A Case Study in the Spec# Programming System

Verifying System.IO

Olivier R. Girard

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

Summer Semester 2006

Supervised by:
Joseph N. Ruskiewicz (MSc in Computer Science)
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

This report documents the results of a case study in the Spec# Programming System. To achieve a good study, we used Mono as real-life project. We encountered many interesting problems beyond class-room examples, such as low-level programming (unsafe code) in Spec# and integrating input from external resources into specification. In assisting with this case study, a testing tool has been developed and documented in this report.

keywords: Spec#, Boogie, specification, verification, C#, Mono, System.IO

Contents

1 Introduction 7

2 Background 9
- 2.1 Spec# . . . 9
  - 2.1.1 Working with Spec# . . . 9
  - 2.1.2 Boogie . . . 11
- 2.2 Mono . . . 12
  - 2.2.1 Overview System.IO . . . 12
  - 2.2.2 Compiling Mono Sources . . . 12
- 2.3 Spec# Testing Tool . . . 13

3 Writing Contracts 15
- 3.1 Non-Null Types . . . 15
- 3.2 Pure Methods and Contracts . . . 16
- 3.3 Transform already existing Contracts! . . . 16
- 3.4 Verified vs. Defensive Code . . . 18
- 3.5 Concluding the Example . . . 19

4 Verifying System.IO 21
- 4.1 Unsafe Code . . . 21
- 4.2 Value / Range Checks . . . 22
- 4.3 Invariants? . . . 22
- 4.4 Recursive Specification . . . 23
- 4.5 Binary Operators . . . 24
- 4.6 Disposable Types . . . 25
- 4.7 Unverifiable Stuff . . . 25
  - 4.7.1 String Arguments . . . 26
  - 4.7.2 File / Directory Existence . . . 27

5 Conclusion 29

6 Future Work 31

A Testing Tool 33
- A.1 Purpose of this Tool . . . 33
- A.2 Boogie Test Project . . . 33
- A.3 How to use the Tool . . . 34
  - A.3.1 Creating Projects . . . 34
  - A.3.2 Compiling and Verifying . . . 35
  - A.3.3 Documentation of Results . . . 37

B Verification Overview 41

Chapter 1

Introduction

For a semester project at ETH Zurich, we have made a case study in the Spec# Programming System. The goal of this case study was to examine the Spec# Programming System from a practitioner's point of view. This case study analyzes and tests the current system in a real-life working environment. It can be regarded as experiment in the Spec# Programming System.

As working project for this case study, we have chosen a subset of the Mono [8] System.IO core component. Mono is a large and complex project and thus not easy to handle. Please see Chapter 2 for background information on Spec# and Mono.

We have transformed about 5'000 lines of C# code (20 types) into Spec# code. Our general approach of code transformation is documented in Section 3. The *application* of these concepts for the Mono System.IO classes and the *main results* of the case study are presented in Chapter 4.

We have chosen the System.IO base class library because it implements core functionalities of many applications in use. System.IO is one of the most used namespaces in common .NET applications and thus an important piece of software that should be as secure as possible. Furthermore, in System.IO many "exotic" features of the C# language, such as low-level, unsafe code or external method calls, are used. This case study will reveal the capabilities of the Spec# Programming System in this area.

Conceptually, the IO classes are rather simple and not so deeply structured as for instance a linked list implementation of the System.Collections namespace[1]. So special attention is given to the IO interacting classes (file/directory, reader/writer). Some interesting questions to think about: What can be verified outside a data structure? (e.g. existence of file), How can path expressions be checked?, Is it possible (and reasonable) to enforce use of informal contracts?

It is not the purpose of this report to reveal all details on the Spec# Programming System. This report documents our attempt to work with Spec# and Boogie as practitioners, trying to verify a software component. This report should point out advantages and disadvantages of the current Spec# Programming System and show ways of how to deal with similar tasks.

Spec# Testing Tool

Working with many different sources and other dependencies turned out to be very challenging and time consuming. Therefore, a tool for managing this special kind of project has been developed [12]. Please see Appendix A for details and support.

[1]Research in this area: Mono's System.Collections Classes: A Spec# Case Study, Benjamin Lutz, ETH Zurich

Chapter 2

Background

2.1 Spec#

Spec# is a new programming system for specification and verification of object-oriented software [1]. The Spec# language is a superset of the programming language C# extending C# by non-null types, method contracts, object invariants and an ownership type system. The behavior of a Spec# program is checked at runtime and statically verified by Boogie, the Spec# static program verifier [2]. Boogie generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover [7] that analyzes the verification conditions to prove the correctness of the program or find errors in it. One of the main innovations of Boogie is a systematic way (a methodology) for specifying and verifying invariants. The Spec# Programming System handles callbacks and aggregate objects, and it supports both object [4] and static [3] class invariants. It enables sound, modular reasoning.

2.1.1 Working with Spec#

There is limited documentation for the Spec# Programming System. In this section, we give some general hints to make it easier to try out and understand the presented examples in this report. This is a very short and incomplete introduction. For more details please visit the Spec# homepage [5] and have a look at these talk presentations [6].

Spec# Contracts

Pre-conditions can be expressed as requirements for a method to be executed whereas a post-condition is an assurance a method can give about a certain state after its execution.

In Spec# code these contracts can be written directly after the method declaration using the keywords `requires` and `ensures`. In the conditions most of the common logical operators known from C# can be used.

Example:

```
public int AddPositive(int a, int b)
    requires a >= 0; // pre-
    requires b >= 0; // conditions
    ensures result == a + b; // post-condition
{
    return a + b;
}
```

If the pre-condition is checked by the normal code too, you can declare the exception that is thrown if this condition does not hold. This can be achieved by using the `otherwise` keyword.

Example:

```
public object Get(int i)
    requires i >= 0 otherwise ArgumentException;
{
    if (i < 0) throw new ArgumentException();
    return list.Get(i);
}
```

Object invariants can be defined in a dynamic or static context. By text book, an invariant has to hold at every state of execution. But of course, this is not enforceable in a real programming language. For this purpose Spec# lets you expose an object so that its invariant can (temporarily) be broken until it's not exposed anymore.

Example:

```
public int a = 1, b = 2;
invariant a < b; // object invariant declaration

public void BreakInvariant()
{
    expose(this)
    {
        this.a = this.b; // invariant broken
        this.b = this.b*2;
    } // invariant holds!
}
```

In Spec# far more complex concepts are implemented for working with invariants. Again, please visit the Spec# homepage [5] for more information.

Assume and Assert

At any point in the implementation, you can add a statement to assert or assume a condition to hold.

Example:

```
int a = 0, b = 1;
assert a < b; // OK
assume a == 1; // WRONG ASSUMPTION
a = 1;
```

The assertion has to be proven, the assumption can be taken as a new fact. Note that at runtime both cases lead to an abortion if the condition does not hold (exception is thrown).

Non-Null Types

One of the simplest (conceptually) but also one of the most effective impacts of the Spec# programming language is the introduction of non-null types. Any variable of a reference type T can be declared as non-null variable by adding an '!' to its type. This means, Spec# asserts that `null` can never be assigned to this variable.

Example:

```
T! t = new T(); // OK
t = null; // NOT ALLOWED!
```

Purity

An important property of methods is the purity. Generally, a pure method does not change the executing program's state (the heap).

Spec# defines three types of purity:
- [**Pure**] Method does not change the existing objects (but it may create and update new objects).
- [**Confined**] Method is pure and reads only `this` and objects owned by `this`.
- [**StateIndependent**] Method does not read the heap at all.

Add one of the three attributes above to a method to declare it as pure method.

Note any called method in a contract has to be pure. Purity is not checked by the current implementation of the Spec# system. However, it's still necessary to declare methods as pure (any of the three types) if you want to use them in contracts.

SpecPublic

In order to use a hidden field in a contract, mark it with `[SpecPublic]` to make it visible for verification.

Example:

```
[SpecPublic] private int a = 2;
private int b = 5;
public void Foo(int a, int b)
   requires a < 5; // OK
   requires b > 0; // NOT ALLOWED! b is hidden
```

Useful Namespaces

- **Microsoft.Contracts** Insert the `using Microsoft.Contracts` directive into the source you want to specify. Spec# annotations are stored in the metadata in terms of user defined attributes in C#. Besides the keywords used above, the described contract declarations can be directly written as attribute (e.g. `[Ensures(...)]`, `[Pure]`, `[SpecPublic]`) and many other additional feature is only accessible through this namespace.

- **BoogieTestEnv** As described in Appendix A, it's possible to define Boogie tests directly in the source code. To use the `BoogieTest` attribute, add the `using BoogieTestEnv` directive to your source.

Similar Approaches

If you're interested in other solutions, have a look at JML [10] or Eiffel [11] too.

2.1.2 Boogie

You can compile the annotated Spec# source code using the Spec# compiler (`ssc`). The output is an assembly (exe or dll) which can be executed on the standard .NET CLR.

Runtime Checks

The Spec# compiler adds special runtime checks to the code (IL level). The specifications are checked and if a condition does not hold, exceptions are thrown. This is the runtime concept and runs independently from the static part of program verification.

Static Verification

Boogie takes an annotated assembly as input and verifies the code statically. The specification annotations are stored as metadata (attributes). Boogie reads this and generates a more abstract specification in its own intermediate language (BoogiePL). It then proves this specification using a theorem prover.

2.2 Mono

Mono is a huge Open Source project led by Novell to create an ECMA standard [9] compliant .NET compatible set of tools, including among others a C# compiler and a Common Language Runtime (CLR). Mono can be run on Linux, FreeBSD, UNIX, Mac OS X, Solaris and Windows based computers. For this case study, we are interested in the System.IO namespace which is part of the .NET core library.

2.2.1 Overview System.IO

System.IO is a complex namespace. Perhaps not from the point of view of algorithm and data structure complexity, but there is a lot of "dirty" stuff in it. This makes it very interesting to analyze if Spec# can handle this.

System.IO consists of classes providing high level interfaces to the programmer to interact with the low level file system implementation of the operating system. System.IO is the door to the world outside. This means at some point these classes have to make external calls to system libraries. Mono provides a layer in between, but still, the IO classes make external calls to Mono libraries. This is implemented in the central `MonoIO` class.

Dealing with addresses and pointer arithmetics requires some low level programming. This can be enabled in C# by allowing code to be unsafe. In Mono System.IO classes, a lot of code is unsafe.

2.2.2 Compiling Mono Sources

The Mono core implementation corresponds to the standard C# language specification [9], but the Mono base class types, namely the ones from the core library, are highly dependent on the Mono specific implementation. Mono defines many additional fields and methods which are accessed by core library classes. In most cases, these members are marked `internal`, thus they can't be accessed from outside the compiled assembly. This means, *Mono classes require Mono for proper compilation*. We need to compile on .NET and Spec# though.

Compiling the whole project

If it would be possible to change some parts of the implementation and remove the accessibility restrictions, System.IO could later be compiled separately from the Mono implementation.

Since Mono is such a huge project - the compile scripts are complicated and the requirements hard to meet - we decided not to spend too much time on this and looked for other solutions.

Compiling the Mono Core Library

Our next attempt was to compile the core library only. So we could still replace parts of the implementation and use our own core library instead of the mscorlib. Perhaps one could even use the Spec# compiler to build the whole core library. This would have been great, since one could build an entirely specified core library in this case.

It seems like this is not the easiest way either. There were many compile errors we couldn't get rid of. Here again, it seems like this would only work if the Mono compiler were used to compile.

The "Hard Way"

We finally decided to compile everything the System.IO classes depend on. This means, one has to look for dependent classes and generate compile scripts for every single class (or group of classes) to be verified. We are using the Spec# compiler to build the libraries. The pre-defined core library classes are replaced by our implementations (this may lead to warnings). The generated libraries can then be verified by Boogie. In most cases, it has turned out to be more efficient to verify method by method than the whole library at once.

Summarizing, we put our test System.IO sources together with all dependent classes from the Mono core implementation. In general, every (recursively) referenced class belongs to this set of dependent classes. Classes with the same interface and behavior as the original .NET classes can be omitted. This can be compiled using the Spec# compiler.

2.3 Spec# Testing Tool

Mono is a big project and one gets easily lost when looking for dependencies. With this case study we implemented a helper tool. This tool's primary purpose is to ease handling sources and dependencies to be compiled. Paths and compile settings are stored together in an XML document. Note that in this context the described test cases do not check the correctness of the Spec# implementation itself, it's rather the verification capability that is tested. This can be regarded as experiments in the Spec# Programming System.

Some additional important functions the tool supports:

- **BoogieTest Attribute** Using reflection, the tool extracts pre-defined (attributes) test cases directly from the assembly. This automates testing which eases it significantly.
- **Result Documentation** Results can be recorded, classified and documented. This data is stored together with the other project data.
- **Archiving Projects** The project file and the referenced sources and dependencies can be packed together into a ZIP file.

All these functions make not only possible to automate testing but also to make *replicable tests*! This has the great advantage that programmers and experimenters can communicate in an easy way just by providing their complete test projects containing all necessary information to reproduce the results.

For details, please see Appendix A.

Chapter 3

Writing Contracts

This section describes our approach to derive essential contracts. Although meant to be as general as possible, the presented concepts refer to System.IO examples and contain ideas of more advanced concepts for informal contracting with I/O functionality.

The static `Delete` method of System.IO.File is used as ongoing example in this chapter (Figure 3.1). `Path.CheckPath` has been added to the existing implementation (Figure 3.2). It checks the correctness of a path-string's format.

```
1  public static void Delete(string path) {
2      if (path == null) // path mustn't be null
3          throw new ArgumentNullException("path");
4      if (!Path.CheckPath(path)) // path format
5          throw new ArgumentException("...");
6      if (!File.Exists(path)) // can't delete non-existing file
7          throw new IOException("...");
8
9      MonoIOError error;
10     if (!MonoIO.Delete(path, out error)) // call MonoIO method
11     { // throw exception if not successful
12         throw MonoIO.GetException("...", error);
13     }
14 }
```

Figure 3.1: System.IO.File.Delete

```
[StateIndependent] public static bool CheckPath(string! path) {
    return path.Trim() != "" && path.IndexOfAny(Path.InvalidPathChars) == -1;
}
```

Figure 3.2: System.IO.Path.CheckPath

3.1 Non-Null Types

In many cases, all you can do in System.IO classes, is to replace method argument and local variable types by non-null types where possible.

In `File.Delete` this would implicate the only parameter:

```
public static void Delete(string! path) { ... }
```

This can and should be done in any case you can claim a permanent non-nullness for a variable (parameter or field). In many cases, non-null checks exist already in the code. In Section 3.4, we will give some food for thoughts about non-null checks and verified code.

3.2 Pure Methods and Contracts

Defining contracts is not always easy. How should one define a pre-condition, to be checked statically, if a file exists or not? (see Figure 3.1, lines 6 and 7). Not all expressions that can be evaluated at runtime can be checked by a static verifier. In the System.IO namespace this is for instance the case with path format or file existence checks. Besides the heap, Boogie does not model the environment (e.g. file system) and has no control over it.

A possible way to treat this kind of condition is to implement pure methods returning the result of the condition (boolean). These methods can be used in contracts as predicates and be checked as usual at runtime. For static verification, it is sufficient to assume the condition to hold (this is discussed in Chapter 6). For this case study, we simplify our world by making the assumption, that there won't be any external influences from other processes (e.g. another program changing a file).

Example: (Figure 3.1)

```
public static void Delete(string! path)
    requires File.Exists(path); ...
{
    /* ... */
}
```

When `File.Delete` is called, `File.Exists(path)` has to hold. So the caller has to *assume* it to hold before doing the call as it only makes sense if the file really exists. Runtime checks are inserted and Boogie accepts this trivially.

```
string! path = "...";
assume File.Exists(path); // important!
File.Delete(path); // OK
```

This concept is very important System.IO classes and will be discussed more detailed in Chapter 4.

3.3 Transform already existing Contracts!

In a big project, like Mono, that has been developed and improved over years by many people, one can assume that some people thought about security and consistency. The most natural thing to do is to check arguments and global states before a method gets executed. This means, informally, many methods are already specified correspondingly to the common sense of the inventors of the software. So a certainly good approach of finding suitable contracts is to take a look at the method's implementation and extract any information from it.

Pre-Conditions

In many cases, *requirements* are checked at the beginning of the method implementation by checking the attribute values and throwing an exception if a condition does not hold (e.g. lines 5-8 of Delete method). This can be regarded as an *informal specification* of the method and transformed one by one into a requires clause.

Example: (Figure 3.1)

Derive pre-condition from the informal checks.

```
5 if (!Path.CheckPath(path)) // path format
6     throw new ArgumentException("...");
7 if (!Directory.Exists(path)) // can't delete non-existing dir
8     throw new IOException("...");
```

This can be formally expressed in Spec#:

```
requires Path.CheckPath(path);
requires Directory.Exists(path);
```

In a second step, the *meaning and purpose* of the code should be analyzed. In System.IO mostly range checks for integer values and and non-null checks for partly used buffer arrays should be considered.

Other pre-conditions can be derived by studying the API documentation or specification of the code.

Post-Conditions

By abstracting and formalizing the method's purpose one can derive the method's *post-conditions*.

Example: (Figure 3.1)

Add ensures clauses to File.Delete:

```
public static void Delete(string! path)
    // ...
    ensures !File.Exists(path);
{
    /* ... */
}
```

In System.IO, it's often not necessary to add any post-condition regarding return or global values, since no objects of internal data structures are changed and almost all methods are static. But adding some, at first glance superfluous, ensures clauses will help a lot to get things verified. This is discussed in details in Chapter 4.

In general, start with as many conditions (pre- or post-condition) you can find and weaken it only where it's necessary.

Invariants

For this case study, invariants are not a central aspect. System.IO classes consist mostly of static methods and very small data structures. It's quite hard to find meaningful invariants.

Further concepts of the Spec# Programming System (as ownership- and visibility-based models) won't be used for this case study. These techniques serve well for verification of complex data structures, where it is crucial to be aware of mutual object dependencies. In System.IO most methods are static and very few permanent object structures are instantiated. So it doesn't make sense to introduce these very complicated and demanding theories for this case study.

3.4 Verified vs. Defensive Code

As shown in the last sections, the System.IO implementation is already very defensive. This means, most of the existing and reasonable requirements are already checked by the code itself and the error-handling is done by the implementation.

By adding contracts, Spec# inserts additional runtime checks into the code, which, similar to the already existing ones, check conditions and throw verification exceptions if they do not hold. This is nice since Spec# does the error-handling automatically now. The programmer can lean back and delete his own checks.

This is a good solution if the project is really built for the Spec# Programming System. But for a project like Mono, one should not remove the defensive elements of the code but rather let the original code handle the error-handling.

The Spec# Programming System lets you declare special exceptions in the otherwise clause in order to indicate, which exceptions are thrown by the program if the condition does not hold. So Spec# does not throw an additional verification exception.

Example: (Figure 3.1 and 3.2)

```
public static void Delete(string! path)
    requires Path.CheckPath(path) otherwise ArgumentException;
{
    if (!Path.CheckPath(path)) // path format
        throw new ArgumentException("...");
    /* ... */
}
```

For instance, this makes sense if code is only partly specified or if there are accurate rules to obey (e.g. C# specification). So the programmer accessing this code can use it as usual and doesn't have to know what Spec# does, he does not even have to know it exists.

In this case, the non-null checks should also be declared as additional requires clause (since the other programmer is not using the non-null type system).

Example: (Figure 3.1)

Replace the method argument types by non-null types (or do it without this restriction) and add requires clause (in any case).

```
public static void Delete(string! path)
    requires path != null otherwise ArgumentNullException;
{
    if (path == null) // path mustn't be null
        throw new ArgumentNullException("path");
}
```

3.5 Concluding the Example

The fully specified `File.Delete` (Figure 3.1) method, using the error-handling of Mono's defensive implementation:

```
1 public static void Delete(string! path)
2     requires path != null otherwise ArgumentNullException;
3     requires Path.CheckPath(path) otherwise ArgumentException;
4     requires File.Exists(path) otherwise IOException;
5 {
6     if (path == null) // path mustn't be null
7         throw new ArgumentNullException("path");
8     if (!Path.CheckPath(path)) // path format
9         throw new ArgumentException("...");
10    if (!File.Exists(path)) // can't delete non-existing file
11        throw new IOException("...");
12
13    MonoIOError error;
14    if (!MonoIO.Delete(path, out error)) // call MonoIO method
15    { // throw exception if not successful
16        throw MonoIO.GetException("...", error);
17    }
18 }
```

We observe that in the final implementation we have redundant code and specifications. Using this specification technique, every requirement is checked twice at runtime.

Chapter 4

Verifying System.IO

Applying the described concepts, we transformed the C# code into Spec# code and verified it using Boogie. This chapter presents the main results of verifying a subset of the Mono System.IO implementation. Sample test cases will demonstrate usable concepts for the verification of System.IO. In general, a problem/task and its solution is presented. In some cases, the current state of the Spec# implementation is sufficient for our needs, in other cases, we say where it lacks and provide some ideas for improvement.

Please see Appendix B for an overview of the verified classes. The here presented test cases can be loaded and run with the presented Boogie testing tool (Appendix A) using the binaries of the Spec# release: Spec# 1.0.6003 for Microsoft Visual Studio .NET 2005 (RTM). All test sources are deployed with this report.

4.1 Unsafe Code

As stated in Chapter 2, System.IO classes contain a lot of unsafe code fragments and external method calls. This is no problem for Spec# compiler (using the `/unsafe` flag). The compiled code is executable and it works as expected. *But Boogie is not able to handle unsafe code.*

Therefore any unsafe code fragment has been removed and replaced either by an analog safe implementation or a non-functional stub.

The external calls to the core IO libraries, this means the most interesting part of the whole I/O interacting implementation, have also been replaced by stubs. We considered it to be better to have non-functional, but specified stubs.

This is a remarkable restriction for such a project. The code to be verified has been changed and, what's the biggest limitation for testing the outcome, not functional any more. This means, we were not able to check if the code would actually work as expected since there were only hypothetical results.

However, for Boogie testing only, this is not so bad. The code's static behavior should still be the same.

Replacing unsafe code - MonoIO The class `System.IO.MonoIO` contains the main IO functionality of Mono's System.IO implementation. This is entirely unsafe code with many calls to external Mono libraries. In a first step, the implementation of this class has been replaced by method stubs containing contracts only.

Example:

```
[MethodImplAttribute (MethodImplOptions.InternalCall)]
public extern static bool CreateDirectory (string path,
                                           out MonoIOError error);
```

... is replaced by:

```
public static bool CreateDirectory (string! path,
                                    out MonoIOError error)
    requires Path.CheckPath(path);
    ensures Directory.Exists(path);
    ensures result == true; // assume everything works
{
    error = MonoIOError.ERROR_SUCCESS;
    assert(error != MonoIOError.ERROR_SUCCESS);

    // Assume that the implementation is correct!
    // Informal contract for external / unsafe code
    // (necessary since it can't really be verified)
    assume Directory.Exists(path);

    return true;
}
```

In the same way, any other unsafe code has been replaced. By doing so, Boogie can finally verify System.IO classes!

4.2 Value / Range Checks

Transforming Reader and Writer classes showed that Spec# is perfectly able to handle “usual” data structures in this context. For instance, the specification of correct buffer usage increases security significantly. It's good to have a system proving that the program to be executed will never generate a buffer overflow or write to field it shouldn't. Boogie can do that.

Example:

Buffers (arrays) must not be null and initialized when writing/reading. Buffer sizes and offset (e.g. buffer of a binary reader, or char-array argument) have to be checked.

This can be granted by specifying the necessary non-null types and pre-conditions for the affected methods.

```
public virtual void Write(char[]! value, int offset, int length)
  requires value != null otherwise ArgumentNullException;
  requires offset >= 0;
  requires (length + offset) <= value.Length;
{

  if (value == null)
    throw new ArgumentNullException(Locale.GetText ("NULL␣ref"));

  byte[] enc = m_encoding.GetBytes(value, offset, length);
  OutStream.Write(enc, 0, enc.Length);
}
```

In most cases, it's sufficient to check if offset positive and if the range to be read or written is valid.

4.3 Invariants?

As stated in Chapter 3, our main focus was not on finding invariants in a namespace almost entirely consisting of static members and having no complex data structures.

However, in some cases, it's nice to model dependencies between two fields using an invariant. As for instance in the class `FileInfo` the dependency between the original path string and the according full path string. This can be helpful to maintain the consistency of an object's state relating to its type specification.

Example:

In System.IO.FileInfo: Assert that the fullpath variable always represents the absolute path of the given path.

```
private string! OriginalPath = "", FullPath = "";

invariant FullPath == Path.GetFullPath(OriginalPath);

public FileInfo (string! path)
    requires Path.CheckPath(path) otherwise ArgumentException;
{
    CheckPath(path); // runtime checks

    OriginalPath = path;
    FullPath = Path.GetFullPath(path);

    base();
}
```

The invariant enforces the dependency of the two fields.

In order to get such invariants verified, the programmer has to make sure that each write access to one of the two fields is guarded by corresponding pre- and post-conditions. This is not easy.

Boogie just takes the specification and verifies it. Nothing gets actually executed. So, another problem with such contracts is, that methods like `Path.GetFullPath`, are not executed by Boogie. So without having an explicit assignment or assumption about the values of these two fields at some point in the program, nothing can be verified (statically). This is discussed in Chapter 6.

4.4 Recursive Specification

The following example is a method from the `Directory` class:

```
public static void Delete (string! path, bool recurse)
  requires Path.CheckPath(path) otherwise ArgumentException;
  ensures recurse ==> !Directory.Exists(path);
  ensures recurse ==> forall{string! s in GetDirectories(path); !Directory.Exists(s)};
  ensures recurse ==> forall{string! s in GetFiles(path); !File.Exists(s)};
{
  CheckPathExceptions (path); // runtime checks -> throws ArgumentException
  if (!recurse)
  {
    Delete(path);
    return;
  }
  RecursiveDelete(path);
}

static void RecursiveDelete (string! path)
     requires Path.CheckPath(path);
E1:  ensures !Directory.Exists(path);
```

```
E2: ensures forall{string! s in GetDirectories(path); !Directory.Exists(s)};
E3: ensures forall(string! s in GetFiles(path); !File.Exists(s));
{
  foreach (string dir in GetDirectories(path))
    RecursiveDelete (dir);

  foreach (string file in GetFiles(path))
    File.Delete (file);

  Directory.Delete (path);
}
```

`void Delete(string! path, bool recurse)` deletes the directory `path` and, if `recurse` is `true`, all its content recursively; otherwise it's only deleted if it's empty.

The non-recursive call is no problem and can be verified as usual. Note that in any case Boogie has to verify that `Path.CheckPath(path)` holds, so this has to be assumed explicitly (see Chapter 6).

Boogie can verify the recursion by deriving everything from the calls to methods having the corresponding ensures clauses. Every ensures clause from `Delete` trivially holds after `RecursiveDelete` has been called. So let's take a look at the ensures clauses E1-E3.

E1 holds automatically after calling `Directory.Delete` (please see exact specifications in code directly).

E2 & E3 hold after looping over the same arrays of file / dir names. `File.Delete(string! path)` ensures `!File.Exists(path)`, `RecursiveDelete` ensures `!Directory.Exists(path)`. So this can be verified.

This is only a partial solution to what should really be expressed. Only one level of recursion can be covered by the post-condition in this case. The `forall` statement loops only over the first level entries in the directory, no subfolders.

Recursive specifications should be implemented by real recursion. But for this purpose, Boogie has to be able to recognize pure methods and execute them. Otherwise, it's not possible to express recursive specification of arbitrary depth.

4.5 Binary Operators

Many methods in the File / Directory context require some kind of file / directory accessibility or general attribute checks. This information is encoded by integers and to be read by combining bit-masks with the stored values.

Example:

```
public static void Copy (string! src, string! dest, bool overwrite)
  /* standard requirements */
  // binary log-ops in Boogie?
  requires (GetAttributes(src) & FileAttributes.Directory) != FileAttributes.Directory
      otherwise System.ArgumentException; ...
{
  /* implementation */
}
```

It would be nice to check these attributes by Boogie, or at least specify it for runtime checking. Boogie does not support binary (bit) operators!

In our case study we've just ignored such conditions since there was no reasonable solution to check that. In future, Boogie should support binary operators too. Therefore, a new theory for binary operators would have to be formalized (including logical bit operations as &, |, >>, etc.).

4.6 Disposable Types

Some types (e.g. `BinaryReader`) are disposable, i.e. by calling the `Dispose` method. In general, fields like the internal buffer of the `BinaryReader` must not be null. But this condition only has to hold while the object is not disposed yet. When the object is disposed, any resource field has to be set to null!

In most cases, the state (disposed / undisposed) of these types is modeled by a special bool field (e.g. `bool m_disposed;`) indicating if the object has already been disposed.

For instance the `BinaryReader`: For any field to be set to null when disposing, define an invariant as follows to ensure the field is not null while the object is not disposed.

```
invariant !m_disposed ==> m_stream != null;
invariant !m_disposed ==> m_encoding != null;
invariant !m_disposed ==> m_buffer != null;
```

When disposing the object, these fields can be set to null:

```
public virtual void Dispose (bool disposing)
  requires m_stream.IsPeerConsistent;
{
  if (disposing && m_stream != null)
  {
    m_stream.Close ();
  }

  expose(this)
  {
    m_disposed = true;

    m_buffer = null;
    m_encoding = null;
    m_stream = null;
    charBuffer = null;
  }
}
```

Note: the private fields `m_stream, m_encoding, m_buffer` have to be declared `[SpecPublic]`.

Problems like this could be avoided or simplified by providing a mechanism to define object states (e.g. disposed/undisposed) and indicate, when exactly invariants or non-nullness of types has to hold.

4.7 Unverifiable Stuff

In System.IO, one of the most often checked conditions are the correctness of a path (string) and the (non-) existence of the addressed file or directory. In this section we present our approaches of handling these checks.

4.7.1 String Arguments

A correct path format is defined by the running operating system. In order to properly access the file system, correctly formatted path strings have to be used by IO methods. To check the path format, we introduced the pure method `Path.CheckPath` which returns true if and only if the parameter string has a correct path format (Figure 3.2). This method can be used in contracts (see Section 3.2).

Example:

Add requires clauses to check path and existence of file:

```
requires Path.CheckPath(src) otherwise ArgumentException;
```

All together (specified `File.Copy` method):

```
public static void Copy (string! src, string! dest, bool overwrite)
    requires Path.CheckPath(src) otherwise System.ArgumentException;
    requires Path.CheckPath(dest) otherwise System.ArgumentException;
    ...
{
    /* ... */
}
```

This is a necessary condition for the Copy method in order to succeed. This is also checked by the original Mono implementation.

For two simple reasons, it doesn't make sense to add these conditions. First, the path strings are usually only given at runtime (input) and thus not known for static verification. Second, Boogie does not execute the method. This means, Boogie is not able to verify the indicated contracts.

The only way to make Boogie accept these contracts and verify them is to assume that this condition holds for these path strings at some point in the implementation. These assumptions have to be somewhere in the control flow before the Copy method is executed and the path strings must not be changed anymore. Without the assume statements, Boogie would not be able to prove correctness of the Copy method call. The most intuitive places for these assumptions would probably be after the string input statement (e.g. a command line read operation). In the example, they directly precede the call statement.

Example: (continued)

In order to execute the `Copy` method, the programmer has to put in assume statements for the conditions in the requires clauses.

```
string! src = "...", dest = "...";
assume Path.CheckPath(src); // necessary assumptions
assume Path.CheckPath(dest);
File.Copy(src, dest, true);
```

This check works at runtime and Boogie verifies. However, it would be nice if Boogie was able to check string formats whenever it's feasible. The most critical point would be the situation, when Boogie has to verify formats of input from any source (user, file, etc.). For such cases, the programmer should have the possibility to use special assertion statements to specify properties (e.g. string format) of the input. This would work like the assumption but if there would be a formalized concept behind it, this would be far more consistent.

4.7.2 File / Directory Existence

In many cases, System.IO classes handle with files and directories. Besides the correctness of a path format, it's even more important if the file or directory to be read or written exists or if it doesn't. *This can only be checked at runtime*, so for static verification with Boogie , it's not important if the file really exists or not. In addition, for this case study, the `File.Exists` and `Directory.Exists` methods do not work since the actual implementations have been removed and replaced by safe stubs. So not even runtime checks are useful in this context.

Again, to enable reasoning with Boogie, the file/dir has to be assumed to exist (or not to exist) explicitly by inserting corresponding assume statements into the code before calling a method.

```
void Test(string! src)
  requires File.Exists(src) otherwise ArgumentException;
{
  // runtime check
  if(!File.Exists(src)) throw new ArgumentException("file␣doesn't␣exist!");
  /* ... */
}

CALL:
  assume File.Exists(src); // assumption necessary
  Test(src);
```

Different from the path string format checks described above, Boogie has no possibility at all to verify this. The file system is an external resource which can't be extensively controlled by Spec#. Even if Boogie would execute a similar method to check the existence, this would only hold for the respective target system in the moment of verification. One cannot enforce the system to freeze after static verification to assert the correctness of a proof until the code really gets executed. Thinking of a multi-process system with other running programs that could create, modify, or delete files and directories while our program is running, even more interferences would have to be considered. But as this gets checked and is handled at runtime, it makes sense to allow these assumptions to pass static verification.
A verification concept could be to model the file system in Boogie. This could be done in a similar way as a transaction system for a database system, supporting locks and serialization. Interference from other programs would still have to be ignored. Finally, conditions like the file-existence can only be checked at runtime. So Boogie cannot verify if a file actually exists or not.

The same problem occurs with any external resource (as a network, other I/O sources, etc.).

Chapter 5

Conclusion

The goal of this case study was to examine the Spec# Programming System from a practitioner's point of view. This case study analyzes and tests the current system in a real-life working environment.

Mono, as a large open source project, and the System.IO namespace in particular, are a very good choice as real-life application example. As described in the previous chapters of this report, there were many uncommon problems in this case study. We have not analyzed standard Spec# verification features and showed verification of a lot of code. This case study's main purpose is to *determine what is missing* from the Spec# Programming System.

The Spec# Programming System does support many key features to specify a high level program. When it comes to low-level programming, the exotic features of the high-level programming language Spec#, the Spec# Programming System lacks of basic concepts to support unsafe code or external method calls. Accessing external resources and handling input, which is the task of System.IO, is really hard to verify in a satisfying way using the current system. We present our ideas in the last Chapter 6.
Dealing with big source arrangements and many different compiler settings is hard. This inspired us to write a helper tool to support tester in managing sources and settings. This tool enables the users to define *replicable tests*. This is, independent from the kind of the code to be verified, a very nice feature and together with the archiving functionality of the tool applicable in practice.

The ability of documenting and classifying test results is another key feature of this this tool. This should encourage programmers and practitioners of the Spec# Programming System to collect their knowledge and document it. This tool should make it easier to share knowledge.

Summarizing, this case study is a success. It demonstrated many interesting aspects of the verification of a real-life software project using the Spec# Programming System and brought up solutions and ideas. We hope, that, in future, the Spec# community will continue with this work in some way. Maybe, some of the presented results/ideas will lead to new projects and research. The tool will be deployed as *Open Source* software [12] and we encourage the community to develop new features and enhance the software.

Chapter 6

Future Work

As this case study showed, there are aspects in real-life programming that have not been fully covered yet by Spec#.

System.IO is the interface to the world outside a program. With input from external resources (file system, user interaction, etc.) determining the program's behavior, there is always a component of insecureness. As Section 4.7 shows, it's not easy or not possible at all to verify a program in this case. How should the verifier be aware of the input? How could a path string be checked? Does a file exist or not? - There are no answers to these questions at the moment.

As proposed in Section 4.7, it would be nice to have a *formalized concept* to work with any kind of input. A possibility would be to enforce checking of certain conditions on the input data to assert properties that can be used for verifying later on. At this point, Boogie would have to *assume* these conditions to hold. Since this can be properly checked at runtime, these assumptions can be made.

The problem of checking file or directory existence is not really satisfyingly solvable. As discussed in the last section, this can be checked at runtime only. The presented solution suggests to assume a file does exist or does not exist at any point needed in the implementation. Boogie does not model the state of the file system, so it has absolutely no knowledge and control about the file system. The programmer has to do this himself.

The programmer can thereby define the state of the environment himself by assuming whatever he wants. Why should it then be checked at all? - we don't have an answer to this question. By enforcing the programmer to place assume statements before calling for instance the `File.Copy` method, the file existence will be checked at runtime for sure and the `File.Copy` method is more likely to succeed.

Of course, this is a trade off between having the most secure software in the world and an exploding runtime because of the repeated checks. So again, why should it be checked at all? At this point, this question has reached an even higher dimension. It depends on the definition of what components can be trusted, this means who has to check if an operation is save to be executed? (e.g. Can the external library be trusted or should the checks be done internally?)

In this case study, we decided to annotate the IO methods and require these checks under the assumption that no other processes (programs) influence the environment while our verified program is executing (i.e. no files are created, changed, or deleted). This has the advantage that the methods are completely specified and the programmer is aware any pre-condition he can influence.

This is certainly a topic[1] to think about in future.

[1]This problem also occurs in other systems resulting in very similar solutions. Please have a look at the Eiffel

[11] or JML [10] projects.

Appendix A

Testing Tool

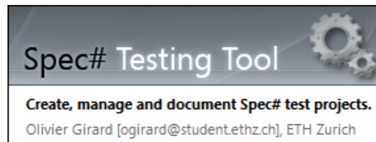

Figure A.1: Spec# Testing Tool

A.1 Purpose of this Tool

This tool has two main purposes. It should support the user to create a transparent project for Boogie and Spec# testing. The tool manages test sources and dependencies (e.g. C# sources or referenced libraries) and helps defining the desired compiler settings. All settings are stored in a pre-defined XML format (explained in the next section). The tool automatically generates batch scripts (for use on Windows platforms) for the given settings of the test project.

Of course, Spec# projects can be created and managed using Microsoft Visual Studio too. This is much more comfortable and one can use all the great features of the IDE. The Boogie Testing tool should not be used instead of Visual Studio, but as additional helper. There are additional features:

Additional settings can be added to the normal source code using the `[BoogieTest(...)]` attribute. The tool uses this to provide several ways of running Boogie on explicit parts (methods) of the source only. In any case, the user is free to configure the compiler settings exactly as he needs to. There are no restrictions.

Next feature and second main purpose of the tool is to give a possibility to the user to classify and to document the results of his tests. This has the advantage that the results and the test settings can been be handled together. In addition, the tool provides an archive functionality to deploy all referenced files in one ZIP file.

This may help the community to share more easily replicable tests.

A.2 Boogie Test Project

In the following, the structure of a *Boogie Test Project* will be described.

The user can select the main project path (%PROJ%) and the root of project dependencies (e.g. C# sources or referenced libraries).

Content of the project folder:

%PROJ%\src	Contains project sources (*.scs).
%PROJ%\bin	Output path for compiled assemblies.
%PROJ%\build	Default location to store generated batch scripts.
%PROJ%\contracts	Folder for Spec# contract files.
%PROJ%\docs	Documentation and additional result files.

All project settings are stored in an XML-file in the %PROJ% directory.
The main nodes:

info	Basic project info.
settings	
dependencies	Paths of dependencies.
sources	Paths of sources.
tests	Paths of test sources.
properties	Compiler and general tool settings.
results	Results (see below).

A.3 How to use the Tool

A.3.1 Creating Projects

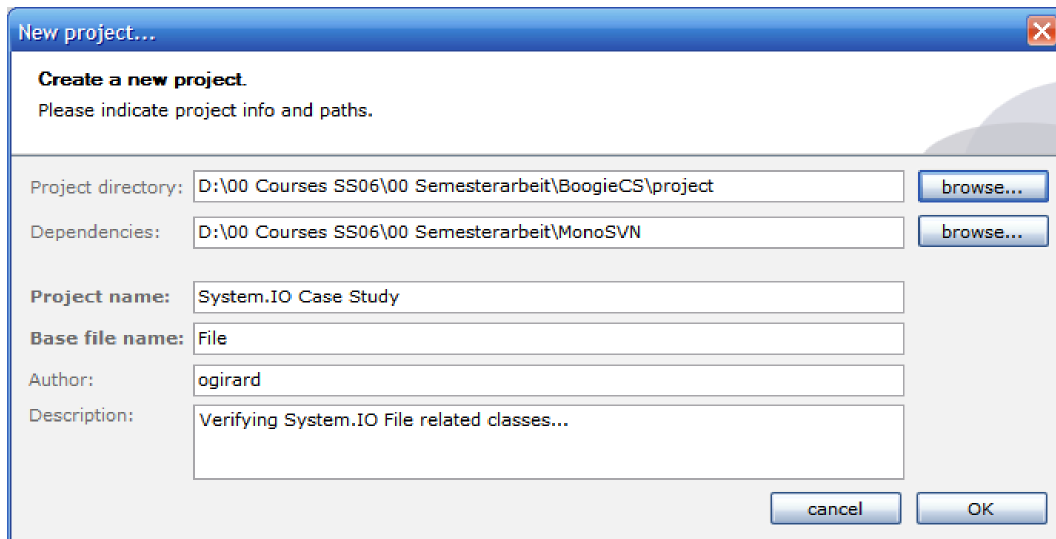

Figure A.2: Creating new projects.

Figure A.2 shows the dialog for creating a new project. Most important settings are the project path (%PROJ%), the dependencies path and the base file name of the project. The base file name will be used as default to compose filenames for automatically generated scripts. The normal name of the project will be displayed in the title bar if the project is loaded.

This dialog can be opened by clicking on `File->New [Ctrl-N]`. By clicking on `File->Open [Ctrl-O]` an existing project can be opened.

Once a project has been opened, the GUI will be enabled and will let you modify the loaded project. The '*' in the title bar indicates that the project has changed. The changes can be

saved by clicking on `File->Save [Ctrl-S]`. `File->Save As...` makes a copy of the project file at the indicated location on the file system. Note that the project's path settings are *not* changed and nothing else is copied automatically. If the project dir has to be changed, click on the `move project path...` button (Figure A.3 (1)). Make sure that the `use relative paths` option is enabled if you like to change all paths at once.

How to configure the project? It's probably best to start with the basic settings as source and dependency paths. Once the project has been created (or loaded), go to the `Settings` tab (Figure A.3).

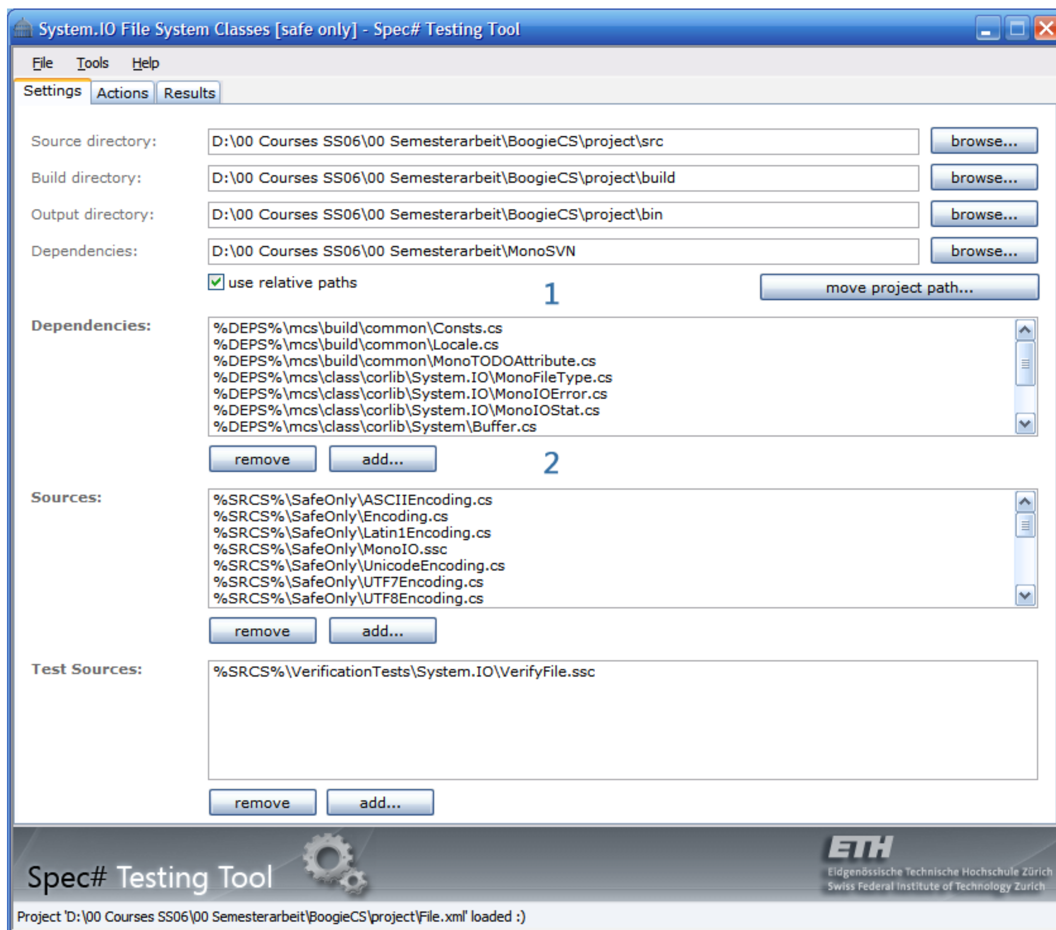

Figure A.3: Settings

Note that the paths can either be added or removed by using the buttons or drag&drop, resp. pressing the `DEL` key. This makes the work with all the different paths a bit more fluent. For the dependencies, the generated `.deps`-files (see below) can be dragged into the box which reloads every listed path (Figure A.3 (2)).

A.3.2 Compiling and Verifying

After configuring the main settings (which can be changed anytime later again), go to the `Actions` tab. Figure A.4 shows the Actions tab. In the upper left (1), you can configure the basic compiler settings. This depends on the type of output you'd like to generate. If Boogie should

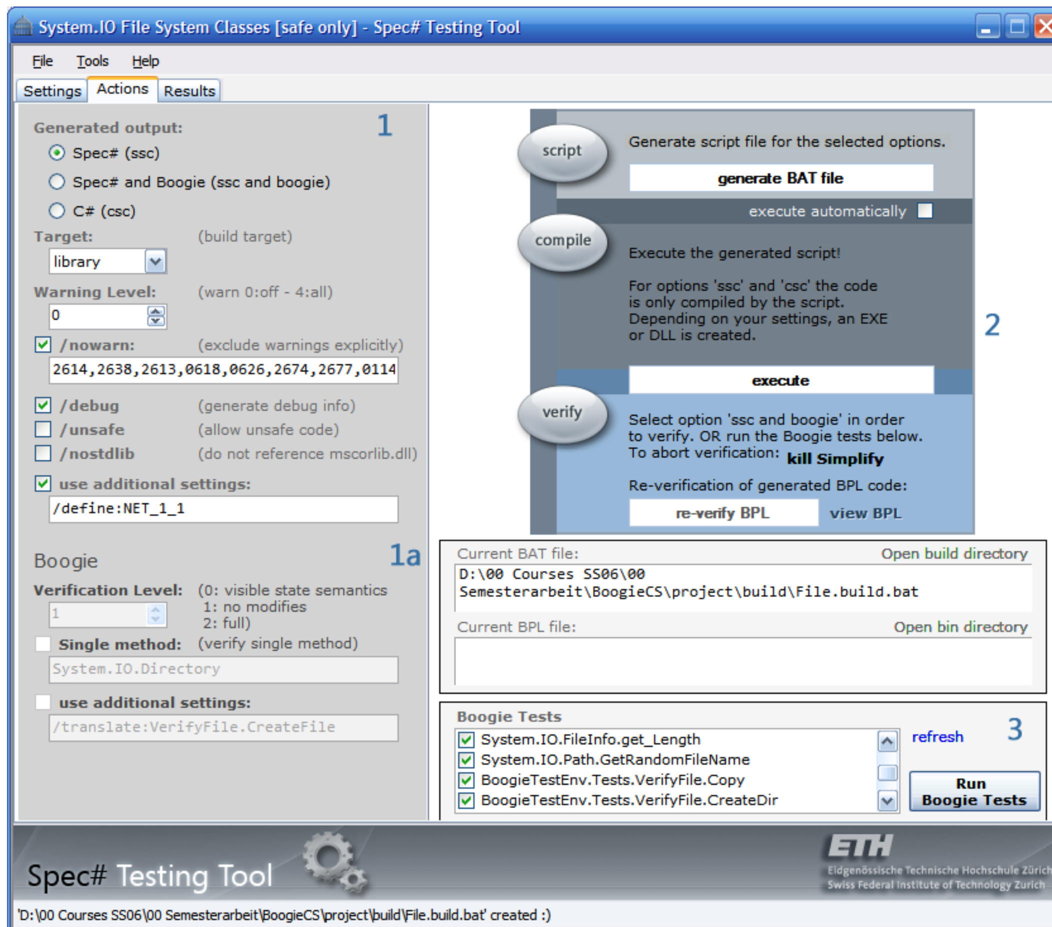

Figure A.4: Actions

be executed after compiling the assembly has been generated (e.g. a dll), please select the option `Spec# and Boogie`. In this case, the options on the lower left (1a) will be enabled to be modified.

You may use any compiler or verifier setting you like. Just check the option `use additional settings`. By right-clicking on the text-box, the context menu displays a selection of useful options (Figure A.5).

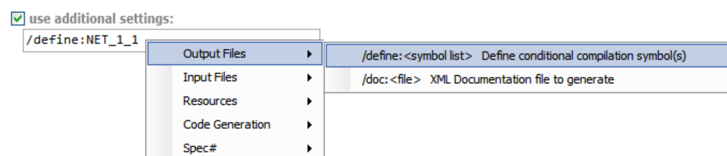

Figure A.5: Additional settings context menu.

How to compile? By clicking on the `generate BAT file` button (2), the tool generates a batch script from your project settings. Note that the tool will remember the path for the script after you've run this once. If you'd like to change it, press `Ctrl` at the same time you click on the button.

If the script file has been generated successfully, the path is displayed below. Press `execute` to run it. If Boogie has been executed, you can modify the generated BPL file and directly re-verify it by clicking on the button below.

Special Boogie Tests The library `BoogieTestsUtils.dll` is referenced by the generated compile scripts per default. This library contains an attribute named `BoogieTest`. It can be used as follows:

```
[BoogieTest(true,1,true,"myBPL.bpl","/noTypeCheck")]
public int MyTest(string! path)
  requires Path.CheckPath(path) otherwise ArgumentException;
{
  /* ... */
}
```

By clicking on `refresh` in the `Boogie Tests` panel (3), the tool reflects your library and generates Boogie commands for every method containing this attribute. You can run them as a bunch (select the desired tests and click `Run Boogie Tests`) or as single test (Figure A.6).

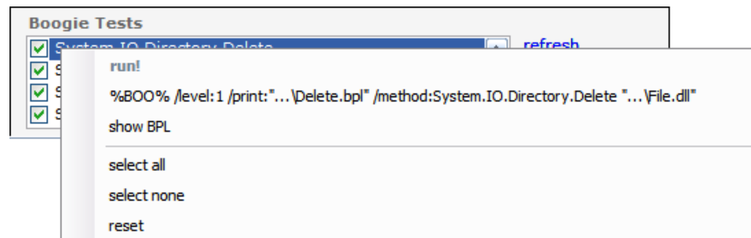

Figure A.6: BoogieTests

The arguments of the attribute:

`bool verify`	True if this test should be run per default.
`int level`	Level of Boogie verification.
`bool print`	True if BPL should be generated.
`* string bpl`	User defined name for BPL-file.
`* string specialargs`	Additional Boogie command line arguments.

* these arguments can be left away.

A.3.3 Documentation of Results

This tool combines the described testing environment with a special documentation functionality. The gained test results can be recorded and stored in the XML project file with all the other project settings. The big advantage of this combination is that the test results are not only *well documented and classified*, but also *easily replicable*.

Result Model A test result consists of a *title*, a *description*, *code fragments* or *console output* and a list of additional *documentationfiles* (Figure A.7, (2)). Additionally, a test result is *classified* (1). There are four pre-defined types of test results:

Show Case Result of a special kind of test to demonstrate some functionality of Boogie or Spec#.

Test Result and documentation of a normal (working) test case.

Issue Problem description of missing functionality or small error.

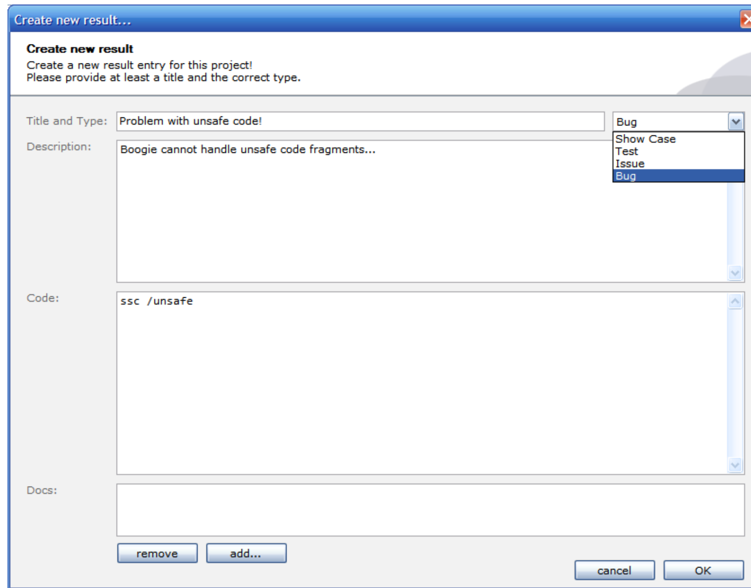

Figure A.7: New Result (dialog)

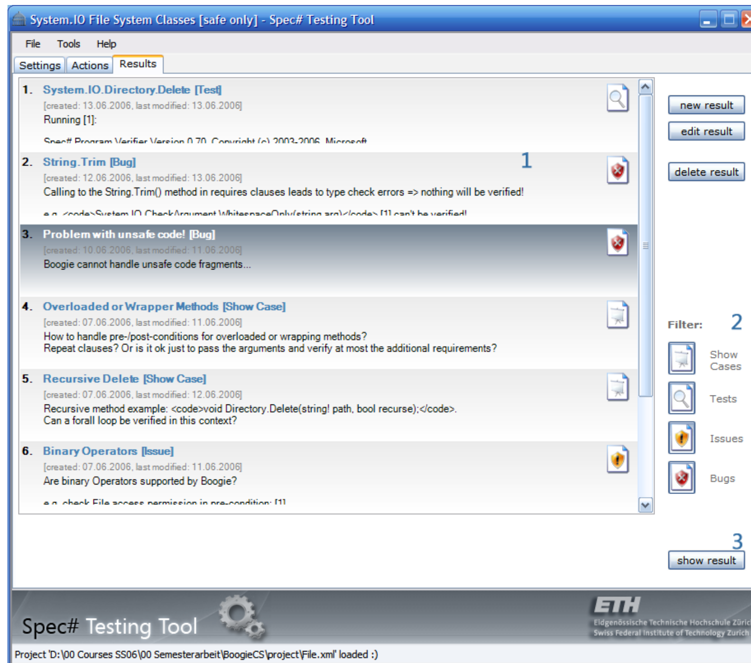

Figure A.8: Results

Bug Severe bug that needs to be fixed.

Figure A.8 shows the `Results` tab of the user interface. On the left (1) the results are listed (sorted by date). Only the results of checked types in the filter section (2) are displayed.

As usual, you can add, edit and remove results. If you don't like the edit form (Figure A.7) representation, you can display the result in the (built-in) browser view. Select the result in the list and click on `show result` (3) or right-click it. The special view with nicer text formatting is

then displayed (Figure A.9).

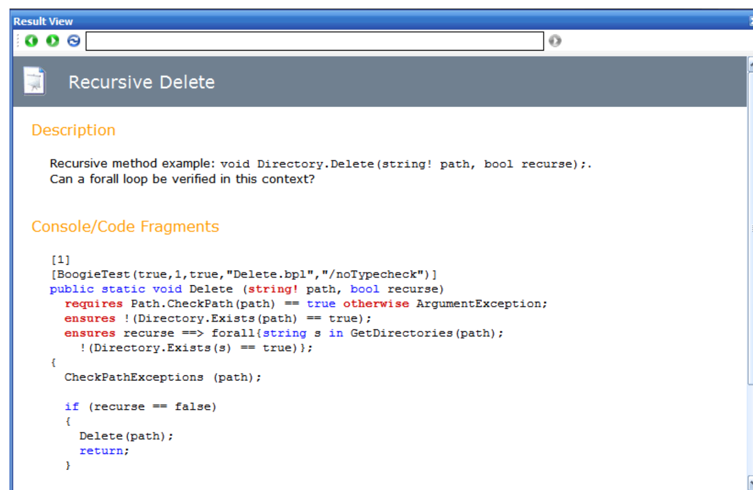

Figure A.9: Result View (HTML/CSS)

Archiving projects In order to archive or exchange projects, including all files, the tool supports a archiving functionality. `Tools->ZIP Project...` opens a dialog (Figure A.10) for storing a ZIP-File containing the current project. For instance, this feature can be used to copy a test setup one-by-one to another test environment.

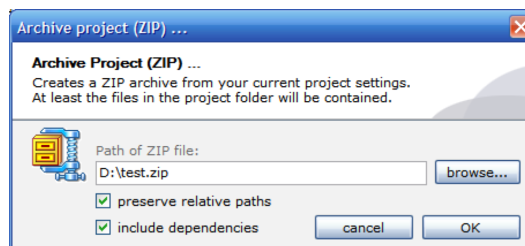

Figure A.10: Archiving Projects

Appendix B

Verification Overview

The System.IO namespace

BinaryReader	BinaryWriter	BufferedStream
CheckArgument	CheckPermission	Directory
DirectoryInfo	DirectoryNotFoundException	DriveNotFoundException
EndOfStreamException	File	FileAccess
FileAttributes	FileInfo	FileLoadException
FileMode	FileNotFoundException	FileShare
FileStream	FileStreamAsyncResult	FileSystemInfo
IntPtrStream	IOException	MemoryStream
MonoFileType	MonoIO	MonoIOError
MonoIOStat	Path	PathTooLongException
SearchOption	SearchPattern	SeekOrigin
Stream	StreamAsyncResult	StreamReader
StreamWriter	StringReader	StringWriter
TextReader	TextWriter	UnexceptionalStreamReader
UnexceptionalStreamWriter		

Verified Classes

File	LOC	Status	Remarks
CheckArgument.ssc	208	annotated	
Directory.ssc	460	(partly) verified	
DirectoryInfo.ssc	283	(partly) verified	
DirectoryNotFoundException.ssc	61	annotated	
File.ssc	548	(partly) verified	
FileInfo.ssc	229	(partly) verified	
FileLoadException.ssc	140	annotated	
FileNotFoundException.ssc	134	annotated	
FileSystemInfo.ssc	260	annotated	
IOException.ssc	66	annotated	
Path.ssc	616	(partly) verified	
PathTooLongException.ssc	66	annotated	
SearchPattern.ssc	204	(partly) verified	
MonoIO.ssc	401	annotated	method stubs only
BinaryReader.ssc	551	(partly) verified	
BinaryWriter.ssc	342	(partly) verified	
TOTAL:	4569		

Bibliography

[1] Barnett, M., Leino, K. R. M. and Schulte, W.: The Spec# programming system: An overview (CASSIS), 2004.

[2] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs (FMCO), 2005.

[3] Leino, K. R. M. and Müller, P.: Modular verification of static class invariants Formal Methods (FM), 2005.

[4] Leino, K. R. M. and Müller, P.: Object Invariants in Dynamic Contexts European Conference on Object-Oriented Programming (ECOOP), 2004.

[5] Microsoft Research Spec#: http://research.microsoft.com/specsharp/

[6] K. Rustan M. Leino at Microsoft Research: http://research.microsoft.com/~leino/

[7] Simplify theorem prover: http://www.hpl.hp.com/downloads/crl/jtk/index.html

[8] Mono Project: http://www.mono-project.com

[9] C# Language Specification, Standard ECMA-334, 3rd edition, June 2005

[10] JML docs: http://www.cs.iastate.edu/~leavens/JML-release/javadocs/

[11] Eiffel language reference: http://archive.eiffel.com/nice/language/

[12] Spec# Testing Tool: http://n.ethz.ch/~ogirard/boogie/