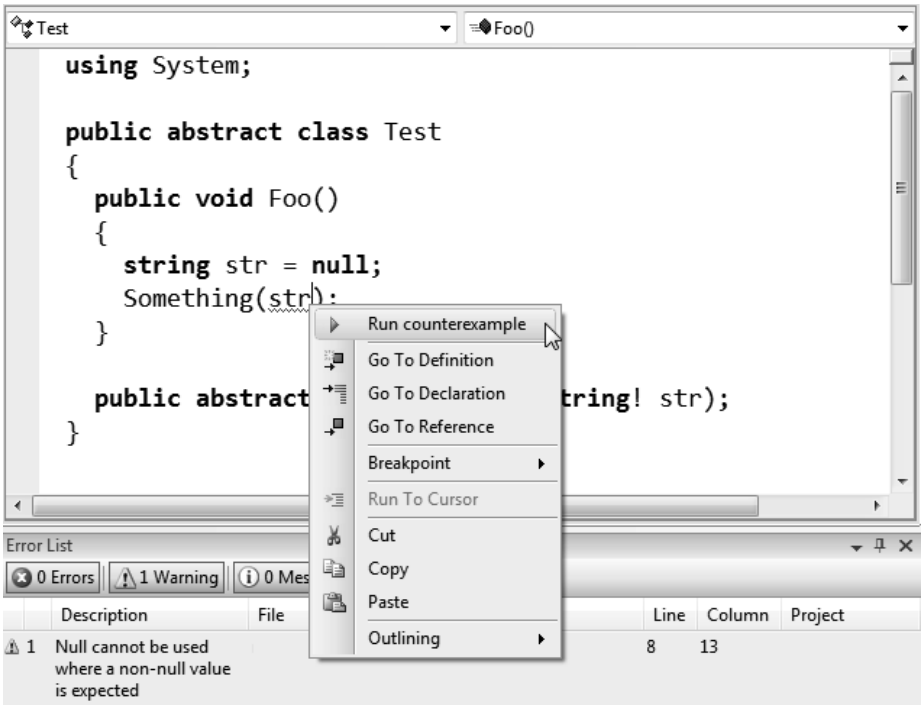

Figure B.1: Screenshot of the "Run counterexample" command provided by the Cee AddIn.

Appendix C

Text Case Definitions

C.1 Example 1: Frame Condition Violation

```
1  public class D
2  {
3    public int x;
4    [Peer] public D! d;
5  }
6
7  public class T
8  {
9    int x;
10   [Peer] D! myobject;
11
12   public void Foo ()
13     modifies this.x;
14   {
15     expose (myobject)
16     {
17       myobject.d.x = 12;
18     }
19   }
20 }
21
22 // Checks whether we can handle frame condition
       violations on fields a few levels deep from a root
23 // variable 'this'.
24
25 // cee-expect: Microsoft.Contracts.ModifiesException:
       Modifies clause violated from method 'T.Foo'. Value of
        'this.myobject.d.x' differs in pre- and post-state.
```

C.2 Example 2: Side Effect Capturing

```
1  namespace FieldUpdate
2  {
3    public abstract class TestClass
4    {
5      public int x;
6
7      public void TestThisFieldUpdate()
8      {
9        x = 4;
10       MethodWithSideEffect();
11       assert this.x == 4;
12     }
13
14     public void TestOtherFieldUpdate(TestClass! other)
15     {
16       other.x = 4;
17       other.MethodWithSideEffect();
18       assert other.x == 4;
19     }
20
21     public abstract void MethodWithSideEffect()
22       ensures this.x == 5;
23
24   }
25 }
26
27 // cee-arguments: --method "FieldUpdate.TestClass.
       TestThisFieldUpdate"
28 // cee-expect: Microsoft.Contracts.AssertException:
       Assertion 'this.x == 4' violated from method '
       FieldUpdate.TestClass.TestThisFieldUpdate'
29
30 // cee-arguments: --method "FieldUpdate.TestClass.
       TestOtherFieldUpdate(optional(Microsoft.Contracts.
       NonNullType) FieldUpdate.TestClass)"
31 // cee-expect: Microsoft.Contracts.AssertException:
       Assertion 'other.x == 4' violated from method '
       FieldUpdate.TestClass.TestOtherFieldUpdate(optional(
       Microsoft.Contracts.NonNullType) FieldUpdate.TestClass
       )'
```