Verifying Spec# Delegates

Samuele Gantner

Master's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

pm.inf.ethz.ch

March 2008 - September 2008

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

Chair of Programming Methodology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Function objects allow to encode references to methods in invocable objects. In C#, the *delegate* construct provides type safe function objects. A delegate instance encapsulates a reference to one or more methods. Showing the correctness of a delegate invocation is equivalent to showing that the pre-condition of the method pointed to by the delegate is satisfied before the invocation of the delegate.

Delegates are commonly used in C# programs for encapsulating and composing methods and implementing eventing patterns. However no verification framework for Spec# delegates has been implemented yet.

In this thesis we implement a verification methodology for singlecast delegates based on refinement of specification and delegates invariants. We then extend the methodology to multicast delegates: delegates pointing to multiple target methods. To ensure correctness, the pre-conditions of all targets must hold throughout the invocation of the multicast delegate; moreover the post-condition of the delegate must hold after the invocation of its last target. We formulate heap and parameters stability conditions, sufficient to ensure the correctness of a multicast delegate invocation. In tightly coupled scenarios - where the contract of a target method depends on both the arguments and the target type - delegate invariants are insufficient to show refinement of specification. To solve this problem we introduce a new construct that allows to fix the type of the target.

We implemented the methodology in the Spec# programming system by extending the Spec# programming language to include pre-, post-, frame conditions, and invariants for delegates. Our extension allows to statically verify singlecast delegates, multicast delegates, and events. We further extended the language to verify delegates where the type of the target can be fixed during declaration. Where applicable, we also introduced runtime checks.

Acknowledgments

I would like to thank my supervisor, Joseph N. Ruskiewicz, for his support and feedback throughout this whole work; Prof. Peter Müller for the useful suggestions about various topics of this thesis, and especially frame conditions; Jürg Billeter for the fruitful discussions, especially about tightly and loosely coupled delegates and all other PhD. student of the Programming Methodology group for their useful suggestions and feedback.

Particular thanks to Hans Dubach and Denise Spicher for their patience in solving all administrative issues during my master.

Thanks to Dario Poggiali for helping me finding a correct balance and finally special thanks to my family for their continuous support during my studies.

Contents

1 Introduction **11**

1.1 Overview . . . 12

1.2 Notation Conventions . . . 13

2 Background **15**

2.1 C# Delegates . . . 15

2.1.1 Delegates in .NET . . . 16

2.1.2 Multicast Delegates . . . 17

2.1.3 Immutability of Delegates . . . 19

2.1.4 Events . . . 19

2.2 The Spec# Programming System . . . 20

2.2.1 The Spec# Programming Language . . . 20

2.2.2 The Spec# Compiler . . . 22

2.2.3 Boogie . . . 22

2.2.4 BoogiePL . . . 22

2.3 Previous Work on Static Verification of Delegates . . . 26

2.3.1 Contracts For Delegates . . . 26

2.3.2 Delegates Invariants . . . 27

2.3.3 Delegate Subtyping . . . 29

2.3.4 Disabling Delegates . . . 29

3 Implementation of Static Verification for Singlecast Delegates **31**

3.1 Contracts and Subtyping . . . 31
3.1.1 Encoding and Definition of Contract Elements . . . 31
3.1.2 Delegate Instantiation in BoogiePL . . . 33
3.1.3 Delegate Invocation in BoogiePL . . . 36
3.1.4 Exposing an Object in BoogiePL . . . 37
3.1.5 Delegate Subtyping in BoogiePL . . . 39
3.2 Disabling Delegates . . . 39
3.2.1 Ownership Based Disabling . . . 39
3.2.2 Disabling Delegates with the Enabled Field . . . 40
3.2.3 Disabling Delegates in BoogiePL . . . 41
3.3 Frame Conditions . . . 43
3.3.1 Encapsulation . . . 43
3.3.2 The Type of the Target . . . 44
3.3.3 Subtyping . . . 45
3.3.4 Frame Conditions in BoogiePL . . . 46

4 Extensions 51

4.1 Delegates with Static Methods . . . 51
4.1.1 Instantiation of Delegates with Static Methods in BoogiePL . . . 52
4.2 Multicast Delegates . . . 53
4.2.1 Stability requirements . . . 53
4.2.2 Encoding Multicast Delegates . . . 54
4.2.3 Combining Delegates in BoogiePL . . . 58
4.2.4 Subtyping and Multicast Delegates . . . 60
4.3 Events . . . 61
4.3.1 Ownership and Events . . . 62
4.4 Closed Target Type Delegates . . . 63
4.4.1 Pre-, Post-Conditions and Invariants . . . 63
4.4.2 Tightly Coupled Delegates . . . 65
4.4.3 CTT Delegates and Subtyping . . . 65
4.4.4 CTT Delegates in BoogiePL . . . 65

5 Implementation and Runtime Checks 67

5.1 From SSC to CIL . . . 67

5.1.1 Delegate Instantiation 67
5.1.2 Contracts for delegates 67
5.1.3 Delegate Subtyping 68
5.1.4 Disabling Delegates 70
5.1.5 Frame Conditions 70
5.1.6 CTT Delegates 71
5.1.7 Compiler Architecture 71
5.2 Boogie 72
5.3 Runtime Checks 73
5.3.1 Building Blocks 73
5.3.2 Implementing Runtime Checks 74

6 Conclusions 79
6.1 State of the Implementation 79
6.2 Conclusions 81
6.3 Future Work 81

A Short User's Manual 85

B Examples 89
B.1 Complete USBStick Example 89
B.2 Events 91

C SSC Syntactic Grammar 93

D Compiler Generated Methods 95
D.1 Compiler-Generated Delegate Classes 95
D.2 Compiler-Generated Events Methods 98

E Found Bugs 101
E.1 Generics 101
E.2 Constructors in Contracts 102
E.3 Array Initialization 103
E.4 Structural Type Rules for Delegates 104
E.5 Out of Band Contracts and Axioms Contradiction 104
E.6 Specification of Array Concatenation 105

CHAPTER 1

Introduction

Function objects allow to encode references to methods, in objects that can be invoked. Function objects are present in many programming languages; they are represented in Eiffel with *agents*, in Scala with *function objects* and in C# with *delegates*: object-oriented type safe function pointers. Function objects pose important challenges on current static verification frameworks. The verification methodology for Spec# delegates described in [15] allows to verify a subset of Spec# delegates and is implemented and extended in this work.

The Spec# programming system allows to statically verify programs with respect to their specification and the semantics of the language. The system is composed by the Spec# programming language, a superset of C#, the Spec# compiler and a static program verifier based on weakest preconditions and first order logic named Boogie. Boogie translates a Spec# program in the intermediate BoogiePL language and then uses a theorem prover to verify it.

A C# delegate declaration includes the signature of possible target methods and is translated during compilation in a class which includes a public *Invoke()* method used to invoke the delegate. Delegates are instantiated like normal objects by providing to the constructor a target object and a target method. Delegates promote encapsulation: the client of a delegate does not have to be aware of the underlying implementation.

Listings 1.1 shows a Spec# program that uses delegates. The *Log()* method of class *Client* invokes a delegate passed as argument, but has no information about the target object and method of *logFile*. In order to verify this program, we need to ensure that the pre-condition of every possible method pointed to by *logFile* holds when the delegate is invoked. If the delegate in the example points to method *Store()* of some object of type *USBStick*, then the invocation might result in a pre-condition violation. This is because the *IsLoaded* condition of the target object might not hold.

In this thesis we implement a modular verification methodology for Spec# delegates based on refinement of specification and delegate invariants [15, 11]. The methodology is then extended to consider object functions pointing to static methods, multiple targets (known in C# as multicast delegates) and events. We further extend the Spec# language to include

Listing 1.1: Introductory example, based on the example provided in [15]

```
class USBStick {
   public bool IsLoaded;

   public void Store(object p)
      requires p != null;
      requires this.IsLoaded;
   { /* Store p */ }
}

delegate void Archiver(object p);

class Client {
   public static void Log(Archiver! logFile, string! s)
   { logFile(s); }
}
```

object functions where the target type is closed during the declaration of the delegate.

1.1 Overview

Chapter 1 introduces the problem this work addresses and the notation conventions needed for reading this document.

Chapter 2 introduces the background information needed to understand this document. The chapter is divided in three section. The first sections explains C# delegates and points out the situations where it is suitable to use them. The second section presents the Spec# programming system, composed by the Spec# programming language, the Spec# compiler and Boogie. The subset of BoogiePL needed for this thesis is also introduced. The third section explains the methodology described in [15]. Readers familiar with these topics can skip this chapter as it does not introduce new concepts.

Chapter 3 presents how the verification methodology for Spec# delegates can be implemented in the Spec# programming system. The first section explains how delegate contracts are encoded and translated in BoogiePL. The second section extends the technique for disabling delegates to make it implementable in Spec#. Finally the third sections explains how frame conditions for delegates can be handled.

Chapter 4 explains the extensions to the approach described in Chapter 3 that we implemented. The first section handles delegates instantiated with static methods. The second section explains how multicast delegates can be verified; in the first part the four conditions necessary for ensuring heap stability are described, while in the second part the problems of disabling multicast delegates is addressed. The third section explains how the verification of multicast delegates is extended to events. The fourth and last section introduces the new concept of closed target type delegates: delegates where the type of possible target objects is defined during the declaration of the delegate.

Chapter 5 covers the implementation in the Spec# compiler and Boogie. The first section explains how the specification information of a source program is encoded in a CIL assembly and also provides a high level overview of the changes done to the Spec# compiler. The second section provides information on how Boogie must be extended in order to accept and translate the new specification information. Finally, the last section introduces runtime checks and explains how they are implemented.

Chapter 6 summarizes the implementation state and concludes the thesis.

1.2 Notation Conventions

Spec# programs Spec# or C# programs are enclosed in horizontal lines and typeset in typewriter font, like in the following example:

```
public class Program {
   public static void Main(string args[]) { }
}
```

Please note that, in general, the examples are not complete: members not strictly relevant to the context are usually omitted.

Inline statements and expressions, such as `Console.WriteLine("Hello, World")`, are shown in typewriter font.

BoogiePL code examples and translations BoogiePL examples and translations are typeset in typewriter font with numbered lines, like in the following example:

```
1 % this is a comment
2 assert Heap[d, System.Delegate._target] != null;
3 havoc m;
```

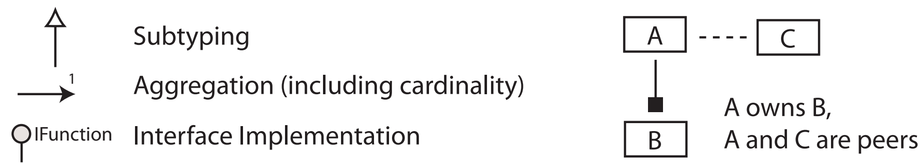
Sometimes we need to express pseudo-code instruction inside BoogiePL. Such instructions are typeset in italic font. The code example below means that Boogie will emit the assertion for each U which is a subtype of T:

```
1 #foreach U s.t. U <: T
2    assert Inv_U(p^i)
```

Discussing code Often code contained in examples is discussed in paragraphs. In this case, code elements, such as *Program.Main()*, are typeset in italic font.

Formulas and concepts Formulas and concepts not directly expressed in Spec# or BoogiePL format, such as $T <: U$ or $Pre_D(i^0)$, follow mathematical notation.

Classes and diagrams Symbols in classes and diagrams have the following meaning:



Operators and functions We present here a short list of ambiguous or non-standard operators.

Symbol	Language / Context	Description
`=`	Spec#	assignment operator
`==`	Spec#	equality operator
`:=`	BoogiePL	assignment operator
`=`	BoogiePL	equality operator
$=$	Mathematical	equality operator
$\bigwedge_i C(i)$	Mathematical	conjunction of clause $C(i)$ for all i
$<:$	Spec#, BoogiePL, Math.	reflexive subtyping operator: $U <: T$ means that U is equal or is a subtype of T
$E[o/n]$	Spec#, BoogiePL, Math.	substitution operator; every occurrence of o in E is replaced by n.

CHAPTER 2

Background

In this chapter we provide the background information needed to understand this work. Starting with an explanation of C# delegates, we introduce the Spec# programming system including the Spec# programming language, the Spec# compiler and Boogie. Finally we present the verification methodology for Spec# delegates [15] on which this thesis is based.

2.1 C# Delegates

C# delegates are object-oriented type safe function pointers [9]. A delegate declaration defines a class derived from *System.MulticastDelegate*. Delegates allow to encapsulate a reference to one or more methods, known as *callable entities*, inside a delegate object which can then be invoked. There are three steps in defining and using delegates: declaration, instantiation and invocation. The following example:

```
public delegate void MyDelegate(int i);
```

declares a delegate named *MyDelegate* with *void* return type and a single *int* argument. Delegates can be instantiated using the compiler generated constructor which takes the callable entity of the new delegate as argument. A callable entity is composed by a method only, in case the method is static, or by an instance and a method of that instance otherwise. We refer to the instance as *target object* and to the method as *target method* of a delegate. When a delegate is instantiated, the compiler checks that the signature of the delegate is the same as the method. A delegate can then be invoked like a normal method; arguments and return types are type-checked in order to ensure type safety.

The simple "Hello, World!" of delegates of Listings 2.1 shows how the delegate declared above can be instantiated and used. *MyDelegate* can point to any method with *void* return type and with one *int* argument. In method *Main()* a new instance of *MyDelegate* with target method *MyMethod()* is created. The new delegate is then invoked: this leads to a call of *MyMethod*.

Listing 2.1: Hello, World!

```
public delegate void MyDelegate(int i);

class Program {
    public static void MyMethod(int i) {
        Console.WriteLine("Hello, World: " + i);
    }

    public static void Main() {
        MyDelegate del = new MyDelegate(MyMethod);
        del(1);
    }
}
```

Use of Delegates

According to MSDN delegate should be used when:

1. An eventing design pattern is used.
2. It is desirable to encapsulate a static method.
3. The caller has no need access other properties, methods, or interfaces on the object implementing the method.
4. Easy composition is desired.
5. A class may need more than one implementation of the method.

In general delegates promote encapsulation. The idea behind it is that the client of a delegate has no need to know the identities of the target object and the target method. As we will see in later chapters however, there are situations (like in scenario 5) where the client wants, potentially, full knowledge of the target object and method.

In the next section we will explain how delegates are implemented on .NET.

2.1.1 Delegates in .NET

Delegates are implemented in .NET with a combination of syntactic sugar, compiler generated code and runtime handling. The general idea behind delegates is the following: for each delegate declaration, the compiler generates a new class which extends System.*MulticastDelegate*. The new class, whose is name is the same as the delegate, defines an *Invoke()* method which has the same signature as the delegate. An invocation of a delegate is translated to an invocation of its *Invoke()* method; *Invoke()* however does not define a body. It is the responsibility of the common language runtime (CLR) to handle the invocation of the target methods of the delegate. Listings 2.2 shows part of the compiler-generated code of the previous example,

As shown in the example the actual implementation of the constructor and the *Invoke()* method is completely handled at runtime. The instantiation of the delegate is analogous to

Listing 2.2: Disassembled version of a program using delegates

```
public class MyDelegate : System.MulticastDelegate {
    [MethodImpl(0, MethodCodeType=MethodCodeType.Runtime)]
    public MyDelegate(object @object, IntPtr method);

    [MethodImpl(0, MethodCodeType=MethodCodeType.Runtime)]
    public void Invoke(int i);
}

class Program {
    ...
    public static void Main() {
        MyDelegate del = new MyDelegate(null, (IntPtr)MyMethod);
        del.Invoke(1);
    }
}
```

the instantiation of an object (in fact it is an instantiation of an object), while the invocation is de-sugared to a invocation to *Invoke()*.

When the callable entity of a delegate is a static method the target object is not specified, e.g. it is set to *null*.

Finally delegate calls obey dynamic dispatch. This means that if a delegate d has a virtual method m for some object o of type T as callable entity, the invocation of the delegate results in a call to the most derived override of m defined in a type U such that $U <: T$, where U is the dynamic type of o.

2.1.2 Multicast Delegates

After instantiation, a delegate has a single callable entity, namely the one specified in the constructor. We refer to delegates having a single callable entity as *singlecast* delegates. Delegates of the same type however can be combined to form a new delegate having multiple callable entities. We refer to delegates having more than one callable entity as *multicast* delegates. The distinction however is only conceptual; in practice all delegate is a subtype of *MulticastDelegate* and singlecast and multicast delegates are used in the same way.

In the program of Listings 2.3, m is obtained by combining the singlecast delegate s with a new singlecast delegate; s and m are both of type *MyDelegate*. An invocation of s results in an invocation of *Target()*; an invocation of m however will first invoke *Target()* and then invoke *AnotherTarget()*. A multicast delegate is defined by its invocation list: an array of singlecast delegates which are invoked by the multicast delegate. This list is obtained through to the *GetInvocationList()* method defined in *System.Delegate*. Let us see how this works in more detail.

In the following we will only consider the invocation list returned by the public *GetInvocationListMethod()*. Delegates also maintain a private invocation list which is different from the one returned by the method and that we will ignore as it is only part of the specific implementation. The invocation list of a singlecast delegate only contains the delegate itself. Method *Delegate Delegate.Combine(Delegate, Delegate)* returns a new delegate of the same type of the delegates passed as argument and with an invocation list obtained by con-

Listing 2.3: Combining delegates

```
public delegate void MyDelegate();

class Program {
    static void Target() { }
    static void AnotherTarget() { }

    public static void Main() {
        MyDelegate s = new MyDelegate(Target);
        MyDelegate m = Delegate.Combine(
            singlecast, new MyDelegate(AnotherTarget));
    }
}
```

catenating the invocation lists of the two arguments[1]. It is also important to mention the possibility of removing delegates from delegates. As for the case with combining delegates, the .NET library offers methods like *Delegate Delegate.Remove(Delegate, Delegate)* which effect is similar to *Degate.Combine* but with the difference that the invocation lists are subtracted instead of concatenated. C# offers some useful syntactic sugar for combining and removing delegates: the plus (`+`) operator is replaced by *Delegate.Combine(Delegate, Delegate)* while the minus (`-`) operator is replaced by *Delegate.Remove(Delegate, Delegate)*. Moreover the plus-equal (`+=`) and minus-equal (`-=`) operators are also defined: `d1 += d2` is equivalent to `d1 = d1 + d2`, while `d1 -= d2` is equivalent to `d1 = d1 - d2`

It is important to note that even though the resulting delegate of *Delegate.Combine* is a new instance, the invocation list contains references to the delegates present in the invocation list of the two arguments; this means that *Delegate.Combine* creates at most one additional delegate. The same also applies for *Delegate.Remove*. To make matters more complicated the invocation lists are flattened and the same instance of a delegate can be contained more than once in the invocation list. Consider the following example:

```
MyDelegate singlecast = new MyDelegate(Target);
MyDelegate multicast1 = Delegate.Combine(singlecast, singlecast);
MyDelegate multicast2 = Delegate.Combine(multicast1, singlecast);
```

After the execution of the code above we have:

1. *singlecast* is a singlecast delegate; its invocation list contains a reference to itself;
2. *multicast1* is a multicast delegate; its invocation list contains two references to *singlecast*;
3. *multicast2* is also a multicast delegate; its invocation list contains three references to *singlecast* (not one reference to multicast1 and one reference to *singlecast*).

Flattening the invocation list and returning a new delegate while combining, ensures that the effects of combining delegates are limited to the newly created delegate. When querying a multicast delegate for its target object and method, the result is equivalent to the target object and method of the last delegate in the invocation list.

[1]When one of the delegates is null, the other one is returned; this means that *Delegate.Combine* returns null in case both arguments are null.

Listing 2.4: Observer pattern implemented with events

```
public delegate void UpdateEventHandler(Observable o);

public class Observable {
    public event UpdateEventHandler Changed;
    protected void Notify() { if (Changed != null) Changed(this); }
}

public class Observer {
    public void Update(Observable o) { }
}

public class Program {
    public static void Main() {
        Observable target = new Observable();
        Observer observer = new Observer();
        target.Changed += new UpdateEventHandler(observer.Update);
    }
}
```

2.1.3 Immutability of Delegates

From a client's perspective a delegate is completely defined by its callable entities. This means that a singlecast delegate is defined by its target object and method, while a multicast delegate is defined by its invocation list.

Delegates override the equality operator. Two multicast delegates a and b are equal if their invocation lists are equal, e.g. if all delegates in the invocation list are equal, the size of the invocation lists is the same and the order is preserved. Two singlecast delegates in the invocation list are equal if their target objects and target methods are the same [6]. Always from a client's perspective, given a delegate d, $d.Target$ and $d.Method$ cannot be changed. If we add this information to the fact that every operation that could modify the invocation list of a delegate, returns a new delegate, we can conclude that from the client's point of view delegates are immutable.

2.1.4 Events

An *event* is a member that enables an object or class to provide notifications [9].

Listings 2.4 shows an example of the Observer pattern implemented with events. The *Changed* event declared in *Observable*, is translated by the compiler in a private field of type *UpdateEventHandler* and public methods used to add and remove delegates from this field. For an observer to register it is sufficient to instantiate a new delegate with an update method as target, and add it to *Changed* event using the `+=` operator which is translated by the compiler to an invocation of the method used to add delegates to the event.

The disassembled version of the example above is shown in Listings 2.5.

Listing 2.5: Decompiled version of the observer pattern implemented with events

```
public class Observable {
    private UpdateEventHandler Changed;

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void add_Changed(UpdateEventHandler modopt(NonNullType) value) {
        this.Changed = (UpdateEventHandler) Delegate.Combine(this.Changed, value);
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void remove_Changed(UpdateEventHandler value) {
        this.Changed = (UpdateEventHandler) Delegate.Remove(this.Changed, value);
    }

    protected void Notify() {
        if (this.Changed != null) {
            this.Changed.Invoke(this);
        }
    }
}
```

2.2 The Spec# Programming System

The Spec# Programming System is composed by the Spec# programming language, the Spec# compiler and Boogie. With Spec#, a programmer includes specification information in the source code. The program correctness, with respect to its specification and the semantics of the language, is then verified by Boogie.

2.2.1 The Spec# Programming Language

The Spec# [5] programming language is a superset of C#. It allows specifying method contracts in the form of pre-, post- and frame conditions and type contracts in the form of invariants. The type system of C# is extended with non-null types and checked exceptions [5]. Specification information is expressed with a combination of additional syntax and attributes. The language also includes *assert* and *assume* statements and quantifications.

With an object invariant it is possible to define the consistent state for that object. An object is said to be consistent if its invariant is known to hold. Consistency is defined with respect to a specific class frame. Consider an object o of type X, where $X <: Y <: Object$; every class in the hierarchy can define some invariant. Object o is said to be valid (consistent) for T if the invariants for all types U, such that $T <: U$, are satisfied; in the example o is valid for Y if the invariants defined in Y and $Object$ (trivially true) are satisfied. An object is fully valid, if it is valid for its dynamic type. In Spec# every object is given an addition field inv, which stores the most derived class frame for which the object is known to be valid [3]. The invariant for a class frame T can be temporarily violated if $\neg o.inv <: T$. Spec# provides a special block statement that allows to modify the inv field of an object: *expose (o)* { S }. Before entering the block, the inv field of o is set to the superclass of T, the type of o: the object is exposed. Inside the block the invariant defined in T can be violated, while at the end of the block T's invariant is checked and $o.inv$ is set to T. The inv field of an object can only be modified through an expose statement; this ensures that outside of

any expose block, an object is always fully valid. The following program invariant is then defined:

Program Invariant 1. *Object invariants*

$$P1 : (\forall o, T \cdot o.inv <: T \Rightarrow Inv_T(o))$$

where Inv_T is the invariant declared in class T. In order to handle invariants for aggregate objects, Spec# employs the concept of ownership. An object is allowed to express invariants for all objects it owns. Ownership requires two additional fields[2] given to every object; $o.ownerRef$ is a reference to the owner of o, while $o.ownerFrame$ is the type of the owner [13]. The validity of an object is now conditional to the validity of all the objects it owns. This lead to the following programming invariant:

Program Invariant 2. *Ownership*

$$\begin{aligned} P2 :& (\forall o, T \cdot o.inv <: T \Rightarrow \\ & (\forall p \cdot p.ownerRef = o \wedge p.ownerFrame = T \Rightarrow p.inv = typeof(p)) \end{aligned}$$

Program invariant **P2** states that if an object is valid, then all object it owns must also be valid. With respect to invariants and ownership, we can define the following properties for objects:

Definition 2.1 (Validity)**.** An object o is valid for class frame T iff $o.inv <: T$.

Definition 2.2 (Full Validity)**.** An object o is fully valid iff $o.inv = typeof(o)$.

Definition 2.3 (Consistency)**.** An object o is consistent iff o is valid and either it has no owner or its owner is exposed, e.g. if $o.inv = typeof(o)$ and

$$o.ownerFrame = PeerGroupPlaceholder \vee \neg o.ownerRef.inv <: o.ownerFrame$$

$PeerGroupPlaceholder$ is used on the $ownerFrame$ field to indicate that the object has no owner.

Definition 2.4 (Peer-Validity)**.** An object o is peer-valid iff o and all its peers are valid, e.g. if

$$\begin{aligned} & o.inv = typeof(o) \wedge (o.ownerFrame = PeerGroupPlaceHolder \vee \\ & \quad (\forall p \ \cdot \ p.owner.Ref = o.ownerRef \wedge p.ownerFrame = o.ownerFrame \\ & \quad \Rightarrow p.inv = typeof(p))) \end{aligned}$$

Definition 2.5 (Peer-Consistency)**.** An object o is peer-consistent if it is peer-valid and either it has no owner or its owner is exposed, e.g. if

$$\begin{aligned} & o.inv = typeof(o) \wedge (o.ownerFrame = PeerGroupPlaceholder \vee \\ & \quad (\neg\ o.ownerRef.inv <: o.ownerFrame \wedge (\forall p \cdot p.owner.Ref = o.ownerRef \wedge \\ & \quad p.ownerFrame = o.ownerFrame \Rightarrow p.inv = typeof(p)))) \end{aligned}$$

Peer-consistency is an important requirement for arguments of methods. A method, receiving an object o as argument, needs to know that o is valid; moreover the method could also modify o. For this reason every non-pure methods has a default pre-condition requiring peer-consistency of all parameters. If an object is peer-consistent then it is also valid and its owner is known to be exposed, which allows the object to be changed.

[2]In the original publication an object is given a single additional *owner* field whose value is a pair [*object obj*, *type typ*]. In our notation however we choose to remain closer to the actual BoogiePL encoding of ownership by providing two distinct fields.

2.2.2 The Spec# Compiler

The Spec# compiler compiles Spec# source files in .NET intermediate language (CIL) and it is responsible for performing initial static checks including:

- Non-null analysis;
- Checked exceptions;
- Side effects-free specifications;
- Admissible specification.

The compiler's type checker verifies the usage of non-null types and generates warnings in case of violations.[3] Checked exceptions are verified to ensure that they are actually caught. Side effects-free specification means that all expressions used for specification are not allowed to have side effects. This is accomplished by forbidding operators that can modify fields - like for example the assignment operator - and requiring all methods used in the specification to be pure. Moreover expressions used in specification must be admissible. As an example, every target of field access or method calls in invariants must be (recursively) owned by the object declaring the invariant. It is up to the compiler to check the admissibility of such expressions.

Additionally the compiler instruments the code for runtime checks. The body of methods is extended so that violations of pre- and post-conditions result in runtime exceptions. When entering an expose block an exception is thrown if the object is not exposable and when leaving it an exception is thrown if the invariant does not hold. Frame conditions are not checked at runtime because of performance reasons.

2.2.3 Boogie

Boogie is a static program verifier that takes as input the abstract syntax tree (AST) generated by the Spec# compiler or a compiled CIL library, translates the program in Boogie [1] programming language (BoogiePL) and finally feeds the translated program to a theorem prover. As a last step Boogie returns the list of failing conditions. Currently Boogie uses the Z3 theorem prover [7], but optionally Simplify can be used as well. The first step in the verification done by Boogie is to obtain the AST of the program to verify. This is either directly fed to Boogie by the Spec# compiler or is reconstructed from CIL.

A BoogiePL program follows a certain number of transformations [4] which can be summarized in three steps: (1) cutting back edges to obtain a loop-free program, (2) transformation in passive form and finally (3) weakest-pre-condition computation to obtain the verification condition that is finally fed to the theorem prover. In this report we focus on the steps leading to the BoogiePL program and we take the later transformations as granted.

2.2.4 BoogiePL

BoogiePL [8] retains procedures, mutable variables and pre- and post-conditions. It introduces constants, function symbols, axioms and non-deterministic control-flow. A BoogiePL

[3] Non-null analysis is unsound: there are situations where a variable of a non-null type can be null without the compiler emitting a warning or error. Please note that is is a bug of the current implementation of Spec#.

program consists of two parts. The first part is a theory used to encode the semantics of the source language consisting of type declarations, symbol declarations and axioms. The second is an imperative part used to encode the source program; it consists of global variable declarations, procedure headers and procedure implementations. A procedure header defines the signature of the procedure and its contract; it is used for verifying callers of the corresponding Spec# method or to express general modifications of the heap. A procedure implementation contains the translation in BoogiePL of the body of the corresponding Spec# method; it is used to verify that the implementation of a method respects its specification.

Data Types BoogiePL has five built-in data types: integers (*int*), booleans (*bool*), references (*ref*), type and field names (*name*) and any (*any*). Boogie also supports arrays and user-defined data types. Arrays can be indexed by any type, not only by *int*.

Commands BoogiePL defines a number of commands; relevant to this report are: *assert* and *assume*, indicating conditions to be checked or used; *havoc*, which assigns an arbitrary value to a variable and finally *call*, used to simulate a procedure call and whose effect is to assert the pre-condition of the procedure, havoc the heap and finally assume the post-condition.

Procedures Procedures support *in* and *out* parameters, this also allows to encode *ref* and *out* Spec# parameters. Frame conditions are encoded as post-conditions. In Spec# frame conditions state what the method might change; in BoogiePL the opposite holds and frame conditions state what the method does not change.

Heap The heap is encoded in BoogiePL as a global, two-dimensional array indexed by a reference to an object and the name of a field of that object. The array is named *Heap* and declared as: *var Heap: [ref,<x>name]x*. Instance fields access of the form *o.f* are translated in *Heap[o, T.f]* where *T* is the class where *f* is declared and *T.f* is a constant of type name. Object allocation is encoded using the additional *allocated* field.

Invariants The Spec# *inv* field is encoded in BoogiePL using two fields: *inv* is used for additive expose, while *localinv* is used for normal expose.

Fields' Axioms Properties of fields are encoded in BoogiePL using axioms. *axiom IncludeInMainFrameConditions(f)* indicates that the field is part of the frame condition, while *IncludedInModifiesStar(f)* indicates that the field can be changed by a *modifies o.** clause; *typeof(o)* encodes the type of object *o*.

Functions Some properties are encoded in BoogiePL using functions. Relevant to this work are *BaseClass(T)* which returns the base class of *T*; *IsHeap(h)* indicating whether array *h* is a well-formed heap and finally *HeapSucc(oldHeap, newHeap)* which indicates whether *newHeap* is a valid successor of *oldHeap*.

Listings 2.7 shows the translation of the program of Listings 2.6 in BoogiePL. For simplicity reason the background theory and other parts of the program are omitted.

Listing 2.6: A simple class representing a point

```
public class Point {
    [SpecPublic] private int x, y;

    invariant x >= 0 && y <= 0;

    public void Add(Point! other)
        requires other.x > 10;
    {
        this.x += other.x; this.y += other.y;
    }
}
```

As the example shows, the BoogiePL program starts by defining constants and emitting axioms describing a theory for a *Point*. This part ends with *Point*'s object invariant. In the second part the procedure headers and implementations are emitted. Spec# includes a large number of default pre- and post- conditions which are shown as comment in the header of *Point.Add*. Finally comes the actual body of the method. The pre-condition *other$in != null* ensures that the assertion *other != null* holds. Like in CIL, stack variables are extensively used in the implementation of procedures.

Listing 2.7: BoogiePL translation of Listings 2.6

```
const unique Point.x: <int>name; const unique Point.y: <int>name;
const unique Point: name;
axiom Point <: Point;
axiom $BaseClass(Point) == System.Object &&
   AsDirectSubClass(Point, $BaseClass(Point)) == Point;
axiom !$IsImmutable(Point) && $AsMutable(Point) == Point;
axiom (forall $oi: ref, $h: [ref,<x>name]x :: && $h[$oi, $inv] <: Point
   ==> $h[$oi, Point.x] >= 0 && $h[$oi, Point.y] <= 0);

procedure Point.Add$Point$notnull(this: ref, other$in: ref
      where $IsNotNull(other$in, Point));
   requires $Heap[other$in, Point.x] > 10;
   // target object is peer consistent (omitted)
   // other is peer consistent (omitted)
   modifies $Heap, $ActivityIndicator;
   // frame condition
   ensures (forall $o: ref, $f: name :: { $Heap[$o, $f] }
      IncludeInMainFrameCondition($f) && ... ==> old($Heap)[$o, $f] == $Heap[$o, $f]);
   free ensures $HeapSucc(old($Heap), $Heap);
   // inv/localinv change only in blocks (omitted)

implementation Point.Add$Point$notnull(this: ref, other$in: ref)
{
   var other: ref, stack0i: int, stack1i: int, temp0: exposeVersionType,
      temp1: exposeVersionType;

   entry:
      other := other$in;
      goto block2499;
   ...
   block2635:
      assert this != null;
      stack0i := $Heap[this, Point.x];
      assert other != null;
      stack1i := $Heap[other, Point.x];
      stack0i := stack0i + stack1i;
      assert this != null;
      $Heap[this, Point.x] := stack0i;
      assert this != null;
      stack0i := $Heap[this, Point.y];
      assert other != null;
      stack1i := $Heap[other, Point.y];
      stack0i := stack0i + stack1i;
      assert this != null;
      $Heap[this, Point.y] := stack0i;
      assert !($Heap[this, $inv] <: Point && $Heap[this, $localinv] !=
         $BaseClass(Point)) || $Heap[this, Point.x] >= 0;
      assert !($Heap[this, $inv] <: Point && $Heap[this, $localinv] !=
         $BaseClass(Point)) || $Heap[this, Point.y] <= 0;
      assume IsHeap($Heap);
      return;
}
```

2.3 Previous Work on Static Verification of Delegates

In this section we describe the methodology [15] which is the foundation of this work. We introduce one concept at the time, by first assuming a simplified situation and then gradually building up a complete solution.

The issue in the verification of programs using delegates, as we saw in the introduction, is related to the specification of methods. Before invoking a method, the caller must ensure that the pre-condition of the method holds, moreover the caller can assume the post-condition of the callee to hold. It is up to the implementer of a method to make sure that the body of the method is correct, e.g. that its execution satisfies its post-condition given the assumption that the pre-condition holds. With delegates this is more difficult because, in principle, the target method is unknown. In this perspective it is useful to see delegates as methods executing a single operation, namely invoking their target. In order for a delegate invocation to be correct, the pre-condition of the target method must hold before invocation and the post-condition of the delegate must hold after invocation.

The idea for the verification of delegates is based on two steps: the first step consists in providing delegates with a specification similar to the one for methods; in the second step delegate instantiations are checked to verify that the delegate is a *refinement* of its target method:

Definition 2.6 (Refinement)**.** The specification of a method m is a refinement of the specification of a method n (or in shorter form m is a refinement of n) if:

1. The pre-condition of m implies the pre-condition of n, for all possible parameters, heaps and
2. The conjunction of the pre-condition of m with the post-condition of n implies the post-condition of m for all possible parameters, return values and pre-state and post-state heaps.

When the delegate is a refinement of its target method, we can be sure that if the pre-condition of the delegate holds, so will the pre-condition of the target method, and conversely if the post-condition of the target method holds after invocation, so will the post-condition of the delegate.

2.3.1 Contracts For Delegates

Consider the example shown in Listings 2.8. We want to ensure that the delegate invocation *logFile()* in the *Log()* method does not break the pre-condition of its target method. If this can be proven, then we can assume the call to be correct.

For the time being we ignore the second pre-condition of *Store*; to prove this program we can simply add a pre-condition requiring the parameter to be non null to the delegate:

```
delegate void Archiver(object p);
    requires p != null;
```

This pre-condition can be seen as a usual method pre-condition; the caller must make sure that the pre-condition of the delegate holds before invoking it and, since this pre-condition is the same as the one defined in the target method (namely that the parameter is not null),

Listing 2.8: A simple USBStick

```
class USBStick {
    public void Store(object p)
        requires p != null;
        requires IsPeerConsistent(this);
    { /* Store p */ }
}

delegate void Archiver(object p);

class Client {
    public static void Log(Archiver! logFile, string! s)
        requires IsPeerConsistent(logFile);
    { logFile(s); }
}
```

at this point we can assume the call to be correct[4].

Delegates are provided with pre- and post-conditions as the ones defined for methods but with the exception that delegate pre- and post-conditions are not allowed to refer to the state of the target object. The limitation is due to the fact that the client of a delegate is not concerned with the identity of the object defining the target method which could also be statically unknown.

The Peer-Consistency Requirement

In the above example we ignored the second pre-condition of the *Store* method which requires its target to be peer-consistent. Given the fact that peer-consistency is required for all non-pure methods, delegates must provide a way to ensure peer-consistency of their target upon invocation with as less overhead as possible. For this reason delegates and their targets are set to be peers. The precondition for a delegate invocation requires the delegate to be peer-consistent; this ensures peer-consistency of the target as well. In the example, given program invariant **P1** we know that *logFile* is a refinement of *Store*; moreover since *Log()* already requires *logFile* to be peer-consistent and *s* is non-null, the pre-condition of *logFile* is established. By refinement this implies the pre-condition of *Store* which allows us to conclude that the program is correct.

The following program invariant can be stated:

Program Invariant 3. *A valid delegate is peer with its target*

$$P3 : (\forall o, d \bullet d.target = o \wedge d.inv <: typeof(d) \Rightarrow d.owner = o.owner)$$

2.3.2 Delegates Invariants

Consider again our USBStick example shown in Listings 2.9, this time with the additional pre-condition in the *Store* requiring the stick to be loaded.

This pre-condition clearly depends on the state of the target, yet the client of the delegate,

[4] As long as Store does not modify existing objects in the heap.

Listing 2.9: USBStick with pre-condition on the target object

```
class USBStick {
    public bool IsLoaded;

    public void Store(object! p)
        requires this.IsLoaded;
    { /* Store p */ }
}

delegate void Archiver(object p);

class Client {
    public static void Log(Archiver! logFile, string! s)
    { logFile(s); }
}
```

Log(), does not have any information about the target. The burden of verifying whether targets of *logFile* of type *USBStick* are loaded should not fall on the client of the delegate; this prevents us from using a normal pre-condition[5].

The solution lies in delegating the responsibility to check target-dependent pre-conditions to the method instantiating the delegate who, in general, is completely aware of the target object. The concept of invariant comes to mind. Delegates are allowed to define visibility based invariants depending on the state of the target object. Delegate invariants can be expressed with an `invariant for T is E` clause, where T is a type and E an expression. Invariants for delegates are de-sugared in the form: `invariant (target is T) ==> S[target/((T)Target)`. The declaration of *Archiver* becomes:

```
delegate void Archiver(object! p);
    invariant for USBStick is target.IsLoaded;
```

This invariant states that when the target of the delegate is of type *USBStick*, then the target must be loaded. All delegate invariants are checked during the instantiation of the delegate and as long as the delegate itself is valid, delegate invariants hold. Delegates in fact, like every other object, are enhanced with an *inv* field representing the validity of the object, hence program invariant **P1** holds for delegates as well.

When an object declaring a field mentioned in some delegate invariant is exposed, the invariant of the delegate could be violated. To ensures that program invariant **P1** holds even in such situations, all objects declaring a field mentioned in a delegate invariant for some delegate D, are required to also declare a *dependent* clause on D. Thanks to this clause, when these objects are exposed it is possible to automatically expose all delegates they depend upon. This leads to the following program invariant

Program Invariant 4 (Dependency of classes)**.**

$$\begin{aligned} P4 :& (\forall o, d, T, D \cdot d.target = o \wedge d.inv <: D \wedge D \in dependents(T) \wedge \\ & typeof(o) <: T \Rightarrow o.inv <: T) \end{aligned}$$

[5]A possible pre-condition for the delegate would be of the form: `requires this.target is USBStick ==> ((USBStick)this.taget).IsLoaded`.

2.3.3 Delegate Subtyping

Consider a situation where a program uses a delegate defined in a library and the target method has a pre-condition depending on the target object. It would be impossible to verify the delegate without modifying the library; this would break modularity. To ensure modularity in delegate invariants the definition of delegates is extended to allow sub-typing: a delegate is allowed to extend another delegate, provided that the signatures are the same. Delegate subtypes, like method overrides, are allowed to refine the specification of their base type: a delegate subtype is allowed to declare additional delegate invariants, provided that these are compatible with the contract of the base delegate. In the context of the above example, instead of adding an invariant to *Archiver*, it is possible to subtype *Archiver* as follows:

```
delegate void USBArchiver(object! p) : Archiver;
   invariant for USBStick is target.IsLoaded;
```

The invariant for *USBStick* is now defined in the delegate subtype. The instantiation of the delegate passed to the *Log()* method becomes:

```
USBStick stick = new USBStick();
Archiver logFile = new USBArchiver(stick.Store);
Client.Log(logFile, "Hello, World!");
```

Since at instantiation refinement checks are performed, we can be sure that only *USBArchiver* can be used with *Store* as target. The new delegate, being a subtype of *Archiver* can easily be stored in a variable of the latter type and this allows for complete transparency in the use of delegate subtype with the existing implementation. The new delegate can in fact be used by the *Log()* method which is not concerned with the actual type of the delegate.

Introducing additional invariants is possible because invariants are checked during instantiation and are ensured to hold as long as the delegate object is valid. This shows the dual role of delegate invariants. With respect to a delegate and its target objects, invariants can be seen as part of the pre-condition of the delegate; this is necessary to show that the delegate's contract is a refinement of its target method contract. With respect to delegates related by a sub-typing relationship however, delegate invariants are not considered to be part of the pre-condition, for this reason new invariants can be added in subtypes without breaking the refinement principle.

2.3.4 Disabling Delegates

The scope of delegate invariants has yet to be explained: can they be broken? And if yes, how? To better understand the need for violating a delegate invariant, consider the scenario presented in Listings 2.10. In this example *USBStick* provides a method for ejecting itself, namely *Eject()*; this method violates the delegate invariant without reestablishing it at the end of the expose block.

As it is the *Eject()* method cannot be verified, we show this with a counterexample. Suppose a delegate d of type *USBArchiver* has been instantiated with an instance o of type *USBStick* as target object and an instance method of *USBStick* as target method, for example *Store*. Also suppose that *o.Eject()* is called. As we know the implicit pre-condition for the invocation of *Eject()* is for its target object o to be peer-consistent; we also know that o and d are peers. This means that just after entering the *Eject()* method o is valid, d is

Listing 2.10: Breaking a delegate invariant.

```
class USBStick {
    public void Eject()
        requires this.IsLoaded;
        ensures !this.IsLoaded;
    {
        expose (this) {
            /* Eject the stick */
            this.IsLoaded = false;
        }
    }
}

delegate void USBArchiver(object! p);
    invariant when { USBStick; target.IsLoaded };
```

valid and o's owner is exposed. In order to violate d's invariant we would have to expose it. However all objects exposed in a method are un-exposed before leaving the method and this requires their invariants to be satisfied. This implies that even if we exposed d, we would not be able to satisfy its invariant before leaving *Eject()*.

Ownership Based Disabling

The main problem related to breaking a delegate invariant by changing the value of a field of an object o, is the existence of valid instances of delegates declaring that invariant and having o has target. That said, breaking a delegate invariant is possible when we can ensure that no such instance of a delegate exists. The problem is solved by adding the following pre-condition to the *Eject()* method:

$$\texttt{requires}\ (\forall d \bullet d.target = this \Rightarrow \neg d.inv <: USBArchiver);$$

This pre-condition states that the *Eject()* method can only be called when all instances d of a delegate of type *USBArchiver* either have a target which is different from *this*, or are exposed. In order to be able to satisfy this condition, the methodology introduces a statement `disable (D for o)`. This statement has the effect of *disabling* all delegates of type D with target object o: all such instances are exposed and un-owned. In other words the execution of the statement disables all delegates d such that $d.target = o \wedge d.inv <: D$ by setting their invariant to *Object* and by un-owning them.

When a delegate is disabled, its invariant is no longer ensured to hold. A call to a disabled delegate could then violate a pre-condition of the target method. For this reason it must be ensured that a disabled delegate cannot be invoked. The approach described above, directly complies with this additional requirement. Given the fact that a disabled delegate is always exposed (e.g. not valid) and that one of the pre-condition for invoking the delegate is for the delegate to be peer-consistent (peer-consistency requires all objects in the peer-group to be valid), it is clear that a disabled delegate cannot be invoked.

CHAPTER 3

Implementation of Static Verification for Singlecast Delegates

3.1 Contracts and Subtyping

As described in [15] we provide delegates with pre- and post-conditions allowed to depend on the arguments of the delegate. By adding pre- and post-conditions to delegates, we are able to verify target methods which specification does not depend on the state of the target object. By introducing delegate invariants we are able to verify target methods having pre-conditions depending on the state of the target and pre- and post-conditions depending on the arguments.

3.1.1 Encoding and Definition of Contract Elements

In Chapter 2 we mentioned that a delegate invocation is de-sugared in an invocation of its *Invoke()* method. For the Spec# implementation of delegates we generate an additional method in the delegate class: *SafeInvoke()*[1]. This method as the same signature as *Invoke()*, but additionally it includes specification information. The pre-, post- and frame conditions of a delegate are in fact copied to its *SafeInvoke()* method. Given the fact that *SafeInvoke()* behaves like any other normal method, Boogie will automatically check the validity of its pre-condition before a call, and assume its post-condition after the call.

Delegate invariants are encoded as normal object invariant in the delegate class. In a way similar to [15], we define an admissible delegate invariant as follows:

Definition 3.1 (Admissible Delegate Invariant)**.** An invariant for T declared in or inherited by a delegate type D is admissible if and only if: (i) its sub-expressions type-check under the assumption that target is of type T; (ii) each of the field-access expressions has the

[1] *SafeInvoke()* is introduced because *Invoke()* does not have a body in CIL (the invocation of the target is performed by the CLR), but an actual executable body is needed for implementing runtime checks. At runtime *SafeInvoke()* performs runtime checks and invokes the *Invoke()* method.

form *target.f*, where f is a field transitively owned by T and f is not one of the pre-defined fields *inv*, *ownerRef*, *ownerFrame*, etc.; (iii) D is mentioned in the dependent attribute of all fields used in the invariant.

Syntax and Naming Conventions

In the original paper, each type having a field mentioned in an invariant of a delegate D was required to declare a dependency to D. In our approach we follow the current Spec# implementation by requiring to add a *Dependent* attribute on fields mentioned in delegate invariants. The original dependency relationship can still be reconstructed: a type T is dependent on a delegate D if T declares one or more fields with a *Dependent* attribute on D.

We express pre-, post- and frame conditions using the same syntax as for methods. A delegate invariant is expressed using a `invariant when { T; E }`, where T is a type and E is an expression. The identifier *target* is used in E to refer to a target object of type T. The following example shows a delegate declaration with an invariant:

```
delegate void USBArchiver(string! s);
    invariant when { USBStick; target.IsLoaded };
```

In CIL, the internal field storing a reference to the target object of a delegate, is named *_target*; in BoogiePL the field is named *System.Delegate._target*. *System.Delegate* provides a public property *Target* for retrieving the target of a delegate. In CIL this is equivalent to a call to the *get_Target* method. When translating a program to BoogiePL, we substitute all occurrence of *d.get_Target* with *Heap[d, System.Delegate._target]*. In the rest of this document, when we do not need to explicitly consider the heap, we refer to *Heap[d, System.Delegate._target]* as *d._target*.

So far, when discussing delegate contracts, we referred to pre-, post-, frame conditions and invariants as single elements. We now need to point out the difference between the theoretical and the practical approach. Consider the following example:

```
public void Divide(Point! p, Point! q)
    requires q.x != 0;
    requires q.y != 0;
```

The complete pre-condition of the method *Divide()* is $q.x \neq 0 \wedge q.y \neq 0$; this precondition however is encoded in two separate requires clauses. In order to be able to reason about concepts such as pre-condition, invariants, etc., we need to precisely define them.

Definition 3.2 (Partial Invariant of a Delegate)**.** The partial invariant $PartInv_D$ of a delegate of type D is the conjunction of all condition E for all `invariant when { T; E }` clause defined in D.

Definition 3.3 (Invariant of a Delegate)**.** The invariant Inv_D of a delegate of type D is the conjunction of all condition E for all `invariant when { T; E }` clause defined or inherited by D.

Definition 3.4 (Precondition of a Delegate)**.** The precondition Pre_D of a delegate of type D is the conjunction of all condition E for all `requires E` clause defined or inherited by D.

Definition 3.5 (Post-condition of a Delegate)**.** The post-condition $Post_D$ of a delegate of type D is the conjunction of all condition E for all `ensures E` clause defined or inherited by D.

The last two definitions also apply to Pre_m and $Post_m$, the pre- and post-condition of a method m.

3.1.2 Delegate Instantiation in BoogiePL

We consider an instantiation of a delegate of the form:

`d = new D(o.m)`, where

- d is a variable of type D;
- o is a possibly null reference to an object of type T;
- m is the name of an instance method defined in a class U such that T is a subtype of U;
- the signature of m is the same as the signature of D.

We first present the pseudo code given in [15], and then discuss how the statement is encoded in BoogiePL.

```
1  assert o != null;
2  #foreach T s.t. D ∈ dependents(T)
3      assert o.inv <: T;
4  d := new D;
5  d.target := o; d.owner := o.owner;
6  #foreach E s.t. D<:E
7      assert Inv_E(d);
8  d.inv := D
9  assert (∀p,h · Pre_D(d,p,h) ⇒ Pre_m(o,p,h));
10 assert (∀p,r,h,h′ · Pre_D(d,p,h) ∧ Post_m(o,p,r,h,h′) ⇒ Post_D(d,p,r,h,h′));
```

Line 1 The first part of a delegate instantiation in BoogiePL is analogous to the instantiation of any other object. In extended format, line 1 is generated as follows:

```
1 havoc m
2 havoc d;
3 assume !d.allocated && d != null && typeof(d) == D;
4 assume d.owner.ref = d && d.owner.frame = PeerGroupPlaceholder;
5 assert stack0o != null;
```

We havoc the target method of the delegate because, for the instantiation of the delegate, its identity is not relevant. The refinement checks explained later in fact ensure that the delegate is a refinement of the method.

Lines 2-3 At lines 2 and 3 we assert the invariant of all objects the delegate depends on. In order to achieve this we keep track of a list of *dependees* for each delegate type; the list contains all types which are dependent on the delegate. The translation is the following:

```
1 #foreach T s.t. D ∈ dependents(T)
2     assert typeof(o) <: T ==> o.inv <: T && o.localinv <: T;
```

Line 4 The actual call to the constructor at line 4 is also translated like a normal constructor call, namely:

```
1 assert d != null;
2 call D..ctorSystem.ObjectSystem.IntPtr(d, o, m);
```

The assertion is necessary to ensure that the target object is indeed non-null. Note however that the constructor call could be omitted; it is maintained in the translation for clarity reasons only.

Line 5 At line 5 the target object and the owner are assigned. For the actual translation the owner is modified using the *SetOwner* procedure, which is already defined in BoogiePL:

```
1 d.target := o;
2 SetOwner(d, o.owner.ref, o.owner.frame);
```

Lines 6-7 At lines 6 and 7, the invariant of the delegate is asserted for all its type frames:

```
1 #foreach E s.t. D<:E
2     assert PartInv_E(d)[System.Delegate._target/o];
```

The assertion is obtained by replacing o to *System.Delegate._target* in $PartInv_D$.

Line 8 At line 8 the new delegate is set to valid:

```
1 d.inv := D; d.localinv := typeof(d);
```

Line 9 The refinement check for the pre-condition must verify that $Pre_D \wedge Inv_D \Rightarrow Pre_m$. In order to accomplish this we must match the names of all parameters in Pre_D and Pre_m. Moreover the name of the target object in Pre_m must match the target of Inv_D. In refinement checks we refer to pre-state parameters of delegate and method as $p_0^0, p_1^0, ..., p_n^0$, to the target object as t and to the pre-state heap as h^0. We define the pre-, post- conditions and the invariant used for refinement checks as:

$Pre_D Subst$ and $Inv_D Subst$ are obtained from Pre_D and Inv_D as follows:

1. Substitute all parameter name with $p_0^0, p_1^0, ..., p_n^0$ while maintaining the correct order
2. Substitute all occurrence of *this._target* in the invariants with t;
3. Substitute all occurrence of the heap with h^0.

$Pre_m Subst$ is obtained from Pre_m as follows:

1. Substitute all parameter name with $p_0^0, p_1^0, ..., p_n^0$ while maintaining the correct order;
2. Substitute all occurrence of *this* with t;
3. Transform each clause C where t is mentioned, in an implication *IsNotNull(t, T) ==>* C
4. Substitute all occurrence of the heap with h^0.

The refinement check at line 9 can now be expressed as:

```
assert (forall p^0_0, p^0_1, ..., p^0_n, t: ref, h^0: [ref,<x>name]x ::
    IsHeap(h^0) &&
    Pre_D Subst(t, p^0_0, p^0_1, ..., p^0_n, h^0) &&
    Inv_D Subst(d)
    ==> Pre_m Subst(t, p0_0, ..., pn_0, h^0));
```

Line 10 At line 10 post-condition refinement is checked. We must show that $Pre_D \wedge Inv_D \wedge Post_m \Rightarrow Post_D$. In addition to the renaming introduced for pre-conditions, we refer to post-state parameters of delegate and method as $p^1_0, p^1_1, ..., p^1_n$, to the result as r, and to the post-state heap as h^1. We define the post-conditions used for refinement checks as follows:

$Post_D Subst$ is obtained from $Post_D$ as follows:

1. Substitute all parameter name with $p^1_0, p^1_1, ..., p^1_n$ while maintaining the correct order;
2. Substitute all *old(...)* such that:
   - all occurrence of the heap is substituted with h^0;
   - all parameter name is substituted with $p^0_0, p^0_1, ..., p^0_n$;
3. Substitute all occurrence of *this* with t;
4. Substitute all occurrence of the heap with h^1;
5. Substitute all occurrence of *result* with r.

$Post_m Subst$ is obtained from $Post_M$ as follows:

1. Substitute all parameter name with $p^1_0, p^1_1, ..., p^1_n$ while maintaining the correct order;
2. Substitute all *old(...)* such that:
   - all occurrence of the heap is substituted with h^0;
   - all parameter name is substituted with $p^0_0, p^0_1, ..., p^0_n$;
3. Substitute all occurrence of *this* with t;
4. Transform each clause C where t is mentioned, in an implication *IsNotNull(t, T) ==>* C;
5. Substitute all occurrence of the heap with h^1.
6. Substitute all occurrence of *result* with r.

The refinement check at line 10 is translated as:

```
assert (forall p^0_0, p^0_1, ..., p^0_n, p^1_0, p^1_1, ..., p^1_n, t: ref, result,
    h^0: [ref,<x>name]x, h^1: [ref,<x>name]x ::
    IsHeap(h^0) && IsHeap(h^1) &&
    Pre_D Subst(t, p^0_0, p^0_1, ..., p^0_n, h^0) &&
    Inv_D Subst(d) &&
    Post_m Subst(p^0_0, p^0_1, ..., p^0_n, p^1_0, p^1_1, ..., p^1_n, t, r, h^0, h^1)
    ==> Post_D Subst(p^0_0, p^0_1, ..., p^0_n, p^1_0, p^1_1, ..., p^1_n, t, r, h^0, h^1));
```

Listing 3.1: Refinement checks

```
public delegate int IncHandler(int a, int b);
    requires a > 0;
    requires b < 0;
    invariant when { T; !target.K };

public class T {
    [Dependent(typeof(IncHandler))]
    public bool K;

    public int Inc(int aa, int bb)
        requires aa >= 0;
        requires bb <= 0;
        requires !this.K;
        ensures result > 0;
    { return 1; }
}

public class Program {
    static void Main(string![]! args) {
        T o = new T();
        IncHandler d = new IncHandler(o.Inc);
    }
}
```

The following example shows how refinement checks look like for the simple delegate instantiation of Listings 3.1:

```
1 assert (forall p_0^0: int, p_1^0: int, t: ref, h^0: [ref,<x>name]x ::
2     IsHeap(h^0) && p_0^0 > 0 && p_1^0 < 0 && !h^0[t, T.K]
3     ==> p_0^0 >= 0 && p_1^0 <= 0 && !h^0[t, T.K]);
4
5 assert (forall p_0^1: int, p_1^1: int, p_0^0: int, p_1^0: int,
6     t: ref, h^0: [ref,<x>name]x, h^1: [ref,<x>name]x, r: int ::
7         IsHeap(h^0) && IsHeap(h^1) && HeapSucc(h^0, h^1) &&
8         p_0^0 > 0 && p_1^0 < 0 && !h^0[t, T.K] && r > 0
9         ==> true);
```

3.1.3 Delegate Invocation in BoogiePL

The precondition of a delegate is directly checked by Boogie before the call to the *SafeInvoke()* method. We consider an invocation of the form:

`r = d(p_0, p_1, ..., p_n)`, where

- d is a possibly null delegate of type D;
- *p_0, p_1, ..., p_n* is the list of arguments for the invocation, the list typechecks correctly with respect to the signature of D;
- the type of r typechecks for an assignment with the return type of D;

$\mathcal{T}[r = d(p_0, p_1, ..., p_n)] =$

```
1 assert d != null;
2 s := call D.SafeInvoke(d, p_0, p_1, ..., p_n);
```

First we check that the delegate is not equal to null, then the delegate is invoked and the return value is assigned to a stack variable *s* representing *r*. In case the return type of the delegate is *void*, the assignment to *s* does not take place. While processing the *call* statement, Boogie will, as for all other methods invocation, assert the precondition of *SafeInvoke()*, save a copy of the pre-state heap, executing an havoc of the heap and then assume the post-condition. In its extended form the invocation as the following format:

```
1 assert d != null;
2 assert PreD(d, p_0, p_1, ..., p_n, Heap);
3 h := Heap;
4 havoc Heap;
5 havoc s;
6 assume PostD(d, p_0, p_1, ... p_n, Heap);
```

In this case *p_0, p_1, ..., p_n* represent the actual parameters, *result* represents the return value and *Heap* the heap. Please note that in this case we are not performing substitutions for parameters, return value and heap (while in case of refinement checks we need substitutions to obtain a match between the specification of the delegate and the one of the method).

3.1.4 Exposing an Object in BoogiePL

When exposing an object, we have to make sure that all delegates having an invariant on the object are also exposed. We consider an expose of the form:

`expose (o) { S }`, where

- o is a reference to some object of type T and T is a direct subclass of U.
- S is a block.

$\mathcal{T}[expose\ (o)\ \{\ S\ \}] =$

```
1 assert o != null;
2 assert o.owner == null || !(o.owner.inv <: o.ownerframe);
3 #foreach D s.t. D ∈ Dependents(T)
4  call UnpackDelegates(o, D);
5 o.localinv := U
6 S;
7 assert (forall p :: o.ownerRef == o && p.ownerFrame == T
8    ==> p.inv == typeof(p) && p.localinv == typeof(p);
9 assert InvT(o);
10 #foreach D s.t. D ∈ Dependents(T)
11  assert (forall d :: IsNotNull(d, D) && d.Target == o &&
12  d.inv == System.Object && d.localinv == typeof(d)
13  ==> InvD(d);
14 #foreach D s.t. D ∈ Dependents(T)
15  call PackDelegates(o, D);
16 o.localinv := typeof(o);
```

In the translation the additional instructions, with respect to the normal Spec# expose, are highlighted. Before exposing an object o, o is required to be non-null and either un-owned or consistent. After this is verified, we unpack (expose) all delegates having o as target and declaring an invariant for one or more fields of o. At this the normal expose continues and the *localinv* of the object is set to its base class; the body is then executed. After the execution it is asserted that all objects owned by o are valid, the invariant of o is asserted and we assert the invariant of all delegates that we previously exposed and we pack them. At this point the *localinv* field of the object is set to its type.

Note that even though the translation above refers to `expose (o) { S }`, the same principles are applied for `expose (o as T) { S }` and the additive versions `additive expose (o) { S }` and `additive expose (o as T) { S }`.

Unpacking and packing delegates was defined in [15] as:

$$
\begin{array}{ll}
1 & \#foreach\ \ D\ \ s.t.\ \ D \in Dependents(T) \\
2 & \quad let\ \ Dep_D := d\ \mid\ d.target = o \wedge d.inv = D; \\
3 & \quad foreach\ \ d \in Dep_D\ \texttt{\{ d.inv := object; \}} \\
4 & \quad \ldots \\
5 & \#foreach\ \ D\ \ s.t.\ \ D \in Dependents(T) \\
6 & \quad foreach\ \ d \in Dep_D\ \texttt{\{ d.inv := D \}}
\end{array}
$$

While translating these pseudo-code in BoogiePL, we realized that creating the set of line 2 and iterating over all its elements as done in lines 3 and 6 is not directly feasible. For this reason, we took a different approach and used two procedures. The advantage of using procedures is that we do not need to specify their body - the actual iteration - if we can specify a sufficiently meaningful post-condition. The idea for the first one, *UnpackDelegates*, is to specify in the post-condition that all enabled delegates having o as target are exposed:

```
1  procedure PackDelegates(o: ref, D: name);
2    modifies Heap;
3    ensures (forall d: ref, F: name :: F != inv || !old(IsNotNull(d, D)) ||
4        !old(Heap[d, System.Delegate._enabled]) ||
5        old(Heap[d, System.Delegate._target] != o) ||
6        !old(Heap[d, inv] == System.Object && Heap[d, localinv] == typeof(d))
7        ==> old(Heap[d, F]) == Heap[d, F]);
8    ensures (forall d: ref :: old(IsNotNull(d, D)) &&
9        old(Heap[d, System.Delegate._enabled]) &&
10       old(Heap[d, System.Delegate._target] == o) &&
11       old(Heap[d, inv] == System.Object &&
12       Heap[d, localinv] == typeof(d)) ==> Heap[d, inv] == typeof(d));
13   free ensures HeapSucc(old(Heap), Heap);
```

At line 2 we specify that the procedure will modify the heap. We then ensure (3-7) that all delegates having a target object different from o are not modified. We then ensure (8-12) that all delegates having o as target are indeed exposed. Finally we ensure that the resulting heap is a well formed successor of the pre-state heap. The procedure used to pack back the delegates we previously unpacked, is defined as follows:

```
1  procedure UnpackDelegates(o: ref, D: name);
2    modifies Heap;
3    ensures (forall d: ref, F: name :: F != inv || !old(IsNotNull(d, D)) ||
4        !old(Heap[d, System.Delegate._enabled]) ||
5        old(Heap[d, System.Delegate._target] != o) ||
6        !old(Heap[d, inv] == typeof(d) && Heap[d, localinv] == typeof(d))
7        ==> old(Heap[d, F]) == Heap[d, F]);
```

```
8      ensures (forall d: ref :: old(IsNotNull(d, D)) &&
9         old(Heap[d, System.Delegate._enabled]) &&
10        old(Heap[d, System.Delegate._target] == o) &&
11        old(Heap[d, inv] == typeof(d) && Heap[d, localinv] == typeof(d))
12        ==> Heap[d, inv] == System.Object);
13     free ensures HeapSucc(old(Heap), Heap);
```

This ensures that whenever we potentially have delegates with o as target object and declaring an invariant on o, program invariant **P1** is not violated.

3.1.5 Delegate Subtyping in BoogiePL

In BoogiePL we are free to encode delegate sub-typing[2] as real class sub-typing. The type definition axioms in case of sub-typing are obtained as follows:

$\mathcal{T}$[*delegate void BaseDel(); delegate void SubDel() : BaseDel;*] =

```
1 axiom BaseClass(BaseDel) == System.MulticastDelegate &&
2     AsDirectSubClass(BaseDel, BaseClass(BaseDel)) == BaseDel;
3 axiom BaseClass(SubDel) == BaseDel &&
4     AsDirectSubClass(SubDel, BaseClass(SubDel)) == SubDel;
5 axiom (forall U: name :: { U <: SubDel } U <: SubDel ==> U == SubDel);
```

The first axiom expresses the fact that *BaseDel* is a subclass of *MulticastDelegate*. The second axiom expresses the subtype relationship between *BaseDel* and *SubDel*. Finally the last axiom says that there is no class that extends *SubDel*.

3.2 Disabling Delegates

3.2.1 Ownership Based Disabling

The technique for disabling delegates described in [15] relies on two facts. First a delegate can only be invoked if it is peer-consistent: its *inv* field must be set to the type of the delegate. Second delegates are disabled by setting their *inv* field to *Object* and un-owning them. When translating this approach to a working implementation however, we realized that the solution is not applicable to Spec# for two main reasons. The first one is a conceptual issue related to ownership specification and non-nullable types. The second issue due to the lack of ownership transfer for already owned objects in the current implementation of Spec#. Even if we could find a solution for implementing ownership transfer, the program of Listing 3.2 shows that ownership based disabling is not applicable.

Listing 3.2: non null delegate]Disabling a [Rep] non null delegate

```
public delegate void MyDelegate();
    invariant when { A; target.IsEnabled; }

public class A {
    [Dependent(typeof(MyDelegate))] public int IsEnabled;
```

[2]Note that in C# delegate subtyping is not allowed. In Chapter 5 we explain how we implemented delegate subtyping in Spec#

```
}

public class Ow {
    [Rep] MyDelegate! d;
    [Rep] A! a;

    public void Break() {
        expose (this)
            disable (a for Del);
    }
}
```

Method *Break* disables all delegate having a as target. Let us suppose that d has indeed a as target object. The disable statement would un-own the delegate pointed to by d. Unfortunately field d must point (by its specification) to a non-null delegate owned by its declaring object. The contradiction is in this case clear.

In conclusion, the original approach suffers from the fact the enabled state of a delegate is encoded, together with the validity state, in the *inv* field of the object. In the next section we present a solution for handling disabling of delegates which solves the issues described above by separating the two states.

3.2.2 Disabling Delegates with the Enabled Field

Instead of using *inv* to express the fact that a delegate is enabled or not, we explicitly add a new *_enabled* field to the delegate; this effectively allows to separate the two concepts. The idea is that only delegates with the *_enabled* field set to true can be invoked. In order to prevent the invocation of disabled delegates, we add a default pre-condition to delegate contracts: `requires this._enabled`. This solves the invocation problem. We are left with defining how a delegate can be disabled.

Let us start by redefining delegate invariants. Up to now the invariant of a delegate type D was only composed by the user-defined invariant expressed in `invariant when { ... }` clauses, and previously name Inv_D. In order to apply the new approach it is necessary to extend the user-defined invariant with an additional expression; for this reason from now on the user-defined part of the invariant will be explicitly referred to by Inv_D^{Usr}.

For every delegate type D, the delegate invariant Inv_D is redefined as

$$Inv_D := this._enabled \Rightarrow Inv_D^{Usr}$$

In other words the new invariant states that the *_enabled* field implies the user-defined part of the invariant. With this new definition the following program invariant is obtained:

Program Invariant 5. *The Enabled State*

$$P5 : (\forall d, T \cdot d.inv <: T \wedge d._enabled \Rightarrow Inv_D^{Usr}(d))$$

Thanks to the new definition of Inv_D, program invariant **P1** is still valid. Disabling a delegate is now possible by setting its *_enabled* field to false. By **P5** this implies that a delegate invariant Inv_D^{Usr} does not have to hold if the *_enabled* field is set to false; the `disable (o for D)` statement sets the *_enabled* field of every delegate of type D with

target object o to false. This ensures that (1) disabled delegates can no longer be invoked and (2) the user-declared invariant Inv_D^{Usr} does not need to hold in order for the delegate, its peers and its owner to be valid. This extension allows disabling delegates and ensures that disabled delegates cannot be invoked.

Once a delegate has been disabled, it is impossible to enable it again. As explained in [15] it is in fact still possible to create a new instance of the delegate with same target and method. To ensure this behavior, the *_enabled* field cannot be directly modified. The field is set to *true* during the instantiation of the delegate; this is possible because before instantiation all the user-declared invariants are checked. Moreover the field can only be set to false by a call to `disable (o for D)`.

Frame Conditions for Disabling Delegates

One last note about disabling delegates concerns frame conditions. Since *System.Delegate._enabled* is not directly accessible by the user, the only way to express the frame condition of a method that disables delegates would be to use `modifies p.**`, where p is a peer of the object o of the `disable(o for D)` statement. We consider forcing the programmer to use such weak frame condition for expressing disabling of delegate to be too restrictive. For this reason *System.Delegate._enabled* is not included in modifies star and we provide a new modifies clause: `modifies p.***`. This clause allows to modify the *System.Delegate._enabled* field of all delegates which are peers of p.

3.2.3 Disabling Delegates in BoogiePL

The axiom expressing the invariant of a delegate, which was previously of the form:

```
1 axiom (forall oi: ref, h: [ref,<x>name]x :: { h[oi, inv] <: Del }
2     IsHeap(h) && h[oi, inv] <: Del &&
3     h[oi, localinv] != BaseClass(Del) ==> InvD(oi));
```

becomes:

```
1 axiom (forall oi: ref, h: [ref,<x>name]x :: { h[oi, inv] <: Del }
2     (IsHeap(h) && h[oi, inv] <: Del &&
3     h[oi, localinv] != BaseClass(Del)
4     ==> h[oi, System.Delegate._enabled])
5         ==> InvD^Usr(oi));
```

In order words when delegate is valid, e.g. *d.inv = typeof(d)*, then either the delegate is enabled and the user declared part of the invariant holds, or the delegate is disabled. This said we add a new, default precondition to the *SafeInvoke()* method as follows:

```
requires Heap[this, System.Delegate._enabled];
```

The consequences of the new definition of the invariant axiom and the additional precondition is that we now that when a delegate can be invoked (the precondition of *SafeInvoke()* holds) only if it is enabled. Moreover we also know, thanks to the peer-consistency precondition, that the delegate must also be valid. This implies by program invariant **P5** that user-declared invariant $InvD^{Usr}$ holds.

We provide a `disable(o for D)` as in the original work. Moreover we also need to provide a way to query a delegate for its enabled state: `d.IsEnabled` returns the boolean enabled

state. Finally we need to provide a way for asserting that there exist no delegate of some type having some object o as target. This is achieved by invoking `DelegateReferencesHolder.NoActiveDelegate(o, typeof(D))`. The method returns a boolean value, *true* if no enabled delegate of type D has o as target.

We now describe the translations of these three additional elements.

`d.IsEnabled`, where d is of type *Delegate*.

`NoActiveDelegate(o, D)`, `disable(o for D)`, where

- o is a non-null reference to an object
- D is a type

$\mathcal{T}[d.IsEnabled] =$

```
1 Heap[d, System.Delegate._enabled]
```

This construct can be used in contracts and expressions. When used in an assertion for example we obtain the following:

```
assert Heap[d, System.Delegate._enabled];
```

$\mathcal{T}[NoActiveDelegate(o, D)] =$

```
1 (forall d: ref :: IsNotNull(d, D) && Heap[d, System.Delegate._target] == o
2    ==> !Heap[d, System.Delegate._enabled]);
```

NoActiveDelegate, as with the case for *IsEnabled*, can be used in contracts and expression.

$\mathcal{T}[disable(o\ for\ D)] =$

```
1 call DisableDelegates(o, D);
```

where *DisableDelegates* is defined as:

```
1  procedure DisableDelegates(o: ref, D: name);
2    requires o != null;
3    requires Heap[o, ownerFrame] == PeerGroupPlaceholder ||
4      !(Heap[Heap[o, ownerRef], inv] <: Heap[o, ownerFrame]) ||
5      Heap[Heap[o, ownerRef], localinv] == BaseClass(Heap[o, ownerFrame]);
6    modifies Heap;
7    ensures (forall d: ref, F: name :: { Heap[d, F] }
8      F != System.Delegate._enabled || old(!IsNotNull(d, D)) ||
9      old(Heap[d, System.Delegate._target] != o)
10     ==> old(Heap[d, F]) == Heap[d, F]);
11   ensures (forall d: ref :: IsNotNull(d, D) &&
12     old(Heap[d, System.Delegate._target] == o)
13     ==> !Heap[d, System.Delegate._enabled]);
14   free ensures HeapSucc(old(Heap), Heap);
```

With respect to the definition in [15], we use a procedure for implementing the disable statement. The precondition (2-5) for disabling delegates is for the target to be non-null and its owner to be exposed. This remains unchanged with respect to the previous definition. At lines 7-10 we ensure that all delegate with target object different from o are not disabled. At

Listing 3.3: A first approach towards frame conditions for delegates

```
public delegate void MyDelegate()
   modifies this.Target.*;

public class A {
    [Rep] public MyDelegate! del;
    [Rep] public C! c;
    [Rep] public D! d;

    public void DoSome(D! otherD)
        requires del.IsEnabled;
        modifies otherD.j;
    {
        expose (this) {
            d.j = 1;
            otherD.j = 2;
            del();
            assert d.j == 1;         // this assertion fails
            assert otherD.j == 2;    // this assertion holds
        }
    }
}
```

lines 11-13 we ensure that all delegate having *o* as target are indeed disabled. We conclude by ensuring that the post-state heap is a successor of the pre-state heap.

3.3 Frame Conditions

In [15] it was stated that modifies clauses can be de-sugared into post-conditions. However, given the guidelines for using delegates, this would break the encapsulation property for delegates.

3.3.1 Encapsulation

Without the ability to provide additional information on the type of the target, the only possible solution for modifies clauses in delegate is to allow all possible fields of the target to change. This can be accomplished with a `modifies this.Target.*` clause. Consider the following example:

Listing 3.3 shows a scenario where no information about the target of *del* is known; this implies that the assertion after the invocation of *del* will fail even if the target of the delegate is *c*. Also note that the assertion concerning *otherD*, still holds after the invocation. This is because *otherD* has a different owner. As shown in Listings 3.4, the situation in the example can be improved by providing additional information on the type of the target.

The additional requires clause, in combination with the modifies clause of the delegate, ensure that only fields of objects of type C are modified by a call to *del*. The example suggests that delegates cannot always be loosely coupled if we want to prove useful information in a program. Leaking some information about the target of the delegate can be considered an

Listing 3.4: Frame conditions with additional information about the target

```
public void DoSome()
    requires del.IsEnabled;
    requires del.Target is C;
{
    expose (this) {
        d.j = 1;
        del();
        assert d.j == 1;
    }
}
```

acceptable solution.

3.3.2 The Type of the Target

The question at this point is whether the situation depicted above can be improved. In this example it would be useful to express a frame condition of the form:

```
public delegate void MyDelegate()
    modifies Target.i;
```

As a first step in solving this problem, we need to provide additional information in the modifies clause, so that the type of the target can be checked to ensure that it actually defines the fields present in the condition. We extend the contract of delegates to include a special modifies clause for their target objects. The new clause has the form:

```
modifies when { T; target.P }
```

where *T* is a type, and *P* is a field of *T* or one of the special placeholders ****, *** or *0*. This modifies clause is de-sugared into

```
modifies ((T)Target).P
```

The additional syntactical information allows us to perform type-checking and ensure that the field specified in the clause is defined in *T*.

To solve the second part of the issue - targets limited to specific types - we define a *modifies when* clause to be conditional to the type it specifies. In other words `modifies when { T; target.P }` means that the target is only allowed to be modified when it is of type *T*. This allows to define multiple *modifies when* clauses for different target types.

The question on what should happen when the type of the target is unknown remain however open. The situation suggests a default frame condition for delegate targets. The are three possible options here: we can define that the default behavior to be *Target.***, *Target.** or *Target.0*. Since it is impossible to make this choice for the general case, we allow the user to specify the default behavior the *ModifiesDefault* attribute. The attribute takes one of the three following arguments:

1. `[ModifiesDefault(Modifies.Peers)]`
2. `[ModifiesDefault(Modifies.Target)]`

3. [ModifiesDefault(Modifies.None)]

The first condition expresses the fact that all fields of all peers of a delegate can change. The second condition limits the changes to fields of the target object only. This condition is equivalent to the first one when there is no knowledge about the target object. Finally the third condition is useful in situations where the programmer requires a delegate to have no side effects on its target object. When the attribute is omitted, the default value is *Modifies.Target*.

With respect to the initial example, the delegate declaration becomes:

```
ModifiesDefault(Modifies.Target)]
delegate void MyDelegate()
    modifies when { C; target.i };
```

The frame condition can be interpreted like shown below:

```
delegate void MyDelegate()
    modifies (Target is C) ==> ((C)Target).i;
    modifies !(Target is C) ==> Target.*;
```

Please note that in the actual implementation, we cannot de-sugar the default condition in an additional *modifies* clause. For this reason we need to keep the *ModifiesDefault* attribute and handle it in Boogie.

3.3.3 Subtyping

Spec# does not allow frame conditions to be refined in method overrides (or in interface implementations). We follow the same convention for normal *modifies* clauses on the parameters. With respect to a delegate defined in a library however, the programmer usually knows more information about possible target objects. Similarly to refinement of invariants in [15], we allow *modifies when* clauses to be refined in delegate subtypes. The first point to note is that the frame condition of a delegate must be stronger or equal to the frame condition of its base delegate; *modifies when* are propagated to delegate subtypes. All clauses for a type T can then be overridden by declaring *modifies when* clauses for T in the subtype. Finally default frame conditions can be refined as well by declaring the *ModifiesDefault* attribute in delegate subtypes.

Consider the following delegate declaration:

```
[ModifiesDefault(Modifies.None)]
delegate void BaseDel();
    modifies when { C; target.* };
    modifies when { D; target.j };
```

When a delegate extending *BaseDel* has no *modifies when* clause, the frame condition is entirely propagated. If the extending delegate however declares additional *modifies when* clauses for a type already defined in the base delegate, then the clauses of the base delegate for that type are not propagated.

```
delegate void SubDel() : BaseDel;
    modifies when { C; target.i };
```

In the example above, the complete frame condition of *SubDel* is:

```
[ModifiesDefault(Modifies.None)]
delegate void SubDel() : BaseDel;
    modifies when { C; target.i };
    modifies when { D; target.j };
```

Finally, as given by the refinement principle, frame conditions cannot be weakened. The following example is indeed incorrect:

```
[ModifiesDefault(Modifies.Target)]
delegate void InvalidDel(): BaseDel;
    modifies when { D; target.* };
```

3.3.4 Frame Conditions in BoogiePL

The encoding of frame conditions for modifies clauses on the arguments of delegates follows the normal encoding for methods. For *modifies when* clauses however, the translations are slightly different and need to be defined.

Consider the following example:

```
public void Foo()
    modifies this.i;
```

The frame condition of *Foo* is translate in BoogiePL as:

```
1 ensures (forall o: ref, f: name :: { Heap[o, f] }
2    IncludeInMainFrameCondition(f) && o != null && old(Heap)[o, allocated] &&
3    (old(Heap)[o, ownerFrame] == PeerGroupPlaceholder ||
4       !(old(Heap)[old(Heap)[o, ownerRef], inv] <: old(Heap)[o, ownerFrame]) ||
5       old(Heap)[old(Heap)[o, ownerRef], localinv] == BaseClass(old(Heap)[o, ownerFrame]))
6       && old(o != this || f != C.i)
7       && old(o != this || f != exposeVersion)
8       ==> old(Heap)[o, f] == Heap[o, f]);
```

Lines 1-5 are standard for all frame conditions and they limit the objects in the quantification to object whose owner is expose (or are un-owned) and the fields to fields included in modifies star and in main frame conditions. Lines 7, also standard, allows the expose version of the object to change while line 8 states that all fields non included in *modifies* clauses remain unchanged. The most interesting part for us is given in line 6, where field i of objects of type C is allowed to be modified. We refer to the condition at line 6 as the *modifiesContrib*, that is, the modifies contribution, of `modifies this.i` for the frame condition of *Foo*. The translations defined below are in fact expressed with respect to the normal *modifiedContrib* for *modifies* clauses.

We consider a *modifies when* clause of the form `modifies when { T; target.P }` as defined in the previous section.

$\mathcal{T}$[*modifies when* { *T; target.P* }] =

```
1 typeof(Heap[this, System.Delegate._target]) <: T ==> 𝒯[modifies target.P]
```

This is valid for single fields, * and **. In case of *target.0* no condition is emitted because that is the default behavior in the BoogiePL frame condition. As an example consider the following declaration:

```
[ModifiesDefault(Modifies.None)]
delegate void MyDelegate()
    modifies when { C; target.i };
```

The frame condition of *MyDelegate* is translated in:

```
1 ensures (forall o: ref, f: name :: { Heap[o, f] }
2     IncludeInMainFrameCondition(f) && o != null && old(Heap)[o, allocated] &&
3     (old(Heap)[o, ownerFrame] == PeerGroupPlaceholder ||
4     !(old(Heap)[old(Heap)[o, ownerRef], inv] <: old(Heap)[o, ownerFrame]) ||
5     old(Heap)[old(Heap)[o, ownerRef], localinv] == BaseClass(old(Heap)[o, ownerFrame])) &&
6     old(o != Heap[this, System.Delegate._target] ||
7         !(typeof(Heap[this, System.Delegate._target]) <: C) || f != C.i) &&
8     old(o != Heap[this, System.Delegate._target] || f != exposeVersion)
9     ==> old(Heap)[o, f] == Heap[o, f]);
```

The modifies contribution of lines 6 and 7 is indeed the translation of a normal modifies clause with the addition of the implication for the target type. In fact:

```
1 old(o != Heap[this, System.Delegate._target] ||
2         !(typeof(Heap[this, System.Delegate._target]) <: C) || f != C.i)
```

is equivalent to:

```
1 old(typeof(Heap[this, System.Delegate._target]) <: C ==>
2     o != Heap[this, System.Delegate._target] || f != C.i)
```

Default Frame Conditions

Default clauses also behave in a similar manner:

$\mathcal{T}[ModifiesDefault.Q] =$

```
1 #forall T_i s.t. T_i is the type of at least one modifies when clause
2     (!(typeof(Heap[this, System.Delegate._target]) <: T_0) &&
3      !(typeof(Heap[this, System.Delegate._target]) <: T_1) && ···
4      !(typeof(Heap[this, System.Delegate._target]) <: T_n))
5     ==> T[modifies target.P]
```

Where Q is `Modifies.Peers` or `Modifies.Target` and P is the corresponding modifies contribution of respectively *Target.*** or *Target.**.

Refinement of Frame Conditions

We implemented refinement checks for frame conditions during the instantiation of new delegates. With respect to the translation of `d = new D(o.m)` we add the following assertions:

```
1 assert (forall p_0^0, p_1^0, ..., p_n^0, o: ref, f: name, t: ref , h: [ref,<x>name]x, ::
2     IsHeap(h)
3     ==> ModifiesContrib_D(p_0^0, p_1^0, ..., p_n^0, o, f, t, h)
```

```
4         ==> ModifiesContrib_m(p^0_0, p^0_1, ..., p^0_n, o, f, t, h));
5 #foreach E s.t. E <: D
6     assert (forall p^0_0, p^0_1, ..., p^0_n, o: ref, f: name, t: ref , h: [ref,<x>name]x, ::
7         IsHeap(h)
8         ==> ModifiesContrib_E(p^0_0, p^0_1, ..., p^0_n, o, f, t, h)
9             ==> ModifiesContrib_D(p^0_0, p^0_1, ..., p^0_n, o, f, t, h));
```

where $p_0^0, p_1^0, ..., p_n^0$ are the parameters, o and f iterate over all possible objects and fields, t represents the target object and h the heap.

These assertions are obtained by removing the redundant parts in the frame conditions of delegates and target methods; this allows to have a simplified (but still sufficient) proof obligation. At lines 1-4 we verify that the set of all fields possibly modified according to the frame condition of the delegate, includes the set of all fields possibly modified by the frame condition of the target method. At lines 5-9 we perform refinement checks to verify if the frame condition of a delegate, is indeed a subset of the frame condition of its base delegate.

$ModifesContrib_D$ is obtained from the modifies contribution of the frame condition of the delegate as follows:

1. Substitute all parameter name with $p_0^0, p_1^0, ..., p_n^0$ while maintaining the correct order;
2. Substitute all occurrence of *Heap[this, System.Delegate._target]* with t;
3. Substitute all occurrence of *Heap* with h.

$ModifiesContrib_m$ is obtained from the modifies contribution of the frame condition of the target method as follows:

1. Substitute all parameter name with $p_0^0, p_1^0, ..., p_n^0$ while maintaining the correct order;
2. Substitute all occurrence of *this* with t;
3. Substitute all occurrence of *Heap* with h.
4. Add *!(typeof(t) <: T)* where necessary in order to match the modifies contribution of the delegate.

For the example in Listing 3.5 the following refinement checks are generated:

```
1 assert (forall o: ref, f: name, i: int, h: [ref,<x>name]x, t: ref ::
2     IsHeap(h)
3     ==> old(o != t || !(typeof(t) <: A) || f != A.i)
4         ==> old(o != t || !(typeof(t) <: A) || f != A.i));
5 assert (forall o: ref, f: name, i: int, h: [ref,<x>name]x, t: ref ::
6     IsHeap(h)
7     ==> old(o != t || !(typeof(t) <: DeclType(f)) || !(typeof(t) <: A)
8         || !IncludedInModifiesStar(f))
9         ==> old(o != t || !(typeof(t) <: A) || f != A.i));
```

Listing 3.5: Refinement checks for frame conditions

```
[ModifiesDefault(Modifies.None)]
delegate void BaseDel();
    modifies when { A; target.* };

delegate void SubDel() : BaseDel;
    modifies when { A; target.i };

public class A {
    public int i;
    public void Foo(B! b)
        modifies this.i;
    { }
}
```

CHAPTER 4

Extensions

In Chapter 3 we analyzed the implementation of the verification methodology for singlecast delegates with a target object. Delegates however are not limited to this and our approach needs to consider delegates instantiated with static methods, multicast delegates and events as well.

4.1 Delegates with Static Methods

The verification of delegates instantiated with static methods requires less overhead than the verification of normal delegates: delegate invariants are omitted as there is no target object.

We examine the situation where a static method has a pre-condition on the (static) state of a class. In Listings 4.1 a delegate is used in combination with the factory pattern. The idea is that an instance of *ProductCreator* allows creating new products. This delegate points to one of the creation methods of *Factory* and the pre-condition for these methods is for the factory to be able to create products in the first place: *Factory.CanCreateProducts*. As the example shows, it is possible to directly express the pre-condition of the target method in the pre-condition of the delegate. With instance target methods, such pre-condition could not be stated because, in order to characterize an object, both the type of the object and a reference to the object itself are needed. With delegates the reference is available but the type information is missing. In case of static methods however the type information is implicit and no reference is needed. This allows us to express pre-conditions on static members.

One last question remains open in terms of modularity. By stating the requirement on the static target directly in the delegate, it becomes impossible to reuse a delegate defined in a library in other situations. For this reason a possible extension to the current methodology consists in allowing delegates (and delegate subtypes) to declare static invariants [14]. The definition of the delegate in the example above would become:

Listing 4.1: Static pre-conditions

```
delegate IProduct ProductCreator(ProductInfo info);
    requires Factory.CanCreateProducts;

class Factory {
    public static bool CanCreateProducts;
    private static FactoryState state;

    public static IProduct CreateConsumableProduct(ProductInfo info)
        requires Factory.CanCreateProducts;
    {
        IProduct product;
        /* create a product based on state and info */
        return product;
    }
}
```

```
delegate IProduct FactoryProductCreator(ProductInfo info)
    : ProductCreator;
    invariant when { Factory; Factory.CanCreateProducts };
```

Note that in the current implementation is it already possible to declare static delegate invariants. However, due to the current state of implementation of static invariants in Spec#, using static delegate invariants is not sound.

4.1.1 Instantiation of Delegates with Static Methods in BoogiePL

We consider an instantiation of a delegate of the form:

`d = new D(m)`, where

- *d* is a variable of type *D;*
- *o* is a possibly null reference to an object of type *T*;
- *m* is the name of a static method.

$\mathcal{T}[d = new\ D(m)] =$

```
1  havoc m;
2  havoc d;
3  assume !d.allocated && d != null && typeof(d) == D;
4  assume d.owner.ref = d && d.owner.frame = PeerGroupPlaceholder;
5  assert d != null;
6  call D..ctorSystem.ObjectSystem.IntPtr(d, null, m);
7  d.inv := D; d.localinv := typeof(d);
8  assert (forall p0^0, p1^0, ..., pn^0, h^0 , : [ref,<x>name]x ::
9      IsHeap(h^0) &&
10     Pre_D Subst(p0^0, p1^0, ..., pn^0, h^0)
11     ==> Pre_M Subst(p0_0, ..., pn_0, h));
```

```
12 assert (forall $p_0^0, p_1^0, ..., p_n^0, p_0^1, p_1^1, ..., p_n^1$, result,
13     h: [ref,<x>name]x, h^1: [ref,<x>name]x ::
14     IsHeap($h^0$) && IsHeap($h^1$) &&
15     $Pre_D Subst(p_0^0, p_1^0, ..., p_n^0, h^0)$ &&
16     $Post_M Subst(p_0^0, p_1^0, ..., p_n^0, p_0^1, p_1^1, ..., p_n^1, r, h^0, h^1)$
17     ==> $Post_D Subst(p_0^0, p_1^0, ..., p_n^0, p_0^1, p_1^1, ..., p_n^1, r, h^0, h^1)$);
```

The instantiation of a delegate with a static method is a subset of the instantiation of a normal delegate. Statements and assertion concerning the target object are, in fact, omitted. Refinement checks are also modified so that they do not consider the target object.

4.2 Multicast Delegates

Multicast delegates are more complex to handle with respect to singlecast delegates. Consider that a single invocation of a multicast delegate (a call to the *SafeInvoke()* method) results in the sequential invocation of all its targets. Given this behavior there are a number of situations that could go wrong. We start the discussion by considering the conditions a multicast delegate must always satisfy. Once this base concepts are clear, we discuss how a multicast delegate is encoded, how the enabled state of a multicast delegate is defined and how a multicast delegate is disabled.

4.2.1 Stability requirements

A first, general, observation is that a post-condition must hold when the pre-condition is valid; this must remain true even in case of multiple targets.

Listing 4.2: Post-conditions of multicast delegates

```
delegate void AddHandler(ref int i)
    requires true;
    ensures i == old(i) + 1;

public abstract void Add(ref int i);
    requires true;
    ensures i == old(i) + 1;
```

Consider the example in Listing 4.2; method *Add()* is used as target for delegate *AddHandler*. Everything works fine when we have a singlecast delegate, but think about the situation where an instance d of *AddHanlder* is a multicast delegate with method *Add()* twice in the invocation list. We could start by invoking d as follows: `int i = 0; d(ref i);`

The first call to *Add()* increments i by one, we obtain $i = 1$; the second call in the invocation list also increments i by 1 and we obtain $i = 2$: from the perspective of the caller of d, the invocation does not respect the contract: the pre-condition is respected but, given an initial value of 0 we obtain 2, which does not respect the post-condition of the delegate. From this example we can extrapolate the first necessary condition for multicast delegates:

Condition 4.1 (External Stability). *The post-condition of a multicast delegate must remain valid after any number of invocations of its target methods. This ensures that the post-condition of the multicast delegate is satisfied.*

The second concern with multicast delegates is to ensure that the pre-condition remains

Listing 4.3: Pre-conditions of multicast delegates

```
delegate void DivisionHandler(ref int i);
    requires i > 0;
    ensures i <= old(i);

public abstract void Divide(ref int i);
    requires i > 0;
    ensures i <= old(i);
```

satisfied in between calls to the targets in the invocation list.

Consider the example in Listing 4.3. Let us say, for example, that *Divide()* returns the result of the integer division by 2 of the argument; let us also imagine that, once again, a delegate d of type *DivisionHandler* is a multicast delegate containing multiple times *Divide()*. We start by invoking d with 2, after the first call $i = 1$, after the second call $i = 0$, the third call fails because of a pre-condition violation. The second condition states the following:

Condition 4.2 (Internal Stability)**.** *The post-condition of a multicast delegate must imply its pre-condition. This ensures that every call to a target method leaves the pre-condition valid.*

These two conditions are sufficient to ensure correctness with respect to pre- and post-conditions, but what about delegate invariants? We already know that a delegate invariant holds at invocation when the pre-condition of the delegate is satisfied. Similar to Condition 4.2, we must now also ensure that the invariant holds after the invocation of one or more targets. This leads to the third necessary condition:

Condition 4.3 (Stability of Invariants)**.** *All methods in the invocation list of a multicast delegate must be enabled before its invocation and its invariant cannot be violated by any of the methods in its invocation list.*

The last condition is related to peer-consistency. Remember that we still need the delegate and its target to be peers so that when the target is invoked we know that its owner is exposed. This remains valid for multicast delegates as well and leads to the following condition:

Condition 4.4 (Stability of Ownership)**.** *The owner of any delegate (and target object) present in the invocation list of a multicast delegate is not allowed to change during the invocation of the multicast delegate.*

We call these four conditions the stability requirements of a multicast delegate; the term indicates that for a multicast invocation to be correct, heap and parameters must remain stable with respect to pre- and post-conditions of the delegate. We claim that these conditions are sufficient to ensure the correct execution of a multicast delegate.

4.2.2 Encoding Multicast Delegates

The conditions expressed above allow us to encode a multicast delegate in the same exact way as a singlecast delegate, at least as long as disabling is ignored. Since the conditions ensure that any number of invocations of possible target methods respect the specification of the delegate, we can in fact ignore the invocation list. The situation however becomes more

complex when disabling in taken into consideration. We start by describing the approach we implemented for multicast delegates; we then explain how this could be extended to increase the information about a multicast delegate.

Current Approach

The current approach consists in reducing a multicast delegate to singlecast. For invocation, if the multicast delegate respects the four stability conditions described above, this is not a problem; but what about disabling? Consider a multicast delegate m containing, in its invocation list, two singlecast delegates pointing to $a.Foo$ and $b.Bar$ respectively. When the two singlecast delegates are combined in m we require them to be both enabled. This ensures that m is enabled as well. We specify the target object of m to be either a or b: this is done with a postcondition for the *Delegate.Combine* method of the form: `ensures result.Target == a || result.Target == b`.

The question is now what happens when a delegate, say a, is disabled by a `disable (a for D)` statement, translated in BoogiePL in a call to *DisableDelegates*. Since the theorem prover cannot show that the target of m is different from a, the enabled state of m becomes unknown. This means that both of the following assertion will fail:

```
assert m.IsEnabled;
assert !m.IsEnabled;
```

Now let us consider this from the point of view of an invocation of m. Since m contains a delegate which is not enabled, the result we need to achieve is for the pre-condition of m to fail. Remember that *m.SafeInvoke()*, requires m to be enabled; since it is now impossible to prove that m is enabled, the pre-condition of the method will fail. So even though we cannot prove that m is, in fact, not enabled, we still have a sound system.

The specification of *Delegate.Combine()* is given below:

```
public static Delegate Combine (System.Delegate a, System.Delegate b);
   requires ((object)a) != null && ((object)b) != null ==>
      (Owner.None(a) && Owner.None(b)) || Owner.Same(a, b);
   requires ((object)a) != null && ((object)b) != null ==>
      ((object)a.GetType()) == ((object)b.GetType());
   requires ((object)a) != null ==> a.IsEnabled;
   requires ((object)b) != null ==> b.IsEnabled;
   ensures ((object)a) == null && ((object)b) == null ==>
      ((object)result) == null;
   ensures ((object)a) == null && ((object)b) != null ==>
      ((object)result) == (object)b;
   ensures ((object)a) != null && ((object)b) == null ==>
      ((object)result) == (object)a;
   ensures ((object)a) != null && ((object)b) != null ==>
      result != null &&
      ((object)result.GetType()) == ((object)a.GetType()) &&
      ((object)result.GetType()) == ((object)b.GetType()) &&
      (Owner.None(a) && Owner.None(b)
         ==> Owner.None(result)) &&
      (Owner.Same(a, b)
         ==> Owner.Same(result, a) && Owner.Same(result, b));
```

The first two requires clauses state that the arguments must be peers (or have no owner) and be of the same type. We then require both delegates to be enabled. The method ensures that if one of the delegates is null, then the other is returned. In case they are both non-null, then the resulting delegate is peer with the arguments and its type is the same.

The drawback of the current approach is that any `disable (o for D)` statement disables all multicast delegates of type D which are peers of o, even if their invocation list does not contain any singlecast delegate with target object o. The advantage, on the other side, is that this approach requires a minimum amount of specification from the user and allows multicast delegates to be handled like singlecast delegates.

In the next two sections we show how these limitations can be overcome. Please note that the two approaches described below are currently not implemented.

Extension 1: Keeping Track of the Targets

The compromise between expressive power and compact specification might lead us to require a more fine grained solution for multicast delegates. The first extension consists in keeping track of all targets of a multicast delegate. This can be accomplished by extending *System.Delegate* with the additional field *object[]! Targets*. In order to handle this additional specification we extend the *Construct()* and *Delegate.Combine()* methods by adding the following specification:

```
public Del! Construct(object @object, IntPtr method)
    ensures object != null ==> result.Target == object &&
        result.Targets.Length == 1 && result.Targets[0] == object;
    ensures object == null ==> result.Target == null &&
        result.Target.Length == 0;
    ...

public static Delegate Combine(Delegate a, Delegate b)
    ensures ((object)a) != null && ((object)b) != null ==>
        result.Targets.Length == a.Targets.Length + b.Targets.Length &&
        forall { int i in (0:a.Targets.Length);
            result.Targets[i] == a.Targets[i] } &&
        forall { int j in (a.Targets.Length:b.Targets.Length);
            result.Targets[j] == b.Targets[j-a.Targets.Length] };
    ...
```

The *Construct()* method ensures that the *Targets* array of the resulting delegate contains a single object, namely the target of the delegate. *Combine()*, on the other side, ensures that the *Targets* of the new delegate are the concatenation of the targets of the two arguments.

This approach is more fine grained with respect to what presented before but still its expressive power is limited. It is insufficient in fact to specify the equality operator for multicast delegates which requires to keep track of target objects and methods.

As additional pre-condition for the disable statement `disable (o for D)` we require all multicast delegates not to have o in their *Targets* array. This allows us to keep the same semantics for disabling delegates that we used in the previous chapter. In order to ensure that no delegate contains o in their *Targets*, we add a new statement `remove (o for D)` which, for each multicast delegate m, does the following: if the invocation list of m contains at least one delegate with target object different from o, then all delegates with target object o are removed from the invocation list. Otherwise m is disabled. The new pre-condition for

disable and the new *remove* statement ensure that program invariant **P5** holds throughout the execution of the program.

Extension 2: Keeping Track of the Invocation List

This last approach requires the largest amount of specification, but also makes it possible to specify equality. The idea is to model the complete invocation list of a delegate as shown the following specification:

```
public Delegate![]! GetInvocationList();

public static operator == (Delegate a, Delegate b)
   ensures ((object)a) != null && ((object)b) != null &&
      a.Target == b.Target && a._methodPtr == b._methodPtr <==> result;

public Del! Construct(object @object, IntPtr method)
   ensures object != null ==> result.GetInvocationList().Length == 1 &&
      result.GetInvocationList()[0] == result &&
      ((object)result.GetInvocationList()[0]) == ((object)result);
   ...

public static Delegate Combine(Delegate a, Delegate b)
   ensures ((object)a) != null && ((object)b) != null ==>
      result.GetInvocationList().Length == a.GetInvocationList().Length +
      b.GetInvocationList().Length &&
      forall { int i in (0:a.GetInvocationList().Length);
         result.GetInvocationList()[i] == a.GetInvocationList()[i] } &&
      forall { int j in (a.GetInvocationList().Length:b.GetInvocationList().Length);
         result.GetInvocationList()[j] ==
            b.GetInvocationList()[j-a.GetInvocationList().Length] } &&
      forall { int i in (0:a.GetInvocationList().Length);
         ((object)result.GetInvocationList()[i]) ==
            ((object)a.GetInvocationList()[i]) } &&
      forall { int j in (a.GetInvocationList().Length:b.GetInvocationList().Length);
         ((object)result.GetInvocationList()[j]) ==
            ((object)b.GetInvocationList()[j-a.GetInvocationList().Length]) };
      ...

public static operator == (MulticastDelegate a, MulticastDelegate b)
   ensures ((object)a) != null && ((object)b) != null &&
      a.GetInvocationList().Length == b.GetInvocationList().Length &&
      forall { int i in (0:a.GetInvocationList().Length);
         ((Delegate)a.GetInvocationList()[i]) ==
          ((Delegate)b.GetInvocationList()[i]) } <==> result;
```

Equality for singlecast and multicast delegates is defined according to the semantics provided in [6]. *Construct()* ensures that the only delegate in the invocation list of the resulting delegate is the new delegate itself. Similarly to the previous approach, *Combine()* concatenates the invocation lists of the two arguments.

Thanks to the additional information given by the invocation list, with this approach we can also have a more fine grained solution for disabling delegates. We defined the enabled state of a delegate as the conjunction of all enabled states of the delegates in its invocation list. The `disable (o for D)` statement remains unchanged from the definition in Chapter 3. Thanks to the new definition of the enabled state for multicast delegates, program invariant **P5** is ensured to hold throughout the execution of the program.

Implementation Issues

The are a certain number of issues preventing us from specifying multicast delegate using the second and third approach.

No Reference Comparison Newly allocated objects cannot be compared by reference in specification or pure methods. This limitation prevents us from specifying *Construct()* and *Combine()* in the third approach. For *Construct()*, the solution is to inject the post-condition as an assume statement after the construction of the delegate. For *Combine()* the problem is different. In the quantifications we are not comparing newly allocated objects: the array itself returned by *result.GetInvocationList()* is a new object indeed, but the elements of the array we are actually comparing are not. The admissibility checker of the Spec# compiler should be extended in order to allow such comparisons.

Array Concatenation Regarding the specification of targets and invocation lists as arrays, the theorem prover seems unable to verify the properties needed for disabling delegates. The details regarding this issue are explained in a bug report (see Appendix E.6, page 105).

4.2.3 Combining Delegates in BoogiePL

Previously in this chapter we analyzed the four requirements necessary for a safe invocation of multicast delegates. In order to satisfy stability Conditions 4.1 and 4.2, we add two new assertions before any call to *Delegate.Combine*[1]. The assertions verify that the post-condition implies the pre-condition and that the post-condition remains valid after any number of invocations of target methods. We consider a call of the form:

`r = Delegate.Combine(a, b)`, where a and b are of type D.

$\mathcal{T}[r = Delegate.Combine(a,\ b)] =$

```
1 r := call Delegate.Combine(a, b);
2 assert Pre_D(p_0^0, p_1^0, ..., p_n^0, h^0) && Post_D(p_0^0, p_1^0, ..., p_n^0, p_0^1, p_1^1, ..., p_n^1, h^0, h^1)
3     ==> Pre_D(p_0^1, p_1^1, ..., p_n^1, h^1)
4 assert Pre_D(p_0^0, p_1^0, ..., p_n^0, h^0) && Post_D(p_0^0, p_1^0, ..., p_n^0, p_0^1, p_1^1, ..., p_n^1, r^1, h^0, h^1) &&
5     Post_D(p_0^1, p_1^1, ..., p_n^1, p_0^2, p_1^2, ..., p_n^2, r^2, h^1, h^2)
6     ==> Post_D(p_0^0, p_1^0, ..., p_n^0, p_0^2, p_1^2, ..., p_n^2, r^2, h^0, h^2)
```

In the first assertion p_i^0 represent a parameter in the pre-state, p_i^1 a parameter in the post-state, h^0 the pre-state heap and h^1 the post-state heap. This verifies that the pre-condition of the delegate still holds after a call. By induction it can be shown that when this condition is true, the pre-condition holds after any number of calls.

The second assertion verifies that the post-condition remains valid after any number of invocations. In this case p_i^0 represent a parameter before the invocation, p_i^1 after a first invocation, p_i^2 after a second invocation and the same for r^1, r^2, h^0, h^1, h^2. On the left hand side, the formula starts by verifying that the pre-condition holds before the first invocation and that the post-condition holds afterwards. At this point, by the previous assertion, we know that the pre-condition for the second invocation also holds and we require the

[1] As a simplification in fact we consider *Delegate.Combine(Delegate, Delegate)* to be the only method able to create new multicast delegates. In reality there are also other methods which can achieve this; our methodology can be extended to such methods as well.

post-condition after the second invocation to hold as well. We then require the left hand side to imply that the post-condition is valid across two invocations; this is accomplished by verifying the post-condition using parameters and heap before the first invocation as pre-state, and parameters and heap after the second invocation as post-state. In this it is possible to show that if the assertion holds, then the post-condition remains valid, with respect to the initial pre-condition, after any number of calls.

To better understand these requirements, let us consider, once again the two examples shown previously:

```
delegate void AddHandler(ref int i)
    ensures i == old(i) + 1;
```

We have:

$(true \wedge i^1 = i^0 + 1 \wedge i^2 = i^1 + 1 \Rightarrow i^2 = i^0 + 1) \Rightarrow \bot$

Which, in fact, does not hold. The second example was:

```
delegate void DivisionHandler(ref int i);
    requires i > 0;
    ensures i <= old(i);
```

We have:

$(i^0 > 0 \wedge i^1 \leqslant i^0 \Rightarrow i^1 > 0) \Rightarrow \bot$

In this case as well the implication does not hold. We now consider an example with return value:

```
delegate int IncrementOneHandler(int i)
    ensures result == old(i) + 1;
```

We have:

$(true \wedge r^1 = i^0 + 1 \wedge r^2 = i^1 + 1 \Rightarrow r^2 = i^0 + 1) \Rightarrow \bot$

Finally we consider a correct example:

```
delegate void IncrementHandler(ref int i)
    requires i > 0;
    ensures i > old(i);
```

We have:

$i^0 > 0 \wedge i^1 > i^0 \Rightarrow i^1 > 0$
$i^0 > 0 \wedge i^1 > i^0 \wedge i2 > i^1 \Rightarrow i^2 > i^0$

In this case both implication can be proven, we deduce that with respect to the first two requirements, the example is indeed correct.

Condition 4.3 states that the enabled state of a delegate possibly included in the invocation list, cannot be changed. In order to ensure this condition we prevent methods used as targets for delegates from defining frame conditions of the form `modifies o.***`. As seen in Chapter 3, this particular frame condition is the only possibility for expressing the fact that a method disables delegates. By preventing its use in targets of delegates, we can ensure that a delegate can never disable any other delegate.

Listing 4.4: Disabling delegates in a target of a delegate

```
public class C {
    [Dependent(typeof(Del))] public bool Var;

    public void Foo()
        modifies this.Var;
    { disable (this for Del); }
}
```

Listing 4.5: Subtyping and generics

```
public static void Foo(List<A> l) {
    l.Add(new A());
}

public static void Bar() {
    List<B> l = new List<B>();
    Foo(l);
    foreach (B b in l) { ... }
}
```

Consider the example shown in Listing 4.4, where method *Foo* is target for some delegate of type *Del*. As it is defined in the example, *Foo* would fail the frame condition checks. In fact the *disable* statement in its body, modifies the *_enabled* field of all delegates which are peers with *this*. On the other side, if we were to add a `modifies this.***` to *Foo*, then any instantiation of a delegate having *Foo* as target would fail.

Finally Condition 4.4 is ensured by the fact that in Spec# ownership transfer for already owned objects is not allowed.

4.2.4 Subtyping and Multicast Delegates

To conclude the section about multicast delegates, it is useful to point out the idea behind the new subtyping relations and type rules in general. In principle the type rules for delegates subtypes are the same as for normal subtypes: nominal type rules. Using structural type rules for delegates in Spec# is an alternative that has been considered but discarded as it does not follow the C# standard.

The most important situation to consider is related to multicast delegates. Multicast delegates define an invocation list of delegates of a specific type. With generics, upcasting some type $T{<}U{>}$ to $T{<}V{>}$ can lead to problems. Consider the C# example in Listing 4.5, where B extends A.

This situation is clearly wrong because in the list of generic type B, an element of type A could be inserted. For this reason upcasting some type $T{<}U{>}$ to $T{<}V{>}$ is never allowed, not even when U extends V. A similar scenario would in principle also be possible with delegates, it suffices to take the previous example and consider the invocation list; an example is shown in Listing 4.6.

Delegates, however are immutable from the client's perspective. When method *Foo()* com-

Listing 4.6: Delegate subtyping with multicast delegates

```
public static void Foo(BaseDel! d) {
    d += new Del(...);
}

public static void Bar() {
    SubDel d = new SubFel(...);
    Foo(d);
    d();
}
```

bines delegates, it is in fact creating a new multicast delegate and it is not modifying its argument. If we do not consider reflection, which is not part of the verification anyway, in Spec# is it impossible for a client to extend the invocation list of a delegate. This allows us to define delegate upcasting as a valid operation.

4.3 Events

In the background chapter we saw that an event is nothing else than syntactic sugar translated in one private delegate and two public methods for adding and removing delegates. In order to extend the methodology to events, is suffices to apply the same principles to the delegate generated by the event. In reality there is a small additional element that we need to consider. In order to be able to verify events, we must have some information about the owner of the event, that is, the owner of the underlying delegate. Finally the pre-condition for adding delegates to events must imply the pre-condition of *Delegate.Combine*. The specification of the compiler-generated methods for a rep event *MyEvent* of type *MyEventHandler* in class *MyClass* is as follows:

```
public void add_MyEvent(MyEventHandler! value)
    requires Owner.Is(value, this, typeof(MyClass);
    requires value.IsEnabled;
```

```
public void remove_MyEvent(MyEventHandler! value)
    requires Owner.Is(value, this, typeof(MyClass);
```

For peer events *Owner.Is* is replaced with *Owner.Same*.

Finally, in order to meet the following pre-condition of *Delegate.Combine()*:

```
requires ((object)a) != null && ((object)b) != null ==>
        ((object)a.GetType()) == ((object)b.GetType());
```

The following *assume* statement is injected before combining the event with the argument in the *add* method:

```
1 assume Heap[this, MyClass.MyEvent] != null && value != null
2     ==> TypeObject(typeof(Heap[this, MyClass.MyEvent])) ==
3         TypeObject(typeof(value));
```

Listing 4.7: GUI programming with Events

```
public delegate void Click

public class Gui : Form
{
    private Button button1;
    private Button button2;

    public Gui() : base() {
        button1 = new Button();
        button1.MouseDown += new MouseEventHandler(this.Button1_click);
        button2 = new Button();
        button2.MouseDown += new MouseEventHandler(this.Button2_click);
        /* init the form by placing the buttons */
    }

    private void Button1_Click(object sender, MouseEventArgs e) { /* ... */ }

    private void Button2_Click(object sender, MouseEventArgs e) { /* ... */ }
}
```

4.3.1 Ownership and Events

Even though the extension of the methodology to events does not require additional concepts, events expose the current limitations of the ownership approach of Spec#.

In Listings 4.7 we have a simple form called Gui with two buttons: *button1* and *button2*. When one of these buttons is clicked, the corresponding handler method is called through the *MouseDown* event of class *Button*. Let us analyze how this example can be verified. We know that a delegate and its target must be peers; in the example, considering *button1* only, this means that an instance *g* of *Gui* and *g.button1.MouseDown* must be peers. There are three possibilities for the corresponding ownership tree.

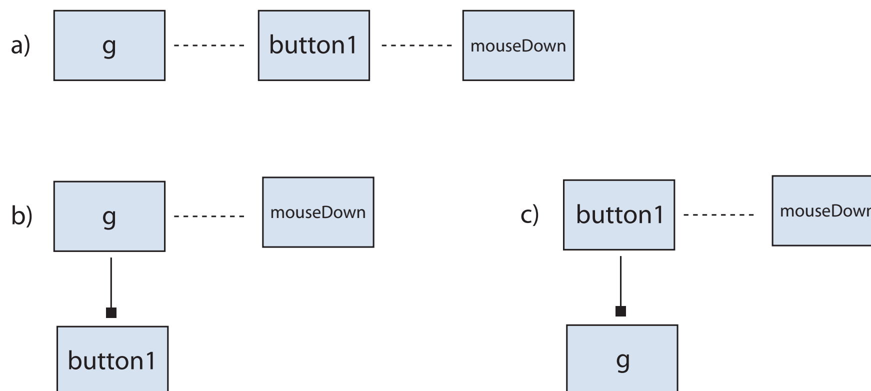


Figure 4.1: Owneship trees for a GUI with events

The first approach, Figure 4.1 a, is straightforward but has a big disadvantage: *Gui* cannot express invariants on the state of the button. A possible invariant would be for the button to

be enabled only in specific circumstances. In practice this first topology forces all elements of a user interface to be peers [2] which makes expressing invariants in general impossible. The second approach, Figure 4.1 b, allows *Gui* to express invariants on *button1*. Unfortunately this design is not applicable. Remember, in fact, that in principle we expect the class declaring an event to be responsible for its invocation as with the example of the observer pattern. If the class is owned by the event, then it becomes impossible for any of its non pure methods [3] to invoke it. With respect to the example above let us suppose that we have a method m of an instance *button1* of class *Button* who is responsible for invoking *MouseDown*. In m we know that the owner of *button1*, *MouseDown*, is exposed. Now the pre-condition for the invocation of *MouseDown* is for the owner of *MouseDown* to be exposed and for *MouseDown* itself to be valid. Since this condition cannot be met, the invocation becomes impossible. In the third approach, Figure 4.1 c), the button owns the *Gui*. This solves the problem of consistency at invocation but it is easy to see that this scenario becomes impossible with two buttons.

Excluding scenarios two and three we are left with only the first scenario: the flattening of the ownership tree. In this scenario there is an additional problem that we need to consider. As seen in the previous paragraph, the owner of *button1* must be exposed before *button1* can invoke the event. The question now is: who is responsible for this operation? This is a general issue that has yet to be solved for ownership and external input. The same problem in fact reappears whenever we have actions which originate outside our program; user's input, networking, and timers. As this topic outside the context of this thesis, we leave the problem of verifying events in combination with external input open. The conclusion about verification of events is that it is an adaptation of the verification of multicast delegates and at the current state of Spec# it is not possible to verify events connected to external input.

4.4 Closed Target Type Delegates

Closed target type (CTT) delegates are a new form of delegate we introduced in this work. The idea of CTT delegates is to close the type T of the target object during the declaration of a delegate. This implies that only methods of instances of T can be used as targets for that delegate. Before going into details on how CTT delegates are implemented and used, we argue why CTT delegates are useful.

4.4.1 Pre-, Post-Conditions and Invariants

The methodology introduced so far, composed by delegate pre- and post-conditions, delegate invariants and delegate subtyping, allows verifying a broad variety of delegate uses. In general the client of a delegate is not concerned about the target of the delegate; we refer to these scenarios as *loosely coupled*. There are situations however where some information must be known about the target, an example is for a method to have a pre-condition depending on the state of its target object. Delegate invariants allow coping with such situations. There are however *tightly coupled* cases where a pre-condition depends on both the state of the target and the arguments, or a post-condition depends on the return value, the argument, and the target's state. In such situations delegate invariants are insufficient. We might argue that when the relationship between a delegate and its target is so strong, then the delegate approach is wrong and interfaces should be used instead. Consider however a scenario where a class offers multiple implementations for the same function and the

[2] All elements with a registered event.

[3] The situation might work with pure methods only, but this is not realistic.

Listing 4.8: Vector operations with CTT delegates

```
public delegate Vector! VectorOperation (Vector! other);

class Vector {
    [Rep] private float[]! elements;
    [Peer] public VectorOperation Operation;
    public int Length;

    invariant this.Length == elements.Count;

    public Vector! Sum(Vector! that)
        requires this.Length == that.Length;
        ensures result.Length == this.Length;
    {
        /* return the componentwise sum */
    }

    public Vector! CrossProduct(Vector! that)
        requires this.Length == that.Length;
        ensures result.Length == this.Length;
    {
        /* return the crossproduct */
    }
}
```

implementation to be used depends on the state of the target; a delegate can be used to point to the implementation that is used at a specific moment. This is a tightly coupled scenario which our methodology should allow to handle.

We propose and implement an extension to normal delegates allowing to close the type of the target. Closing the target's type permits to express more information and allows to verify scenarios that could not be handled before. Let us present this extension with an example.

The example of Listings 4.8 shows a partial implementation of a vector (as we could find in a mathematical library). The idea behind this particular implementation is that a client of a *Vector* v, performs some operation using v and some other vector in order to obtain a new one, but it is not responsible for deciding which exact operation must be performed. An example for such situation is a class responsible for drawing on screen the result of the operation; while drawing we are not interested in whether the operation is a sum or cross product. The requirement for all such operation is for both vectors to be of the same length. Moreover all operations ensure that the length of the resulting vector is also the same. How can this simple scenario be verified? We need to express a pre-condition and a post-condition both depending on the state of the target object and the state of the argument. A requires clause is not sufficient because in principle a delegate does not know about the identity of the target object while an invariant is also not enough, in fact it cannot use the arguments.

Finally, delegate invariants allow to express pre-conditions on the state of the target object. However invariants are a much stronger construct than pre-conditions. A pre-condition must be showed to hold only before an invocation of the method which is declaring it; an invariant on the other side, must hold whenever the object is valid. By expressing pre-conditions of a method as delegate invariants, we actually force those pre-conditions to hold whenever

there are enabled delegates with the corresponding object as target: a requirement which is stronger than the pre-condition itself. Even though this approach ensures modularity, we believe that in certain situations a less demanding solution would be useful.

4.4.2 Tightly Coupled Delegates

The idea is for the delegate declaration to specify the type of the target object. Once the type of the target is known, it becomes possible to express the information that was previously contained in invariants directly in the pre-condition of the delegate. Moreover with closed target types we can also express pre- and post-conditions depending on both the state of the target object and the arguments. The previous example can be rewritten as:

```
public delegate<Vector> Vector! VectorOperation (Vector! other);
   requires this.Target.Length == other.Length;
   ensures result.Length == this.Target.Length;
```

Note the additional *<Vector>* after the *delegate* keyword. We can close the target type T at declaration by specifying the class or interface for which the delegate is closed by adding *<T>* after the delegate keyword.

In the previous sections we explained that with normal delegates, a pre-condition could not depend on the state of the target because of the lack of information about its type. CTT delegates solve this problem by providing full knowledge about the target.

In CTT delegates we do not allow delegate invariants. Moreover CTT delegates do not accept static methods as those methods do not have a target object. Target type dependent frame conditions are also forbidden because they can be replaced by normal modifies clauses.

4.4.3 CTT Delegates and Subtyping

Even though it would be possible to allow subtyping for CTT delegates, we decided not to allow this feature. Subtyping could be allowed for modularity reasons, but the idea of CTT delegate is to allow handling tightly coupled situations and not having a generalized and modular system. For this reason it is our belief that subtyping in CTT delegates would introduce additional complexity for the programmer which would be, in practice, seldom practical. It is still possible to extend the methodology to allow subtyping in CTT delegates[4]. In this case the type of the target can be further restricted, e.g. a CTT delegate subtype with a base delegate for type T is allowed to close the target for a type U which is a subtype of T. Moreover it would also be possible for a CTT delegate to extend a normal delegate.

4.4.4 CTT Delegates in BoogiePL

At the BoogiePL level, CTT delegates are handled like normal delegates but with the only difference that the *SafeInvoke()* method does not require the delegate to be enabled (as there are no invariants declared in the delegate). The *_enabled* field is still part of the delegate

[4]Like in the case of normal delegates, subtyping and upcasting CTT delegates would be sound: the invocation list of a particular instance of a multicast delegate cannot be changed. For this reason the upcasting issues that are present with generics do not apply in case of CTT delegates.

since it is defined in *System.Delegate*, but remains ignored. Concerning the specification of a delegate, since invariants are missing but the target type is known, the contract is handled as in normal pre-, post- and frame conditions.

CHAPTER 5

Implementation and Runtime Checks

In this chapter we explain how the methodology described in Chapters 3 and 4, is implemented in the Spec# programming system. There are three levels where this implementation happens. First, the compiler has be changed in order to accept new syntax and typing rules. Second, the specification information is translated in CIL while preserving its full meaning and keeping CIL valid. Third, Boogie has been adapted so that the specification of CIL can be translated to BoogiePL.

5.1 From SSC to CIL

5.1.1 Delegate Instantiation

During compilation, every call to a constructor of a delegate is replaced with a call to the static *Construct()* method generated by the compiler. As we will see later in this chapter, this is necessary for performing runtime checks. An instantiation of the form `Del d = new Del(a.Foo)` is translated in `Del d = Del.Construct(a, (IntPtr)a.Foo)`.

5.1.2 Contracts for delegates

Contracts for delegates are encoded, during compilation, in *DelegateContract*, part of the *DelegateNode* class in the compiler. A *DelegateContract* specifies a list of pre-conditions, post-conditions, frame conditions and invariants. It is a merging of a method contract (pre-, post-, frame conditions) and a type contract (invariants). The method contract part of the delegate contract is copied to the contract of the *SafeInvoke()* method and discarded during the compilation process. This allows us to handle specification information for delegates in the same way as for methods and classes. Specifically pre-, post- and frame conditions are serialized in the *Requires*, *Ensures*, and respectively *Modifies* attributes

of method *SafeInvoke()*, while invariants are serialized in the *Invariant* attribute of the corresponding delegate class.

During compilation delegate invariants of the form `invariant when { T, E }`, where T is a type and E is a boolean expression, are stored in the *DelegateInvariant* class and the expression E is modified to obtain *E'* such that *E' = E[target/((T)this.get_Target)]*. In other words all instances of the special identifier *target* are replaced with an expression which typecasts a call to the *get_Target* method of *System.Delegate* to type T. For each delegate invariant, *E'* is obtained and copied to a new class invariant for the corresponding delegate class. These new invariants are then serialized like normal invariants.

The definition of an admissible delegate invariant (see Definition 3.1, page 31) is verified by the compiler. The correctness of (i) is ensured by setting the type of *target* to T and using a restricted scope. In fact the special identifier *target* in an invariant is set during compilation to T. Conditions (ii) and (iii) are also verified during compilation by checker and admissibility checker.

5.1.3 Delegate Subtyping

Delegate subtyping is a new concept which was not previously known in neither C# nor Spec#. In [15] it was stated that delegates subtypes are only a specification construct and that they do not affect program execution; even though this is true for static verification, in order be able to implement runtime checks we need to provide a notion of delegate subtypes for execution. The first idea with delegate subtyping would be to directly encode the subtype relation in the class generated for a delegate by the compiler. Unfortunately these classes are sealed: unsealing them should allow to extend them. Even though this is possible from a compilation perspective, the .NET common language runtime (CLR) requires all delegate classes to be sealed. This requires us to find a different approach for handling delegate subtypes.

Instead of encoding delegate subtyping as normal class subtyping, we extend *DelegateNode* with an additional *BaseDelegate* field. This field keeps a reference to the type of the delegate we are extending. We then continue to treat the delegate as before, with the addition that we perform checks to verify that the signature of a delegate is identical to the signature of its base delegate. The problem arises when we want to upcast a delegate d of type T to U, where T is a delegate subtype of U. We achieve this by wrapping the delegate d to be upcasted in a new instance of a delegate of type U, which has d as target object and *d.SafeInvoke()* as target method.

Delegate upcasting follows the format described in the example below:

```
delegate void BaseDel();
delegate void SubDel() : BaseDel;

public static void Main() {
    SubDel s = new SubDel(...);
    BaseDel d = s;
}
```

The example is translated during compilation as follows:

```
public static void Main() {
    SubDel s = new SubDel(...);
    BaseDel d = new BaseDel(s.Invoke);
```

```
}
```

or, considering the replacement of constructors with the *Construct()* method:

```
public static void Main() {
    SubDel s = SubDel.Construct(...)
    BaseDel d = BaseDel.Construct(s, (IntPtr)s.Invoke);
}
```

There is still the open question of the opposite operation: downcasting. Given the fact that we expect downcasting operations to be less frequent than upcasting, we do not allow for a direct cast. The programmer however is still free to extract the subtype delegate:

```
public static void Main() {
    SubDel s = new SubDel(...);
    BaseDel d = new BaseDel(s.Invoke);
    s = d.Target as SubDel;
}
```

In order to implement delegate subtyping as normal class subtyping we would have to either modify the .NET CLR or largely modify the typesystem and typechecking of the compiler. Neither option is feasible in the context of this thesis; however we believe the pseudo-subtyping as described above to be sufficient for effectively using delegates in verified programs.

Delegate subtype information is finally encoded in a *BaseDelegate* attribute of the corresponding delegate class. The following example shows how this is achieved:

```
[BaseDelegate(typeof(BaseDel))]
public sealed class SubDel {
    ...
}
```

Inheritance of Specification

For subtyping, the compiler performs the following checks:

1. A delegate subtype is not allowed to define additional *requires* clauses because of the refinement principle. This means that all requires clauses are completely inherited from its base delegate;
2. A delegate subtype can define additional *ensures* clauses;
3. A delegate subtype can define additional *modifies when* clauses, but its is not allowed to define additional *modifies* clauses;
4. A delegate subtype can define additional delegate invariants;

With respect to the first rule, following the refinement principle it should be possible for a delegate to weaken the pre-condition of its base delegate. However in our implementation we decided to follow the current Spec# overriding rules for methods by directly inheriting the precondition. The same reasoning also applies for forbidding the strengthening of the normal frame condition.

5.1.4 Disabling Delegates

In order to implement the disabling of delegates as described in Chapter 3, we need the ability to:

1. Maintain information about the enabled state of a delegate;
2. Disable delegates of a certain type and with a specific target object;
3. Query delegates with regard to their enabled state;
4. Assert that no delegate of a certain type and with a specific target object is enabled.

In order to accomplish this, we add to the *Microsoft.Contracts* namespace a static class named *DelegateReferencesHolder*. This class is responsible for providing the four requirements described above and also for performing some of the runtime checks explained later in this chapter.

Class *DelegateReferencesHolder* provides the following private member:

```
private static Dictionary<object, List<WeakReference>> references
```

and the following public methods:

```
public static void Disable(object! o, Type! t);
public static bool IsEnabled(Delegate d);
public static bool NoActiveDelegate(object! o, Type! t);
```

The dictionary *references* is responsible for internally maintaining the enabled state of a delegate; *Disable()* allows to disable a delegate; *IsEnabled()* uses the information contained in *references* to return the enabled state of a delegate, *NoActiveDelegate()* returns whether there are enabled delegates of type t with o as target. The information regarding disabling delegates is completely encoded in calls to the three methods described above. To make it easier for the programmer to use this information we also provide the following syntactic sugar.

Disabling a delegate `disable (o for T)`, where o is a non null object and T is a type expression is translated into `DelegateReferencesHolder.Disable(o, T)`.

Getting the enabled state of a delegate `d.IsEnabled` where d is a delegate is translated into `DelegateReferenceHolder.IsEnabled(d)`.

5.1.5 Frame Conditions

For target type dependent frame conditions, *modifies when* clauses are translated back to normal modifies clauses. During compilation when parsing a *modifies when* clause, we create a new *ModifiesWhen* object that we add to the delegate contract. As with the case of delegate invariants, later in the compilation the expression E of a *ModifiesWhen* clause is transformed in *E'* such that *E' = E[target/((T)this.get_Target)]*; *E'* is then added as normal modifies clause to the *SafeInvoke()* method. When the clause is serialized in the

Modifies attribute, we add the type of the target as additional parameters to the attribute. This allows us to easily distinguish *modifies when* clauses from normal *modifies clauses.*

All *modifies* clauses are serialized in the *Modifies* attribute. Spec# already handles this as follow: if the clause is of the form *o.f*, where *o* is a reference and *f* is a field, then the corresponding expression *E'* will be of the form *ref o.f.* If the clause is of the form *o.***, *o.** or *o.0*, then *E'* is a method call to respectively *Guard.ModifiesPeers(o)*, *Guard.ModifiesObject(o)* or *Guard.ModifiesNone(o)*. We extend this approach by allowing clauses of the form *o.**** which are encoded as *Guard.ModifiesEnabled(o)*.

5.1.6 CTT Delegates

Concerning closed target type delegates, the target type is kept is a field of *DelegateNode*. The idea with CTT delegates is that we only need to worry about pre-, post- and frame conditions by continuing to use the *SafeInvoke()* and *Construct()* methods. With respect to normal delegates however, we do not have invariants and modifies *when clauses*; besides this, handling CTT delegates is similar to normal delegates.

Since the type of the target is know, the *Target* property, or more specifically the *get_Target()* method, is hidden by a new definition where the return type is non-null and equal to the target type. Moreover the *Construct()* method takes a non-null reference to a target object of a type equal to the target type instead of a possibly null reference to a target of type object.

5.1.7 Compiler Architecture

As introduced above, new classes are defined in order to maintain the additional information during compilation. Figure 5.1 shows an extract of the class diagram for the nodes of the compiler.

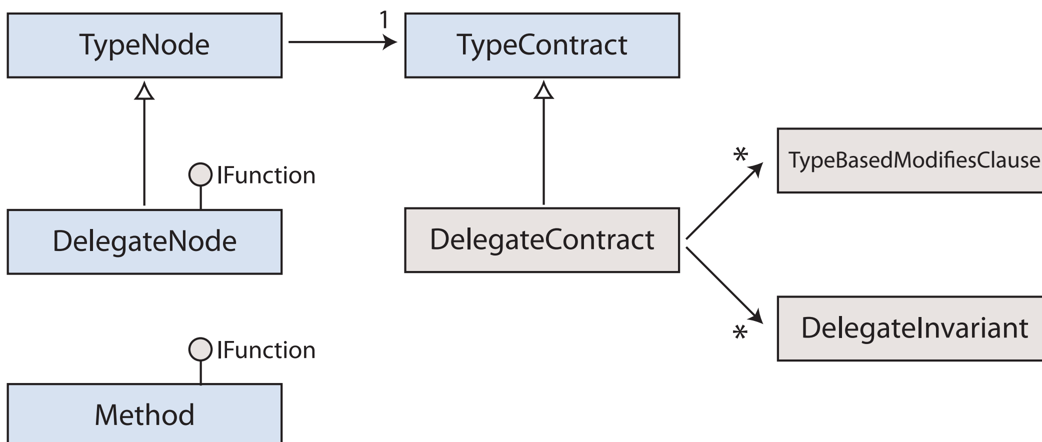


Figure 5.1: Modifications of the class diagram of the ssc compiler.

DelegateContract is used to handle the contract before it is copied to the contract of *SafeInvoke()*, *TypeBasedModifiesClause* and *DelegateInvariant* represent respectively a *modifies when* clause and an *invariant when* clause. Moreover, in order to reduce code duplication, *DelegateNode* and *Method* now implement the common interface *IFunction*. This allows

handling the compilation of parts which are common between methods and delegates with less overhead.

The compiler pipeline is structured as shown in Figure 5.2. In general terms, our extension of the compiler includes the following parts: *Parser* is modified in order to accept the new syntax for delegates and delegate specification; in *Looker* the additional members of the compiler-generated class for a delegate are defined and added. *Checker* is then responsible for verifying delegate subtyping and for serializing parts of the contract, while the *Normalizer* executes the remaining serialization tasks. Finally, we extend *Analyzer* in order to perform non-null analysis in delegate contracts.

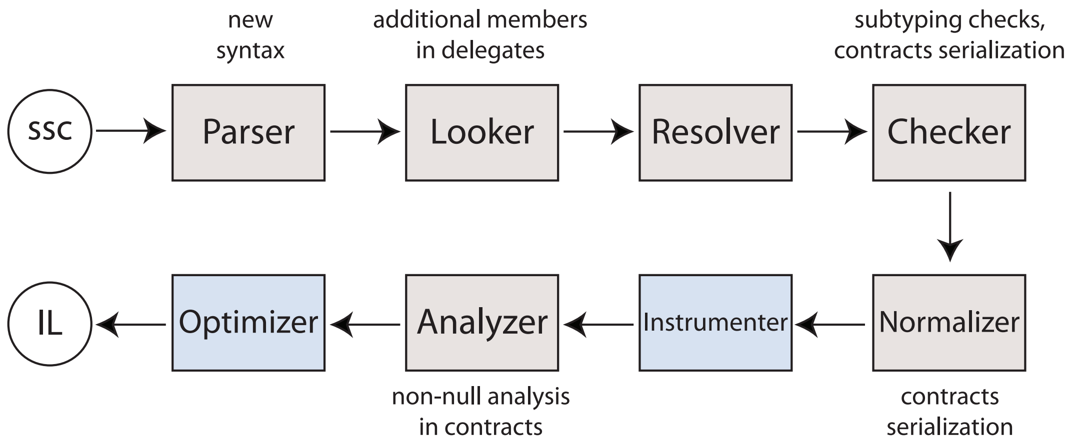


Figure 5.2: Compilation pipeline of ssc.

5.2 Boogie

In this brief section we point out the important topics regarding the deserialization of the contracts encoded in CIL and attributes in Boogie. An extract of the Boogie pipeline is shown is Figure 5.3.

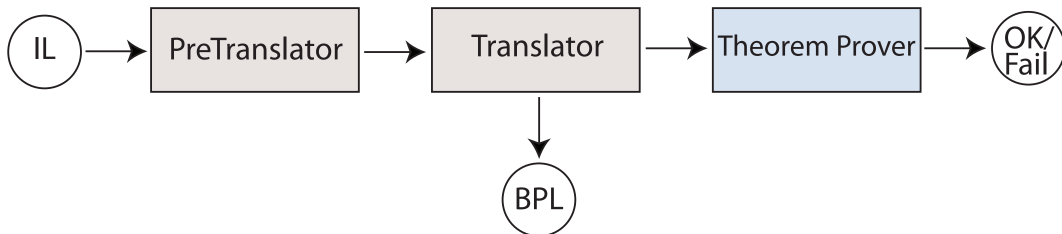


Figure 5.3: Boogie pipeline.

The CIL is first deserialized back to an AST, then the *PreTranslator* prepares the AST for the actual translation phase where the AST is translated in BoogiePL. Finally the theorem prover verifies the program and, if it finds problems, outputs the list of failing conditions. During pre-translation we go through the AST and remove all wrappings of delegates inserted for subtyping and we replace all calls to the *Construct()* method with the original constructor. This means that after pre-translation, statements such as `Del d = new Subdel(a.Foo)` are possible.

Listing 5.1: Public interface of DelegateReferencesHolder

```
public static class DelegateReferencesHolder {
    public static bool IsEnabled(Delegate);
    public static void Disable(object, Type);
    public static void NoActiveDelegate(object, Type);
    public static void Add(Delegate);
    public static void Remove(Delegate);
    public static void CheckEnabled(Delegate);
    public static void CleanDeadReferences(Delegate);
    public static void StartWriting([Delayed] object);
    public static void EndWriting([Delayed] object);
}
```

This modified version would fail the typechecking phase of the compilation but fortunately with Boogie we do not have to worry about this and we can safely treat delegate subtypes as actual class subtype. Moreover, as the example shows, while removing the wrapping we also re-introduce the normal constructor by removing the call to the *Construct()* method which is only used for runtime checks. Note that *Construct()* is also replaced when the constructor is not wrapped.

Since the actual behavior of the bodies of delegate methods is defined at runtime and is not relevant for the static verification, all bodies of delegates classes are not emitted. In fact we trust the actual implementation to be correct.

In order to be able to correctly emit target type dependent frame conditions, we analyze all expressions of the frame conditions of *SafeInvoke()* methods. We a type cast is found in one of the expressions, then we know that the condition corresponds to a *modifies when* clause. This allows us to emit the correct BoogiePL expressions.

5.3 Runtime Checks

The Spec# programming system implements both static and runtime checks. The same is also valid for the verification of delegates. There are some limitations however that one must consider for runtime checks; first performance must be considered; second some checks are impossible because of the lack of a theorem prover. Because of this limitations, runtime checks are usually a subset of static checks.

5.3.1 Building Blocks

In the previous chapters we already introduced the helper class *DelegateReferencesHolder* and some of the new methods generated by the compiler for delegate classes. In this section we re-examine these additional elements in the perspective of runtime checks. Listings 5.1 shows the public interface of *DelegateReferencesHolder*.

DelegateReferencesHolder is responsible for handling all information regarding the enabled state of delegates. In order to do this, it keeps a dictionary which associates objects with delegates having that object as target. The values of the dictionary are lists of weak references to delegates. It is necessary to use weak references because otherwise the garbage collector would never free the memory used by non reachable delegates. *Add(Delegate)*

Listing 5.2: Partial interface of a compiler generated class for delegate LogFunction

```
delegate int LogFunction(string s);
    requires s != null;
    ensures result < s.Length
    invariant when { Console; target.IsVisible };

public sealed class LogFunction : System.MulticastDelegate {
    ~LogFunction();
    static LogFunction();
    public LogFunction! Construct(Object, IntPtr);
    public void SafeInvoke(String);
    public bool SpecSharp::CheckInvariant(Boolean);
    private static Guard SpecSharp::GetFrameGuard(object o);
    public Guard! SpecSharp::FrameGuard { get; }
    private void SpecSharp::InitGuardSet();
}
```

and *Remove(Delegate)* are used to add and remove delegated from the dictionary. *Check-Enabled(Delegate)* is used to throw an exception in case the delegate passed as argument is not enabled; `d.IsEnabled` is translated in a call to *IsEnabled(d)*; `disable (o for D)` is translated in a call to *Disable(o, typeof(D))*; *NoActiveDelegate* checks that no delegate with specified target and type is enabled; *CleanDeadReferences(Delegate)* is used when a delegate is garbage collected. The class also provides two helpers for implementing checks when objects are exposed: the last two methods are used when exposing objects to pack and unpack delegates.

Listings 5.2 shows the class diagram of a delegate class of the console example of Chapter 3. *SafeInvoke()* and *Construct()* were already introduced in previous chapters. For runtime checks *SafeInvoke()* replaces *Invoke()* and checks that the delegate contract is respected, *Construct()* is used instead of the normal constructor. Finally the destructor is used in combination with *DelegateReferencesHolder*. The last four methods are also present in other classes. Their purpose is to maintain runtime information equivalent to the validity of objects in Boogie. A *Guard* is used to prevent race conditions, and enforce ownership and invariants. *SpecSharp::CheckInvariant()* contains runtime checks for checking delegate invariants; *GetFrameGuard()* and *FrameGuard* return the *Guard* of the delegate and finally *InitGuardSet()* is used for initialization.

5.3.2 Implementing Runtime Checks

Instantiation

All calls to a delegate constructor are replaced with calls to the static *Construct()* method. The body of *Construct()* has the format shown in Listings 5.3.

The method first instantiates a new delegate of the corresponding type using the normal delegate constructor. The second instruction is used to force the *Guard* of the delegate to be instantiated. This also forces the delegate invariant to be checked. The delegate is then added to *DelegateReferencesHolder* and finally the post-condition stating that the new delegate is enabled is checked. The post-condition check is wrapped in the standard

Listing 5.3: Construct() method.

```
[NoDefaultContract]
public static D modopt(NonNullType) Construct(object @object,
    IntPtr method) {
    D d = new D(object, method);
    Guard g = d.SpecSharp::FrameGuard;
    DelegateReferencesHolder.Add(d);
    try {
        if (!DelegateReferencesHolder.IsEnabled(d)) {
            throw new EnsuresException("Postcondition violated from
                method D.Construct(System.Object,System.IntPtr)'");
        }
    }
    catch (ContractMarkerException) { throw; }
    return d;
}
```

try-catch (*ContractMarkerException*) which indicates to Boogie that the content of the try block has to be ignored for static checks.

Invocation

The Spec# compiler already de-sugares statements of the form $d(p_0, p_1, ..., p_n)$, where d is a delegate, in $d.Invoke(p_0, p_1, ..., p_n)$; for our static and runtime checks, we desugar it in $d.SafeInvoke(p_0, p_1, ..., p_n)$. The specification of *SafeInvoke()* contains all requires, modifies and ensures clauses of the original delegate. Moreover the body of *SafeInvoke()* also executes runtime checks, as shown in Figure 5.4

The body of *SafeInvoke()* is composed by three parts. The first part checks the enabled state of the delegate and the other user-defined pre-conditions. The second part invokes the delegate. The last part checks the user-defined post-condition and returns the result (if any). The implementation of the *SafeInvoke()* method for *LogFunction* is shown in Figure 5.5

Exposing an object

When exposing an object, runtime checks are performed first to check whether the object is exposable, and then to check the object's invariant at the end of the block. As seen in Chapter 3, delegates having the exposee as target must also be exposed. In the following example object c, which is target for l is exposed:

```
    Console c = new Console();
    c.IsVisible = true;
    LogFunction l = new LogFunction(c.WriteLine);
    expose (c)
    {
    }
```

The example is transformed during compilation as shown in Listings 5.6.

Listing 5.4: SafeInvoke() method.

```
[Requires("[System.Compiler.Runtime]Microsoft.Contracts.
    DelegateReferencesHolder::IsEnabled([mscorlib]System.Delegate){this}"),
    /* other pre- and post-conditions */]
public R SafeInvoke(P0 p0, ..., PN pn) {
    try {
        if (!DelegateReferencesHolder.IsEnabled(this))
        {
            throw new RequiresException("Precondition violated from
                method 'D.SafeInvoke(P0, ..., PN)'");
        }
    /* other pre-condition checks */
    }
    catch (ContractMarkerException) { throw; }
    int return value = this(p0, ... pn);
    try { /* post-condition checks */ }
    catch (ContractMarkerException) { throw; }
    return return value;
}
```

Listing 5.5: SafeInvoke() method for the example of LogFunction example.

```
[Requires("[System.Compiler.Runtime]Microsoft.Contracts.
    DelegateReferencesHolder::IsEnabled([mscorlib]System.Delegate){this}"),
    Requires("::!=(string,string){\$1,null}"),
    Ensures("::<(i32,i32){\${i32,\"return value\"},\$1@string::
        get_Length(){}}"]
public int SafeInvoke(string s) {
    try {
        if (!DelegateReferencesHolder.IsEnabled(this)) {
            throw new RequiresException("Precondition violated from method
                'LogFunction.SafeInvoke(System.String)'");
        }
        if (s == null) {
            throw new RequiresException("Precondition 's != null' violated
                from method 'LogFunction.SafeInvoke(System.String)'");
        }
    }
    catch (ContractMarkerException) { throw; }
    int return value = this(s);
    try {
        if (return value >= s.Length) {
            throw new EnsuresException("Postcondition 'result < s.Length'
                violated from method 'LogFunction.SafeInvoke(System.String)'");
        }
    }
    catch (ContractMarkerException) { throw; }
    int SS$Display Return Local = return value;
    return return value;
}
```

Listing 5.6: Exposing an object.

```
Console c = new Console();
LogFunction l = LogFunction.Construct(c, (IntPtr) this.WriteLine);
Guard.StartWritingAtNop(c, typeof(Console));
DelegateReferencesHolder.StartWriting(c);
try
{
}
finally
{
    Guard.EndWritingAtNop(c, typeof(Console));
    DelegateReferencesHolder.EndWriting(c);
}
```

The compiler insert a call to *Guard.StartWritingAtNop()* which throws an exception in case the object is not exposable and exposes the object. We add an additional call to *DelegateReferencesHolder.StartWriting()*, which has the effect of exposing all delegates with the exposee as target. This call also throws an exception in case a delegate is not exposable. At the end of the expose block, *Guard.EndWritingAtNot()* is called to pack the object; in the same way *DelegateReferencesHolder.EndWriting()* checks the invariants of the objects it previously exposed and them repacks them.

Disabling and The Enabled State

Disabling delegates is directly done with a call to `DelegateReferencesHolder.Disable(Object, Type)`. As explained above, *DelegateReferencesHolder* keeps a dictionary of the form `Dictionary<Object, List<WeakReference>>`. In the BoogiePL encoding of Spec# delegates, we added the additional *_enabled* field for storing the enabled state of a delegate. For runtime however it is not possible to directly add an *_enabled* field to all delegates as this would require us to modify *System.Delegate* defined in the .NET library. For this reason the enabled state of a delegate is maintained in the dictionary of *DelegateReferencesHolder*; a singlecast delegate is considered enabled at runtime if one of the following conditions is met:

1. It has no target object (e.g. its target method is static);
2. It is a CTT delegate;
3. It is present in the dictionary.

A multicast delegate is enabled if it is a CTT delegate or if all the delegates in its invocation list are enabled with respect to the previous definition. This also ensures that only singlecast delegates are kept in the dictionary. This is indeed the definition employed by *DelegateReferencesHolder.IsEnabled()*. The asymptotic complexity of retrieving the enabled state of a delegate is $\mathcal{O}(m \cdot t \cdot l)$ where m is the size of the invocation list, t is the average number of delegates per target object and l is the complexity of a dictionary lookup.

We already know that all new, non-CTT delegates with a target object are inserted in the dictionary during the construction of the delegate. Adding a delegate is done in $\mathcal{O}(l \cdot t)$, where l is the complexity of a dictionary lookup and t is the number of enabled delegate

having the same target as the new one. When delegates for a specific target object are disabled, *DelegateReferencesHolder* removes all such delegates from the lists in the dictionary. Disabling all delegates for an object has is done in $\mathcal{O}(l \cdot t \cdot r)$, where t is the number of delegates having the object as target and r is the complexity of removing one element from a list.

Garbage Collector

The last question that still has to be answered, concerning disabling delegates, is what happens when a delegate is garbage collected. The idea is to disable delegates before they are garbage collected. In order to achieve this a destructor is added to delegate classes. The destructor simply calls *DelegateReferencesHolder.Remove(Delegate)*. This method simply removes the delegate from the dictionary. The complexity in this case is the same as adding a new delegate, namely $\mathcal{O}(l \cdot t)$.

CHAPTER 6

Conclusions

In the first part of this last chapter we summarize the state of the implementation. We then conclude the work by providing some general impressions and discussing possible improvements and extensions.

6.1 State of the Implementation

In this section we summarize the state of the implementation of the verification of Spec# delegates. Figure 6.1 shows a summary of the implementation.

Singlecast Delegates We implemented the methodology proposed in [15] for singlecast delegates. When a delegate is instantiated, we perform refinement checks. When exposing an object having fields involved in delegate invariants, we also expose all delegates the object is dependent on. Delegate subtyping is implemented in BoogiePL as normal class subtyping. We implemented the `disable (o for D)` statement by adding an additional *_enabled* field to every delegate; to improve the way frame conditions can be expressed, we included an additional `modifies o.***` which allows to disable all delegates which are peers of *o*. For modifies clauses on the target object, we allow defining default frame conditions and target type dependent frame conditions of the form `modifies when { T; target.f }`. Moreover we allow a delegate subtype to refine the frame condition of its base type.

Additional Delegates We handle delegates with static methods as target as normal delegates. Even though static delegate invariants are allowed, their verification is not complete. This is due to the fact that static invariants are only partially implemented in Spec#. We extended the methodology to multicast delegates. When delegates are combined, we perform heap and parameter stability checks to verify that the specification of the multicast delegate respects four heap stability conditions. Thanks to this checks we are able to reduce a multicast delegate to a singlecast delegate. When disabling delegates for some

	Static analysis: Compiler	Static verification: Boogie	Runtime checks: Compiler
Instantiation of delegates			
Typechecking	✔	(✔)	(✔)
Validity of the target object	✘	✔	✔
Delegate invariant	✘	✔	✔
Refinement for pre-conditions	✘	✔	✘
Refinement for post-conditions	✘	✔	✘
Refinement for frame conditions	✔	✔	✘
Exposing objects			
Exposing of delegates	✘	✔	✔
Invoking delegates			
Typechecking	✔	(✔)	(✔)
Only enabled delegates can be invoked	✘	✔	✔
Exposed delegates cannot be invoked	✘	✔	✔
Misc			
Modification of fields of unexposed objects	✘	✔	✘
Admissible invariants	✔	(✔)	(✔)
Disabling singlecast delegates	✘	✔	✔
Non-null analysis in contracts	✔	(✔)	(✔)
Multicast delegates			
Stability checks	✘	✔	✘
Disabling multicast delegates	✘	✔	✔

✔ Implemented
(✔) Achieved by static analysis
✘ Non-implementable

Figure 6.1: State of the implementation.

target object, multicast delegates not having that object as target might be disabled as well. This is due to the fact that by reducing a multicast delegate to singlecast, we loose all information about its invocation list. We proposed two possible approaches to improve this situation; because of technical limitations and timing reasons, we could not implement them. We handle events like normal multicast delegates. To reduce the overhead for the programmer, we allow to declare rep and peer event fields and we automatically generate the corresponding pre-conditions necessary for combining them.

Closed Target Type Delegates In order to allow handling *tightly coupled* situations, where a delegate is instantiated with a method having pre- or post-conditions depending on both the state of the target object and the arguments, we introduced a new construct allowing to fix the type of the target object of a delegate during declaration. We handle CTT delegates like normal delegates, with the exception that invariant and target type dependent frame conditions are not necessary. For this reason a CTT delegate is always enabled.

Runtime We perform runtime checks during instantiation of new delegates to verify that the target object is not exposed and that the delegate invariant holds. We also check pre- and post-conditions during invocation of delegates; this is especially useful in case of multicast delegates. When exposing an object at runtime we also expose all delegates it

depends on; at the end of the expose block, we verify that the delegate invariants of all delegates we exposed hold. As in Spec#, we do not perform runtime checks for frame conditions or modification of fields involved in delegate invariants for performance reasons.

Delegate subtyping has been implemented using an additional attribute. In fact the CLR does not allow us to inherit from delegate classes. By wrapping delegates in instantiations of other delegates, we can provide upcasting operations without additional overhead for the programmer.

Tools It is currently not possible to directly compile and verify a program with the */verify* option of the Spec# compiler. This is due to the transformations we perform on the AST, which, for caching reasons, are not completely reversible. Compilation and verification must be executed in two separate steps.

6.2 Conclusions

This thesis provides a functioning framework for the verification of function objects which can be used in practical applications. The verification methodology implemented and extended in this work, allows to verify singlecast and multicast delegates. Thanks to delegate invariants, pre-conditions on the target object can be expressed without the need for leaking information about the target. Expressing frame conditions however, requires to either reveal additional information about the type of the target object, or to have very weak frame conditions, where all possible peers of the delegate are modified. Concerning *tightly coupled* scenarios, CTT delegates allow us to express all possible specifications of a target method. However this comes with a price: the type of the target object must be fixed during declaration of the delegate.

These limitations lead us to believe that function objects need additional research in order to be better understood and modeled. The current approach is based on dividing the problem of verifying delegates in subproblems, and then finding independent solutions. This leads to pre-, post-, frame-conditions, invariants, target type dependent frame conditions and CTT delegates. We expect future research to be able to find a unified solution for the problem.

The hope is that by using the tools we implemented, it will be possible in the future to devise a more sophisticated approach toward the verification of delegates.

6.3 Future Work

In future the priority should be given to completing the implementation of the methodology; this include:

- Extending the Spec# programming system to handle static invariants and implement the missing features necessary for allowing static invariants in delegates;
- Optionally: implementing one of the two extensions for disabling multicast delegates (see 4.2.2, page 56 and 4.2.2, page 57).

Future research can be oriented in combining Spec# delegates with the concepts of data groups [12] and dynamic frames, a special case of specification variables [10]. A new methodology based on dynamic frames could, in fact, improve the way frame conditions are cur-

rently handled. Dynamic frames have been implemented in a custom build of Spec# in [16].

Another possible research topic would propose to solve the ownership issues with events. In this direction the research of Friends [2] seems to be leading to some interesting scenarios that should be explored further.

Bibliography

[1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[2] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *MPC*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2004.

[3] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants, 2003.

[4] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *LNCS*, volume 3362, 2004.

[6] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A high-level modular definition of the semantics of c#. *Theor. Comput. Sci.*, 336(2-3):235–284, 2005.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

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

[9] ECMA International. Ecma-334: C# language specification, 2006.

[10] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.

[11] Gary. T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[12] K. Rustan M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Not.*, 33(10):144–153, 1998.

[13] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[14] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, 2005.

[15] Peter Müller and Joseph N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007. To appear.

[16] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008.

APPENDIX A

Short User's Manual

This appendix contains a short user's manual for delegate verification.

Declaring a delegate The syntax for delegate declaration remains the same as with normal delegates. Moreover it is possible to specify the contract using *requires*, *requires otherwise*, *ensures*, *modifies*, and *throws* clauses. Delegate invariants are declared as:

```
invariant when { T; E };
```

where T is the type of the target, and E is an expression. The special identifier *target* represents an instance of type T. All fields mentioned in a delegate invariant, must declare a `[Dependent(typeof(Del))]` attribute, where *Del* is the delegate declaring the invariant.

Modifies when clauses are declared as:

```
modifies when { T; F };
```

where T is defined as above and F is a field of target.

All non-CTT delegates have a default modifies clause for the target which is expressed with the *ModifiesDefault* attribute. The attribute takes as argument an enumeration with the following possible values:

1. *Modifies.Peers*
2. *Modifies.Target*
3. *Modifies.None*

The default modifies is inherited. Subtypes are not allowed to weaken the frame condition. Checks for this are only done for the default frame condition at compiler level.

Instantiating a delegate Delegate instantiation works like usual. Boogie will generate an error in case one of the conditions is not met.

Disabling a delegate Disabling delegates is accomplished by the

```
disable (o for T)
```

statement, where o is the target of delegates to be disabled and T is the type of the delegate. The frame condition of a method using the *disable* statement, must include

```
modifies p.***;
```

where p is a peer of the object o in the *disable* statement.

In order to get the enabled state of a delegate, it is possible to call

```
d.IsEnabled
```

where d is a delegate. The call is valid in both expressions and specification.

A method that violates the invariant of a delegate usually needs a precondition stating that no delegate with target equal to the object whose invariant is violated is enabled. This can be accomplished in the specification by calling

```
DelegateReferencesHolder.NoActiveDelegate(o, T)
```

where o and T are defined like in the case of disable.

Multicast delegates Multicast delegates work like usual. Boogie will generate an error in case one of the conditions for combining them is not met. Disabling a delegate always results in disabling all multicast delegates peers of the object o in the disable statement, unless Boogie can prove that the multicast delegate has a different target.

Closed Target Type Delegates While declaring a CTT delegate the user must provide the type for which the delegate is closed. The declaration is similar to the usual one, with the difference that the type in form $<T>$ is specified after the keyword delegate:

```
delegate <T> void MyDelegate();
```

CTT delegates do not contain delegate invariants and *modifies when* clauses. To access the target is it possible to use the *Target* property of the delegate itself, which is redefined in the delegate class to return an object of type *T!*. The syntax is similar to generis but with the following differences

- The type must be valid in the scope where the delegate is declared;
- No type constraints are allowed;
- The type is provided during declaration only.

Delegate Subtyping and Casting Delegate subtypes expressions can be downcasted to any of the the delegate base types directly without using cast operators. Using any cast operator results in an error. Consider the following example, where *SubDel* is a delegate subtype of *BaseDel*:

```
SubDel s = ...
BaseDel d = s;                  // allowed
BaseDel d = s as BaseDel;       // error
BaseDel d = (BaseDel) s ;       // error
```

The opposite operation is however never allowed. Upcasting can be safely accomplished as follows:

```
if (d != null && d.Target is Subdel) {
   Subdel d = d.Target;
}
```

APPENDIX B

Examples

B.1 Complete USBStick Example

The following code shows the complete version of the USBStick example used throughout this document.

```
using System;
using System.Text;
using Microsoft.Contracts;

[ModifiesDefault(Modifies.Target)]
public delegate void Archiver(object p);
   requires p != null;

public delegate void USBArchiver(object p) : Archiver
   invariant when { USBStick; target.IsLoaded };
   modifies when { USBStick; target.buffer };

public class USBStick {
    [Dependent(typeof(USBArchiver))]
    public bool IsLoaded;
    [SpecPublic][Rep] private StringBuilder! buffer;

    public USBStick()
        ensures this.IsLoaded;
    {
        this.IsLoaded = true;
        buffer = new StringBuilder();
    }

    public void Store(object p)
        requires p != null;
```

```
        requires this.IsLoaded;
        modifies this.buffer;
    {
        string value = p.ToString();
        expose (this) {
            buffer.Append(value);
        }
    }

    public void Eject()
        requires this.IsLoaded;
        requires DelegateReferencesHolder.NoActiveDelegate(this, typeof(USBArchiver));
    {
        expose (this) {
            /* eject the stick */
            this.IsLoaded = false;
        }
    }
}

public class Client {
    public static void Log(Archiver! logFile, string! s)
        requires logFile.IsEnabled;
        modifies logFile.**;
    {
        logFile(s);
    }
}

public class Program {
    public static void Main() {
        // create a new stick, a new USBArchiver and log a test
        USBStick stick = new USBStick();
        Archiver logFile = new USBArchiver(stick.Store);
        Client.Log(logFile, "Hello, World!");

        // eject the stick
        assume stick.IsLoaded; // the call to Log could have modified stick
        disable (stick for USBArchiver);
        stick.Eject();
    }
}
```

The example suffers from the fact that the type of the target of the delegate in method *Log()* is unknown. This requires us to add a *modifies logFile.*** modifies clause. An alternative version of the *Main()* method shows how knowledge about the target allows to improve frame conditions:

```
public class Program {
    public static void Main() {
        // create a new stick, a new USBArchiver and log a test
        USBStick stick = new USBStick();
        USBArchiver logFile = new USBArchiver(stick.Store);
        logFile("Hello, World");

        // eject the stick
        disable (stick for USBArchiver);
```

```
        stick.Eject();
    }
}
```

B.2 Events

The following program shows an implementation of the observer pattern using events.

```
using System;
using Microsoft.Contracts;

public delegate void UpdateHandler(string! status);
    requires status.Length > 0;
    ensures ((object)status) == ((object)old(status));

public class Observable {

    [Rep] public event UpdateHandler Changed;
    private string! status;

    invariant status.Length > 0;

    [NotDelayed]
    public Observable()
        ensures status.Length > 0;
    {
        status = "Initialized";
        base();
    }

    protected void Notify() {
        expose(this)
        {
            if (Changed != null && Changed.IsEnabled)
                Changed(status);
        }
    }

    public void DoSome() {
        expose (this)
            this.status = "Done something";

        Notify();
    }
}

public class Observer {
    public void Update(string! status)
         requires status.Length > 0;
         ensures ((object)status) == ((object)old(status));
    {
        Console.WriteLine("Status: " + status);
```

```
    }
}

public class Program {
    public static void Main() {
        Observable target = new Observable();
        Observer observer = new Observer();
        Owner.Assign(observer, target, typeof(Observable));
        target.Changed += new UpdateHandler(observer.Update);
        target.DoSome();
    }
}
```

As shown in the example, as long as the ownership can be expressed, it is possible to verify events. The additional post-condition of the delegate is necessary for the stability checks; without it Condition 4.2 cannot be verified. Stability checks could indeed be enhanced to consider immutable object; after such extension the additional post-condition would no longer be necessary because status cannot change.

APPENDIX C

SSC Syntactic Grammar

In this appendix we present the new syntax introduced by this work. The notation follows the ECMA standard [9].

Keywords The keyword `when` is introduced in the context of delegate invariants.

Delegate declaration Delegate declaration is specified for normal and CTT delegates.

$delegate - declaration :$

$\quad attributes_{opt}\ delegate - modifiers_{opt}\ \texttt{delegate}\ return - type$

$\quad\quad identifier\ type - parameter - list_{opt}\ \texttt{(}\ formal - parameter - list_opt\ \texttt{)}$

$\quad\quad delegate - base_{opt}\ type - parameter - constraints - clauses_{opt}$

$\quad\quad \texttt{[;}\ \texttt{|}\ \texttt{;}_{opt}\ delegate - contract\texttt{]}$

$\quad attributes_{opt}\ delegate - modifiers_{opt}\ \texttt{delegate}\ <class - type>\ return - type$

$\quad\quad identifier\ type - parameter - list_{opt}\ \texttt{(}\ formal - parameter - list_opt\ \texttt{)}$

$\quad\quad type - parameter - constraints - clauses_{opt}$

$\quad\quad \texttt{[;}\ \texttt{|}\ \texttt{;}_{opt}\ method - contract\texttt{]}$

$delegate - modifiers :$

$\quad delegate - modifier$

$\quad delegate - modifiers\ delegate - modifier$

$delegate - modifier :$

    new
    public
    protected
    internal
    private

$delegate - base :$

$\quad :\ class - type$

$delegate - contract$:
    $method - contract$
    $delegate - invariant$
    $ttd - modifies - clause$

$delegate - invariant$:
    `invariant when {` $class - type$ `;` $expression$ `};`

$ttd - modifies - clause$:
    `modifies when {` $class - type$ `;` $modifies - expression$ `};`

Statements The disable statement is added with the following syntax:

$embedded - statement$:
    . . .
    $disable - statement$

$disable - statement$:
    `disable (` $expression$ `for` $class - type$ `);`

APPENDIX D

Compiler Generated Methods

In this appendix we present the full bodies of compiler generated methods.

D.1 Compiler-Generated Delegate Classes

The following methods refer to the delegate declaration below:

```
delegate int LogFunction(string s);
    requires s != null;
    invariant when { Console; target.IsVisible };
    ensures result < s.Length;
```

Construct The *Construct()* method is used, instead of the normal constructor, to create new instances of delegates.

```
[NoDefaultContract]
public static LogFunction modopt(NonNullType) Construct(object @object,
    IntPtr method)
{
    LogFunction d = new LogFunction(object, method);
    Guard g = d.SpecSharp::FrameGuard;
    DelegateReferencesHolder.Add(d);
    LogFunction return value = d;
    try
    {
        if (!DelegateReferencesHolder.IsEnabled(return value))
        {
            throw new EnsuresException("Postcondition '<unknown condition>'
                violated from method 'LogFunction.
                Construct(System.Object,System.IntPtr)'");
        }
```

```
    }
    catch (ContractMarkerException)
    {
        throw;
    }
    LogFunction SS$Display Return Local = return value;
    return return value;
}
```

SafeInvoke The *SafeInvoke()* methods is used instead of the *Invoke()* method in a delegate. It carries the contract of the delegate (with the exception of delegate invariants) and performs runtime checks.

```
[Requires("[System.Compiler.Runtime]Microsoft.Contracts.
        DelegateReferencesHolder::IsEnabled([mscorlib]System.Delegate){this}"),
    Requires("::!=(string,string){$1,null}", Filename=@"...Simple.ssc",
        StartLine=7, StartColumn=12, EndLine=7,
        EndColumn=0x15, SourceText="s != null"),
    Ensures("::<(i32,i32){${i32,\"return value\"},$1@string::get_Length(){}}",
        Filename=@"...Simple.ssc", StartLine=8, StartColumn=11,
        EndLine=8, EndColumn=0x1c, SourceText="result < s.Length")]
public int SafeInvoke(string s) {
    try {
        if (!DelegateReferencesHolder.IsEnabled(this)) {
            throw new RequiresException("Precondition '<unknown condition>'
                violated from method 'LogFunction.SafeInvoke(System.String)'");
        }
        if (s == null) {
            throw new RequiresException("Precondition 's != null' violated
                from method 'LogFunction.SafeInvoke(System.String)'");
        }
    }
    catch (ContractMarkerException) { throw;}
    int return value = this(s);
    try {
        if (return value >= s.Length) {
            throw new EnsuresException("Postcondition 'result < s.Length'
                violated from method 'LogFunction.SafeInvoke(System.String)'");
        }
    }
    catch (ContractMarkerException) { throw; }
    int SS$Display Return Local = return value;
    return return value;
}
```

CheckInvariant The *CheckInvariant()* method performs runtime checks for delegate invariants.

```
[NoDefaultContract,
    Requires("[System.Compiler.Runtime]Microsoft.
        Contracts.Guard::FrameIsPrevalid(optional([System.Compiler.Runtime]
        Microsoft.Contracts.NonNullType,object),
        optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,
        [mscorlib]System.Type)){this,$typeof(LogFunction)}")]
public bool SpecSharp::CheckInvariant(bool throwException) {
```

```
    bool return value;
    try {
        if (!Guard.FrameIsPrevalid(this, typeof(LogFunction))) {
            throw new RequiresException("Precondition '<unknown condition>' violated
                from method 'LogFunction.SpecSharp::CheckInvariant(System.Boolean)'");
        }
    }
    catch (ContractMarkerException) { throw; }
    if ((base.Target is Console) && !((Console) base.Target).IsVisible) {
        if (throwException) {
            throw new ObjectInvariantException();
        }
        return value = false;
    }
    else {
        return value = true;
    }
    bool SS$Display Return Local = return value;
    return return value;
}
```

GetFrameGuard The *GetFrameGuard()* method returns the *Guard* of a delegate.

```
[CciMemberKind(CciMemberKind.Auxiliary)]
private static Guard SpecSharp::GetFrameGuard(object o)
{
    if (o == null) {
        throw new ArgumentNullException();
    }
    Guard guard2 = ((LogFunction) o).SpecSharp::frameGuard;
    Guard SS$Display Return Local = guard2;
    return guard2;
}
```

InitGuardSets The *InitGuardSets* method initializes the *Guard* of a delegate.

```
[NoDefaultContract, CciMemberKind(CciMemberKind.Auxiliary)]
private void SpecSharp::InitGuardSets() {
    this.SpecSharp::FrameGuard.AddRepFrame(this, typeof(MulticastDelegate));
}
```

FrameGuard The *FrameGuard* property returns the *Guard* of a delegate.

```
public Guard modopt(NonNullType) SpecSharp::FrameGuard {
    [NoDefaultContract, Delayed, Pure,
        CciMemberKind(CciMemberKind.FrameGuardGetter)]
    get {
        try {
            if (this.SpecSharp::frameGuard == null) {
                this.SpecSharp::frameGuard = new Guard(
                    new InitGuardSetsDelegate(this.SpecSharp::InitGuardSets),
                    new CheckInvariantDelegate(this.SpecSharp::CheckInvariant));
                this.SpecSharp::FrameGuard.EndWriting();
            }
```

```
        }
        catch (ContractMarkerException) { throw;}
        Guard return value = this.SpecSharp::frameGuard;
        Guard SS$Display Return Local = return value;
        return return value;
    }
}
```

D.2 Compiler-Generated Events Methods

The following compiler-generated methods for events are generated with respect to the following program:

```
public delegate void Del();

public class C {
    [Rep] public event Del RepEvent;
    [Peer] public event Del PeerEvent;
}
```

add_RepEvent The *add_RepEvent* method is used to combine the event with a new delegate.

```
[MethodImpl(MethodImplOptions.Synchronized),
    Requires("::==>(bool,bool){::!=(object,object){$coerce($1,object),null},
        [System.Compiler.Runtime]Microsoft.Contracts.Owner::
            Is(optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,object),
        optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,object),
        optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,
            [mscorlib]System.Type))
        {$1,this,$typeof(C)}}")]
public void add_RepEvent(Del modopt(NonNullType) value)
{
    try {
        if ((value != null) && !Owner.Is(value, this, typeof(C))) {
            throw new RequiresException("Precondition '<unknown condition>'
                violated from method 'C.add_RepEvent(optional(
                    Microsoft.Contracts.NonNullType) Del)'");
        }
    }
    catch (ContractMarkerException) { throw;}
    AssertHelpers.AssumeStatement("::==>(bool,bool){::&&(bool,bool)
        {::!=(Del,Del){this@C::RepEvent,null},::!=(optional([System.Compiler.
            Runtime]Microsoft.Contracts.NonNullType,Del),Del){$1,null}},
        ::==(optional([System.Compiler.Runtime]Microsoft.Contracts.
            NonNullType,[mscorlib]System.Type),optional([System.Compiler.Runtime]
            Microsoft.Contracts.NonNullType,[mscorlib]System.Type))
            {this@C::RepEvent@object::GetType(){},$1@object::GetType(){}}}");
    try {
        if ((((this.RepEvent == null) || (value == null))  1 :((this.RepEvent.GetType()
== value.GetType())  1 : 0)) == 0) {
            AssertHelpers.Assume(false);
        }
    }
```

```
    catch (ContractMarkerException) { throw; }
    C o = this;
    Guard.StartWritingFrame(o, typeof(C));
    DelegateReferencesHolder.StartWriting(this);
    try {
        this.RepEvent = (Del) Delegate.Combine(this.RepEvent, value);
    }
    finally {
        Guard.EndWritingFrame(o, typeof(C));
        DelegateReferencesHolder.EndWriting(this);
    }
}
```

remove_RepEvent The *remove_RepEvent* method is used to remove a delegate from the event.

```
[MethodImpl(MethodImplOptions.Synchronized),
    Requires("::==>(bool,bool){::!=(object,object){$coerce($1,object),null},
        [System.Compiler.Runtime]Microsoft.Contracts.Owner::
            Is(optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,object),
        optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,object),
        optional([System.Compiler.Runtime]Microsoft.Contracts.NonNullType,
            [mscorlib]System.Type))
        {$1,this,$typeof(C)}}")]
public void remove_RepEvent(Del value) {
    try {
        if ((value != null) && !Owner.Is(value, this, typeof(C))) {
            throw new RequiresException("Precondition '<unknown condition>'
                violated from method 'C.remove_RepEvent(Del)'");
        }
    }
    catch (ContractMarkerException) { throw; }
    C o = this;
    Guard.StartWritingFrame(o, typeof(C));
    DelegateReferencesHolder.StartWriting(this);
    try {
        this.RepEvent = (Del) Delegate.Remove(this.RepEvent, value);
    }
    finally {
        Guard.EndWritingFrame(o, typeof(C));
        DelegateReferencesHolder.EndWriting(this);
    }
}
```

add_PeerEvent and remove_PeerEvent The *add_PeerEvent* and *remove_PeerEvent* methods are similar to the methods shown above, but with the difference that the *value* is checked for having the same owner as the declaring object.

APPENDIX E

Found Bugs

In this appendix we present the most important bugs of the Spec# programming system found while implementing this work.

E.1 Generics

Boogie crashes when the program contains a modifies clause on a generic type.

```
using System;
using Microsoft.Contracts;

class C<T> {
   public T t;

   public C() { }

   public void SetT()
      modifies this.t;
   { }
}
```

Steps

```
$ ./ssc.exe Simple.ssc
$ ./Boogie.exe Simple.exe
```

Error

```
Spec# program verifier version 0.90, Copyright (c) 2003-2008, Microsoft.

Unhandled Exception: System.ApplicationException: Error in the application.
   at Omni.Parser.ContractDeserializer.System.Compiler.IContractDeserializer.
    ParseContract(MethodContract mc, String text, ErrorNodeList errs)
   at System.Compiler.MethodContract.get_Modifies()
   at System.Compiler.Method.get_Contract()
   at Microsoft.Boogie.PreTranslationVisitor.VisitMethod(Method method)
   at Microsoft.Boogie.PreTranslationVisitor.VisitTypeNode(TypeNode type)
   at System.Compiler.StandardVisitor.VisitClass(Class Class)
   at System.Compiler.StandardVisitor.Visit(Node node)
   at System.Compiler.StandardVisitor.VisitTypeNodeList(TypeNodeList types)
   at Microsoft.Boogie.PreTranslationVisitor.VisitModule(Module module)
   at Microsoft.Boogie.PreTranslationVisitor.VisitAssembly(AssemblyNode assembly)
   at System.Compiler.StandardVisitor.Visit(Node node)
   at Microsoft.Boogie.CilTranslator.TranslateCilToBoogie(Module module, Boolean
    needsDeserialization, Analyzer analyzer, ErrorHandler errorHandler)
   at Microsoft.Boogie.CilTranslator.TranslateCilToBoogie(String filename, List`1
    contractAssemblies, ErrorHandler errorHandler)
   at Microsoft.Boogie.BoogiePLMain.ProcessFileBasedOnType(FileType fileType,
     List`1 fileNames)
   at Microsoft.Boogie.BoogiePLMain.Main(String[] args)
```

Status Notified to the Spec# team.

E.2 Constructors in Contracts

The compiler allows to call constructor in method contracts, even if there is no reason for doing so. When trying to verify such programs Boogie crashes.

```
using System;
using Microsoft.Contracts;

class Target {
    public int i;

    public void Foo()
        requires new Target().i > 0;
    {}
}
```

Steps

```
$ ./ssc.exe /debug+ /target:library Simple.ssc
$ ./Boogie.exe /print:Simple.bpl Simple.dll
```

Error

```
Spec# program verifier version 0.90, Copyright (c) 2003-2008, Microsoft.
```

```
Unhandled Exception: Microsoft.Contracts.AssertException: Exception of type
    'Microsoft.Contracts.AssertException' was thrown.
   at Microsoft.Contracts.AssertHelpers.Assert(Boolean b)
   at Microsoft.Boogie.ExpressionTranslator.TranslateCall(MethodCall call,
    String heapName) in ExpressionTranslator.ssc:line 1020
   at Microsoft.Boogie.ExpressionTranslator.TranslateExpression(
    Expression expression, String heapName) in ExpressionTranslator.ssc:line 542
   at Microsoft.Boogie.GenerateModifiesContribution.VisitMemberBinding(
    MemberBinding binding) in FrameConditions.ssc:line 418
   at System.Compiler.StandardVisitor.Visit(Node node) in
    StandardVisitor.cs:line 316
   at System.Compiler.StandardVisitor.VisitExpression(Expression expression)
    in StandardVisitor.cs:line 992
   [...]
   at Microsoft.Boogie.CilTranslator.TranslateCilToBoogie(Module module, Boolean
    needsDeserialization, Analyzer analyzer, ErrorHandler errorHandler)
    in CilTranslator.ssc:line 153
   at Microsoft.Boogie.CilTranslator.TranslateCilToBoogie(String filename,
    List`1 contractAssemblies, ErrorHandler errorHandler) in
    CilTranslator.ssc:line 231
   at Microsoft.Boogie.BoogiePLMain.ProcessFileBasedOnType(
    FileType fileType, List`1 fileNames) in Main.ssc:line 263
   at Microsoft.Boogie.BoogiePLMain.Main(String[] args) in Main.ssc:line 106
```

Status Notified to the Spec# team.

E.3 Array Initialization

When arrays are initialized during declaration, there is a local contradiction in the method declaring the array. The contradiction only appears when the type of the elements of the array is a reference type and the array is passed to a method. Value types work as expected.

```
using System;
using Microsoft.Contracts;

public class A {
    public void Foo() {
        A[] a = { new A(), new A() };
        Bar(a);
        assert false;
    }

    public void Bar(A[] i) { }
}
```

Steps

```
$ ./ssc.exe /debug+ /target:library Simple.ssc
$ ./Boogie.exe /print:Simple.bpl Simple.dll
```

Error

```
Spec# program verifier version 0.90, Copyright (c) 2003-2008, Microsoft.

Spec# program verifier finished with 3 verified, 0 errors
```

The *assert false* statement in method *Foo()* should have risen an error.

Status Notified to the Spec# team; issues with array initialization are a known problem.

E.4 Structural Type Rules for Delegates

The typing rules for Spec# delegates are different from C#. In C# normal (nominal) rules are applied, while in Spec# delegates structural rules are used. The following code complies in Spec# but not in C#:

```
using Microsoft.Contracts;
using System;

class A { }

delegate void Del1(A a);
delegate void Del2(A a);

class Program {
    public static void Foo(A a) { }

    public static void Main() {
        Del1 d1 = new Del2(Foo);
    }
}
```

The delegate instantiation is modified by the Spec# compiler in:
`Del1 d1 = new Del1(new Del2(Program.Foo).Invoke)`.

Status Notified to the Spec# team. The problem is solved in our version of Spec#.

E.5 Out of Band Contracts and Axioms Contradiction

When using library classes inheriting from *System.Reflection.MemberInfo*, there is an axiom violation in the generate BoogiePL code. The problem is caused by the fact that out of band contracts for some classes are not included in the verification. This problem is made clear by *MethodInfo* because of the fact that it extends *MemberInfo* which is an immutable class. Since Boogie does not use the out of band contract for *MethodInfo*, it emits the latter as mutable class; resulting in an axiom violation.

```
using System;
using Microsoft.Contracts;
using System.Reflection;

delegate void Del();
```

```
public class Program {
    static void Main(string![]! args) {
        MethodInfo m = typeof(Program).GetMethods()[0];
        assert false;
    }
}
```

```
1 axiom IsImmutable(System.Reflection.MemberInfo) &&
2     AsImmutable(System.Reflection.MemberInfo)
3         == System.Reflection.MemberInfo;
4 axiom !IsImmutable(System.Reflection.MethodBase) &&
5     AsMutable(System.Reflection.MethodBase)
6         == System.Reflection.MethodBase;
```

Status Notified to the Spec# team. The problem is patched in our version of Spec#.

E.6 Specification of Array Concatenation

Boogie in unable to prove useful properties after concatenation of arrays. More precisely, give a method that concatenates two arrays which are fields of a class, Boogie is unable to prove equivalence for the elements of the second array.

```
using System;
using Microsoft.Contracts;

public class Test {
    public static int[]! Combine(int[]! a, int[]! b)
        ensures result.Length == a.Length + b.Length;
        ensures forall { int i in (0:a.Length);
            result[i] == a[i] };
        ensures forall { int i in (0:b.Length);
            result[i+a.Length] == b[i] };
    { return null; }

    static void Combining() {
        int[] a = new int[1];
        a[0] = 1;
        int[] b = new int[1];
        b[0] = 1;
        int[] i = Combine(a, b);
        assert i[0] == 1;
        assert i[1] == 1;
    }
}
```

In the example above, the last assertion should hold.

Error

```
$ ./ssc.exe /debug+ /t:library Simple.ssc
$ ./Boogie.exe /translate:Combining Simple.dll
Spec# program verifier version 0.90, Copyright (c) 2003-2008, Microsoft.
Simple.ssc(22,3): Error: Assertion might not hold:  i[1] == 1

Spec# program verifier finished with 0 verified, 1 error
```

Status To be analyzed further.

Listings

1.1 Introductory example, based on the example provided in [15] . . . 12
2.1 Hello, World! . . . 16
2.2 Disassembled version of a program using delegates . . . 17
2.3 Combining delegates . . . 18
2.4 Observer pattern implemented with events . . . 19
2.5 Decompiled version of the observer pattern implemented with events . . . 20
2.6 A simple class representing a point . . . 24
2.7 BoogiePL translation of Listings 2.6 . . . 25
2.8 A simple USBStick . . . 27
2.9 USBStick with pre-condition on the target object . . . 28
2.10 Breaking a delegate invariant. . . . 30
3.1 Refinement checks . . . 36
3.2 Disabling a [Rep . . . 39
3.3 A first approach towards frame conditions for delegates . . . 43
3.4 Frame conditions with additional information about the target . . . 44
3.5 Refinement checks for frame conditions . . . 49
4.1 Static pre-conditions . . . 52
4.2 Post-conditions of multicast delegates . . . 53
4.3 Pre-conditions of multicast delegates . . . 54
4.4 Disabling delegates in a target of a delegate . . . 60
4.5 Subtyping and generics . . . 60

4.6 Delegate subtyping with multicast delegates 61

4.7 GUI programming with Events 62

4.8 Vector operations with CTT delegates 64

5.1 Public interface of DelegateReferencesHolder 73

5.2 Partial interface of a compiler generated class for delegate LogFunction 74

5.3 Construct() method. 75

5.4 SafeInvoke() method. 76

5.5 SafeInvoke() method for the example of LogFunction example. 76

5.6 Exposing an object. 77

List of Figures

4.1 Owneship trees for a GUI with events 62

5.1 Modifications of the class diagram of the ssc compiler. 71

5.2 Compilation pipeline of ssc. 72

5.3 Boogie pipeline. 72

6.1 State of the implementation. 80