A Modular Verification Methodology for C# Delegates

Peter Müller[1] and Joseph N. Ruskiewicz[2]

[1] Microsoft Research, USA, mueller@microsoft.com
[2] ETH Zurich, Switzerland, joseph.ruskiewicz@inf.ethz.ch

Abstract. Function objects are used to express higher-order features in object-oriented programs. C# provides the delegate construct to simplify the implementation of function objects. A delegate instance represents a method together with a target object. Sound reasoning about delegates requires that the precondition of the underlying method holds whenever a delegate is invoked. This is difficult to achieve if the method precondition depends on the state of the target object. Proving such a precondition when the delegate is invoked is in general not possible because properties of the target object are typically not known at the invocation site. Proving the precondition when the delegate is instantiated is not sufficient either because the state of the target might change before the delegate is invoked. In this paper, we present a verification methodology for C# delegates. Properties of the target object are expressed as invariant of the delegate. Our methodology keeps track when this invariant can be assumed to hold. It enables modular verification of interesting implementations and is proven sound.

1 Introduction

Higher-order features are a common programming idiom. Typical examples include a generic sort algorithm whose comparison method is passed as parameter, an algorithm that approximates an integral of a function which is passed as a method reference, and a GUI that stores references to methods that are called upon certain events.

In object-oriented programs, references to methods are encoded as *function objects*. A function object represents a method, possibly with some actual method arguments. Function objects are often implemented using the Command pattern, whose class diagram is shown in Fig. 1. A function object is an instance of class $ConcreteCommand$. As described by Gamma *et al.* [8], a function object stores exactly one actual argument of the underlying method, namely its target. The target is fixed when the function object is created. The function object is invoked by calling its $Execute$ method, which will call $Action$ on the stored target object.

In C#, the Command pattern is built into the programming language in the form of delegates [6]. Each delegate type corresponds to a $ConcreteCommand$ class. It prescribes the signature of the underlying method, but not its name. The name—like the target object—is determined when the delegate is instantiated.

We illustrate delegates by an implementation of a simple storage system. We use the delegate $Archiver$ (Fig. 2) to create function objects for the store methods of different archives. Method $Client.Log$ takes an $Archiver$ instance as parameter and invokes it. Class $TapeDrive$ (Fig. 3) implements such an archive. The boolean field $IsLoaded$ is $true$ if and only if a tape is loaded into the drive. Hence, the $Store$ method requires

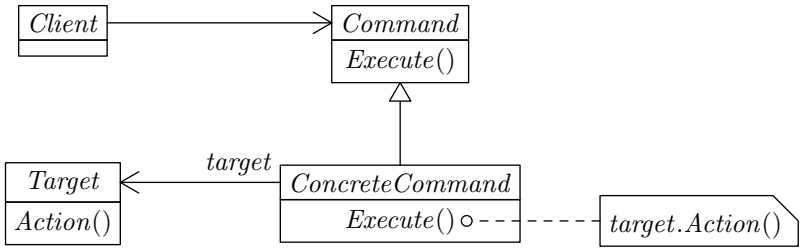

Fig. 1: The command pattern.

```
delegate void Archiver(object p)
  requires p ≠ null ∧ IsPeerConsistent(this) ;

class Client {
   static void Log(Archiver logFile, string s)
     requires logFile ≠ null ∧ IsPeerConsistent(logFile) ;
     requires s ≠ null ;
  { logFile(s) ; }
}
```

Fig. 2: A client of the storage system. We explain the annotations along with the presentation of our methodology.

IsLoaded to be *true*, and method *Eject* sets *IsLoaded* to *false*. In method *Main* (Fig. 4), *Archiver* is instantiated with method *Store* and target *tapeDrive*.

The invocation of a delegate instance triggers a call of the underlying method on the stored target object. In our example, the invocation of *logfile* in method *Client.Log* (Fig. 2) triggers a call of *tapeDrive.Store*. A sound verification methodology has to ensure that the requires clause of this method holds in the prestate of the call. This is difficult to achieve if the requires clause of the method depends on the state of the target object. This verification challenge was pointed out in an earlier paper [10], from which we took the storage example to show that our methodology can handle it.

The problem is illustrated by method *Store*, whose second requires clauses refers to the *IsLoaded* field of its target. Proving this condition when a delegate instance is invoked is not possible because properties of the target are typically not known at the invocation site. For instance, method *Client.Log* does not have any knowledge about the particular archive used by *logFile*. In fact, *logFile* could even represent a static method such that no target exists. Proving the condition when a delegate is instantiated with *tapeDrive.Store* is not sufficient either because the state of *tapeDrive* might change before the delegate is invoked. For instance, if method *Main* called *tapeDrive.Eject* before calling *Client.Log*, the invocation of the delegate would violate *Store*'s requires clause, even though the condition held when the delegate was instantiated.

In the Command pattern, the desired condition could be expressed as invariant of the function object; that is, class *ConcreteCommand* could declare the object invariant

```
class TapeDrive {
  bool IsLoaded ;

  void Store (object p)
    requires p ≠ null ∧ IsPeerConsistent(this) ;
    requires IsLoaded ;
  { ... }

  void Eject ()
    requires IsPeerConsistent(this) ∧ IsLoaded ;
    ensures ¬IsLoaded ;
  { expose (this for TapeDrive) { IsLoaded := false ; } }

  void ChangeMedia ()
    requires IsPeerConsistent(this) ∧ IsLoaded ;
  {
    expose (this for TapeDrive) {
      IsLoaded := false ; /* exchange media */ IsLoaded := true ;
    }
  }

  // Constructors and other methods omitted.
}
```

Fig. 3: An implementation of a tape archive. **expose** blocks are used by the Boogie methodology to denote regions in which an object invariant is potentially violated.

```
static void Main(string[] args) {
  TapeDrive tapeDrive := new TapeDrive() ;
  Archiver archiver := new Archiver(tapeDrive.Store) ;
  Client.Log(archiver, "HelloWorld") ;
}
```

Fig. 4: Main method for the storage example.

$target \neq \mathbf{null} \wedge target.IsLoaded$. This invariant is strong enough to guarantee that *Store*'s requires clause is satisfied whenever the function object is invoked. However, modular reasoning about invariants that depend on the state of several objects (here, the function object and its target) is difficult, and existing solutions are not suitable for function objects. Ownership-based invariants [11, 13] would require the function object to own the target object. This means that the target could be modified only through a single function object, which is clearly too restrictive. Solutions based on visibility-based invariants [13] lead to complicated proof obligations that essentially quantify over all instances of a delegate type, which are difficult to verify. Solutions based on friendship invariants [3] introduce a ghost state in each target object to keep track of function objects attached to it. This ghost state is used to impose the appropriate proof obligations about invariants of function objects whenever the state of the target is changed. Handling the ghost state leads to significant specification and verification overhead.

In this paper, we present a modular verification methodology for C# 1.0 delegates using delegate invariants. Our methodology is based on the Boogie methodology for object invariants [11] and can be adopted by the Spec# programming system [2]. It exploits the syntactic structure of delegates to generate simple proof obligations while keeping the annotation burden small. In particular, our methodology does not introduce any overhead for programs that do not use delegates, and very little overhead for the common applications of delegates.

Overview. This paper is organized as follows. Sec. 2 summarizes those parts of the Boogie methodology for object invariants that are used in the rest of the paper. Sec. 3 explains our methodology informally. The technical details including a soundness result are presented in Sec. 4. We review related work in Sec. 5 and offer conclusions in Sec. 6.

2 Background on Boogie Methodology

In this section, we summarize those parts of the Boogie methodology for object invariants [1] that are needed in the rest of this paper. The motivation for the design and the technical details are presented in an earlier paper [11].

Meaning of Invariants. To handle temporary violations of object invariants, the Boogie methodology introduces for every object the concrete field inv that represents explicitly whether the object invariant is required to hold. The inv field ranges over class names. If $o.inv <: T$ for an object o of type T (where $<:$ denotes the reflexive subtype relation), then o's invariants declared in class T and its superclasses must hold and we say o is *valid* for T. If o is not valid for T then the invariant of o declared in T is allowed to be temporarily violated and we say o is *mutable* for T. We say o is *fully valid* if it is valid for its dynamic type: $o.inv = \mathbf{typeof}(o)$.

The inv field is modified only by a special block statement: $\mathbf{expose}\,(o\ \mathbf{for}\ T)\,\{\,\mathcal{S}\,\}$. Before executing the statement $\mathcal{S}$, object o is *exposed*, that is, $o.inv$ is set to T's superclass. After the execution of $\mathcal{S}$, o is *un-exposed* by checking that the object invariant declared in class T holds for o and then setting $o.inv$ back to T.

Since the update of a field $o.f$ potentially breaks the invariant of o, $o.f$ is allowed to be assigned to only at times when o is mutable for the class F that declares f. To enforce this policy, each update of $o.f$ is guarded by an assertion $\neg o.inv <: F$. The assertions for field updates and **expose** statements ensure that throughout the program execution the following program invariant holds:

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

where $Inv_T(o)$ expresses that the invariant declared in class T holds for object o. The quantification ranges over non-null allocated objects.

Aggregate Objects. The Boogie methodology uses *ownership* to handle invariants of aggregate objects, that is, an aggregate object is the owner of its component objects. Updating a field of a component object potentially affects the invariant of the aggregate object. Therefore, the Boogie methodology enforces that the aggregate object is mutable whenever one of its component objects is mutable.

The Boogie methodology encodes ownership by adding a field $owner$ to each object. This field ranges over pairs $\langle obj, typ \rangle$, where obj is the owner object and typ is a

superclass of the dynamic type of obj at which the ownership is established. Like inv, $owner$ cannot directly be assigned to. The owner of an object is set when the object is created. Because it would be a distraction in this paper, we omit the program statement for changing the $owner$ field (but see [11]).
The relation between the validity of an aggregate object and its components is expressed by the following program invariant:

$$\text{P2: } (\forall\, o, T \bullet\ o.inv <: T \Rightarrow (\forall\, p \bullet\ p.owner = \langle o, T\rangle \Rightarrow\ p.inv = \mathbf{typeof}(p)\ ))$$

We say that two objects are *peers* if they have the same owner. A method m typically requires its target and all of its peers to be fully valid, which allows m to call methods with the same requires clause on these objects. Moreover, m typically requires the owner of its target to be *sufficiently exposed*, that is, the owner object is mutable for the owner type. This allows m to expose its target. An object o that satisfies these two requirements, is called *peer-consistent*. In specifications, peer-consistency of an object o is expressed by the following predicate:

$$\begin{aligned} IsPeerConsistent(o)\ \equiv\ & (\forall\, p \bullet\ p.owner = o.owner \Rightarrow\ p.inv = \mathbf{typeof}(p)\ ) \wedge \\ & (o.owner.obj \neq \mathbf{null} \Rightarrow \neg o.owner.obj.inv <: o.owner.typ) \end{aligned}$$

By the program invariants of the Boogie methodology, peer-consistency of an object o implies that the invariants of o, o's peers, and all objects owned by these objects are fully valid.

Static Verification. The proof rules of the Boogie methodology are formulated as assertions, which cause the program execution to abort if evaluated to $false$. For static verification, each assertion is turned into a proof obligation, which is proved using an appropriate program logic. As justified by earlier work [11], one may assume the program invariants for this proof. All of the proof obligations can be generated and shown modularly. That is, a class C can be verified based on the specifications of the classes used by C, but without knowing the complete program in which C will be used.

3 Main Concepts

In this section, we present our verification methodology informally. Our presentation will focus on single cast delegates, that is, delegates with exactly one underlying method and target. C# also provides for multicast delegates whose invocation triggers calls to several methods and targets. An extension of our methodology to multicast delegates follows directly from single cast delegates.

3.1 Delegate Specifications and Refinement

In the Command pattern (Fig. 1), invocations of function objects are verified using the specifications of method $Execute$. Since $Execute$ simply calls $target.Action$, the requires clause of $Execute$ must be strong enough to guarantee $Action$'s requires clause, and the converse holds for ensures clauses. In other words, the specification of $Action$ must refine the specification of $Execute$.

To adapt this approach to delegates, we associate each delegate declaration with a specification similar to method specifications. In this paper, we focus on requires and ensures clauses for delegates and assume that frame conditions are encoded in the

ensures clause. When a delegate D is instantiated with a method $o.m$, one has to prove that m's specification (with o for $\mathbf{this}$) refines D's specification. More precisely, one has to prove that D's requires clause is stronger than m's and that D's ensures clause is weaker than m's when D's requires clause holds. At the invocation site of the delegate, it suffices to prove that the requires clause of D holds, which implies that the weaker requires clause of m holds as well. Conversely, one may assume D's ensures clause after the invocation.

As explained in Sec. 2, most methods in the Boogie methodology require their target to be peer-consistent. To support this idiom, we arrange for delegate instances and their target objects to be peers. Therefore, we may assume that the target is peer-consistent whenever the delegate instance is peer-consistent. The peer relationship between a delegate instance d and its target o is established when d is created and maintained afterwards. Therefore, the following program invariant holds in all execution states.

$$\text{P3: } (\forall\, o, d \bullet\ d.target = o \wedge d.inv = \mathbf{typeof}(d) \Rightarrow d.owner = o.owner\ )$$

The refinement of specifications is illustrated by the instantiation of delegate $Archiver$ (Fig. 2) with $tapeDrive.Store$ in method $Main$ (Fig. 4). We ignore for the moment the second requires clause of method $Store$ (Fig. 3), which will be discussed in the next subsection. $Archiver$'s requires clause implies $Store$'s first requires clause because (1) the first conjunct, $p \neq \mathbf{null}$, appears in both requires clauses, and (2) because of program invariant P3, the target is peer-consistent whenever the delegate instance is peer-consistent. The default ensures clause, $\mathbf{true}$, of $Store$ trivially implies the default ensures clause of $Archiver$. Still ignoring $Store$'s second requires clause, $Store$'s specification refines the specification of $Archiver$, which allows us to verify the delegate instantiation in method $Main$. When the delegate is invoked in method $Client.Log$ (Fig. 2), we have to prove that the requires clause of the delegate is satisfied, which follows trivially from the requires clause of $Client.Log$.

Equipping delegates with requires and ensures clauses, and checking a refinement relation when a delegate is instantiated allows us to verify most applications of delegates. We looked at all delegate instantiations in Microsoft's compiler framework CCI and the Spec# compiler. The vast majority of delegates are instantiated with static methods, for which the methodology introduced so far is sufficient as static methods do not have target objects. It is also sufficient for instance methods whose requires clauses do not refer to the state the target besides requiring validity or peer-consistency. In the rest of this section, we discuss how to handle the remaining cases such as method $Store$, whose second requires clause requires the $IsLoaded$ field of the target to be $true$.

3.2 Delegate Invariants

We allow delegates to declare invariants that may refer to the state of the target. Analogously to C#, we assume that each delegate has an immutable field $target$ that holds a reference to the target. An invariant of the form $\mathbf{invariant\ for}\ T\ \mathbf{is}\ P(target)$ expresses that if the target is a non-null object of class T then it satisfies P. Such an invariant declared in a delegate type D is desugared into the invariant $o.target\ \mathbf{is}\ T \Rightarrow P((T)o.target)$. Note that our notation implicitly casts $target$ to T. In our example, an invariant for $Archiver$ could require that if its target is a reference to a $TapeDrive$ object, then its $IsLoaded$ field is $true$:

invariant for $TapeDrive$ **is** $target.IsLoaded$;

With this invariant, it is trivial to show that the specification of $Store$ refines the specification of $Archiver$, in particular, that $Store$'s second requires clause $IsLoaded$ is implied by $Archiver$'s require clause $IsPeerConsistent(\mathbf{this})$. By program invariant P1, peer-consistency of the $Archiver$ instance implies that its invariant holds. With the appropriate substitution, this immediately yields $Store$'s second requires clause. Therefore, the instantiation of $Archiver$ in method $Main$ (Fig. 4) verifies.

Delegate Subtypes. As illustrated by the above invariant, delegate invariants specify a type for the $target$ object in order to access its fields. This means that the specifier of the delegate has to foresee that the delegate might be instantiated with a method of that type. This deprives delegates of their flexibility. In particular, adding a new class $DiskDrive$ with method $Save$ to the program in general requires an additional invariant for $Archiver$, which cannot be added without changing the existing code.

To solve this problem, we allow programmers to declare subtypes of delegates, which may refine the specification of the supertype. Instead of adding the above invariant to $Archiver$, we declare the following subtype:

delegate $TapeArchiver$: $Archiver$
  invariant for $TapeDrive$ **is** $target.IsLoaded$;

Delegate subtypes prescribe identical signatures as their supertypes, which is not repeated in the subtype declaration. Moreover, subtypes inherit the specifications of their supertypes to enforce behavioral subtyping [5], but they may refine the inherited specifications. In particular, delegate subtypes are allowed to declare additional invariants.

To make use of the invariant of $TapeArchiver$, we have to adapt method $Main$ (Fig. 4) to instantiate $TapeArchiver$ rather than $Archiver$:

$Archiver\ archiver := \mathbf{new}\ TapeArchiver(tapeDrive.Store)$;

Since $TapeArchiver$ is a subtype of $Archiver$, the instance $archiver$ can be passed to method $Client.Log$ without further adaptations. In particular, $Client.Log$ (Fig. 2) need not be aware of the existence of the delegate subtype.

Note that delegate subtypes are merely a specification construct that allows us to associate invariants with delegates. In particular, they do not affect program execution. When a program is compiled, all occurrences of delegate subtypes can be replaced by their supertypes, and the subtype declarations can be eliminated.

Maintaining Delegate Invariants. Our verification methodology treats delegate instances basically like other objects. In particular, every delegate instance d has a field inv that indicates which invariants of d may be assumed to hold.

The invariant of a delegate instance d may depend on the immutable field $d.target$ and on fields of the object referenced by $d.target$. Therefore, the only operations that potentially violate d's invariant are modifications of the state of d's target. Consequently, programs never have to expose d in order to change its own state, but d must be exposed before its target object is modified. In other words, whenever a field $o.f$ is modified, we have to enforce that all delegate instances whose invariants depend on $o.f$ are exposed. We achieve that as follows:

1. *Visibility requirement*: If a delegate D declares or inherits an invariant for T then class T must contain the *dependent clause* $\mathbf{dependent}\ D$. Otherwise, D's invariant is not admissible and will be rejected by the compiler. Conversely, if a class T contains $\mathbf{dependent}\ D$ then delegate D must declare or inherit an invariant for T. Otherwise, T's specification is not admissible.
2. *Automatic expose*: We adapt the semantics of $\mathbf{expose}$ as follows. Besides object o, $\mathbf{expose}\ (o\ \mathbf{for}\ T)$ exposes each instance d of a delegate D where $d.target = o$ and D is mentioned in a dependent clause of class T. At the end of the $\mathbf{expose}$ block, d's invariants are checked and d is un-exposed.

The visibility requirement ensures that the dependent clause of a class T lists all delegates whose invariants are potentially broken by updates of fields of T. This allows us to determine in a modular way all the invariants that have to be checked at the end of an $\mathbf{expose}$ block.

The automatic expose guarantees that whenever a target object o is mutable for a class T, then all dependent delegate instances d are also mutable. The following program invariant states the contraposition of this property. $D \in dependents(T)$ expresses that class T contains a depends clause $\mathbf{dependent}\ D$.

$$\text{P4: } (\forall\, o, d, T, D \bullet\ d.target = o \wedge d.inv <: D \wedge D \in dependents(T) \wedge \mathbf{typeof}(o) <: T \Rightarrow o.inv <: T\ )$$

The visibility requirement and automatic expose guarantee that P1 also holds for delegate instances, where Inv_D denotes the desugared delegate invariant of delegate D.

The visibility requirement seems to be a severe restriction since it forces a class T and a dependent delegate D to be implemented together as they refer to each other in their dependent clause and delegate invariant. However, the requirement is not overly restrictive for the practical examples we have considered. First, as stated above, most delegates do not have invariants at all. Second, if the implementer of T wants to use an existing delegate D, they can declare a subtype of D that contains the invariant for T. This is illustrated by $Archiver$ and $TapeArchiver$. Third, if a delegate D needs to declare an invariant for an existing class T, it is not possible to add a dependent clause to T. In that case, one can declare a subtype S of T or a wrapper class S for T and establish the relation between D and S.

In our example, the invariant of $TapeArchiver$ refers to the $IsLoaded$ field of class $TapeDrive$. Therefore, we have to add the following dependent clause to $TapeDrive$:

$\mathbf{dependent}\ TapeArchiver\ ;$

Method $ChangeMedia$ (Fig. 3) illustrates how delegate invariants are verified. Because of the above dependent clause, the statement $\mathbf{expose}\ (\mathbf{this\ for}\ TapeDrive)$ exposes each $TapeArchiver$ instance d where $d.target = \mathbf{this}$. The subsequent update of $IsLoaded$ violates the invariant of d. However, since d is exposed, this violation is permitted by P1. At the end of the $\mathbf{expose}$ block, d's invariant is asserted. Since $IsLoaded$ is set to $true$ before the end of the block, this assertion holds.

It is important to understand how our methodology prevents delegate invocations when the requires clause of the underlying method does not hold. Consider an execution of method $ChangeMedia$ on target object o, and let d be a $TapeArchiver$ instance representing $o.Store$. We show how our methodology prevents $ChangeMedia$ from

invoking d between the two updates of $IsLoaded$, that is, when $IsLoaded$ is $false$ and, thus, the second requires clause of $Store$ does not hold. $TapeArchiver$ requires $IsPeerConsistent(\mathbf{this})$. Therefore, an invoker of d must prove that d is fully valid. However, according to program invariant P4, this is not the case while its target o is mutable, and the invocation does not verify. Note that if $TapeArchiver$ would not require $IsPeerConsistent(\mathbf{this})$ then an instantiation with $Store$ would not verify because $TapeArchiver$'s invariant is needed to prove the refinement relation.

Disabling Delegates. Method $ChangeMedia$ can be verified because it re-establishes the invariants of all $TapeArchiver$ instances before the end of the **expose** block. Other methods such as $Eject$ violate delegate invariants without re-establishing them. Such methods can only be verified under the requirement that the target does not have any dependent $TapeArchiver$ delegate instances. To ensure this requirement, we have to add the following requires clause to $Eject$:

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

This strong requires clause enables the verification of the method body, but is difficult to be satisfied by callers of $Eject$. In particular, if in some execution state a $TapeArchiver$ instance d refers to an object o then $o.Eject$ cannot be called in any subsequent execution state, even if d is not used anymore. This is because program verifiers typically do not model garbage collection, which means that formally d will never be deleted.

To support methods that violate certain delegate invariants, we provide a statement $\mathbf{disable}(D\ \mathbf{for}\ o)$, which disables all delegate instances with target object o that are valid for a delegate type D. A delegate instance d is disabled by exposing it—such that its invariant does not have to be maintained anymore—and by making d un-owned. The change of ownership is necessary to be able to un-expose d's owner—recall that P2 requires owned objects to be fully valid when the owner is valid. We do not provide a statement to re-enable a delegate instance since one can simply create a new instance.

It is generally necessary to execute $\mathbf{disable}(TapeArchiver\ \mathbf{for}\ o)$ before each call to $o.Eject$ to establish the above requires clause. This might seem tedious, but is only necessary if a delegate declares an invariant for class T and T contains methods that break this invariant. Such delegates are error-prone and we consider the overhead of adding **disable** statements acceptable in these rare cases.

4 Technical Treatment

In this section, we present the technical treatment of our methodology. We define precisely which delegate invariants are admissible, formalize the proof rules, and prove that our methodology is sound.

4.1 Admissible Delegate Invariants

Our methodology permits the invariant of a delegate instance d to depend on the field $d.target$ as well as the state of d's target object and all objects (transitively) owned by the target. However, to make the presentation and, in particular, the soundness proof self-contained; we use a slightly more restrictive definition of admissible delegates invariants here, which does not permit dependencies on the objects owned by the target.

Definition 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 form $\mathbf{this}.target$ or $\mathbf{this}.target.f$, where f is declared in T or a superclass of T and f is not one of the pre-defined fields inv or $owner$; (iii) D is mentioned in the dependent clause of T.*

4.2 Proof Rules

We define the proof rules of our methodology by translating the relevant statements into pseudo code, which makes the assertions and state changes explicit.

Delegate Instantiation. The instantiation of a delegate D with an instance method $o.m$ (Fig. 5) checks that the target o is non-null and valid for each class T that D depends on. The latter assertion is necessary to maintain program invariant P4. Note that the visibility requirement (Sec. 3.2) allows us to determine each dependee T modularly by inspecting the invariants of D and D's supertypes. Next, a fresh object d is allocated, its $target$ field is set to o, and its $owner$ field is set to $o.owner$ to make the delegate instance and the target peers. New delegate instances start off being fully valid. To maintain program invariant P1, we assert the invariant of D and D's supertypes before setting the inv field of the new instance to D. Finally, we check that the specification of m refines the specification of D. $Pre_D(d, p, h)$ and $Pre_m(o, p, h)$ denote the requires clauses of D and m, respectively. The ensures clauses are denoted by $Post_D(d, p, r, h, h')$ and $Post_m(o, p, r, h, h')$, where d and o are the targets of D and m, respectively, p is the (only) explicit parameter, r is the result, h is the heap of the prestate, and h' is the heap of the poststate.

$d := \mathbf{new}\ D(o.m)\ \equiv$
  $\mathbf{assert}\ o \neq \mathbf{null}$;
  $\#\mathbf{foreach}\ T\ \mathbf{such\ that}\ D \in dependents(T)\ \{\ \mathbf{assert}\ o.inv <: T\ ;\ \}$
  $d := \mathbf{new}\ D$;
  $d.target := o$; $d.owner := o.owner$;
  $\#\mathbf{foreach}\ E\ \mathbf{such\ that}\ D <: E\ \{\ \mathbf{assert}\ Inv_E(d)$;$\}$
  $d.inv := D$;
  $\mathbf{assert}\ (\forall\, p, h \bullet\ Pre_D(d, p, h) \Rightarrow Pre_m(o, p, h))$;
  $\mathbf{assert}\ (\forall\, p, r, h, h' \bullet\ Pre_D(d, p, h) \wedge Post_m(o, p, r, h, h') \Rightarrow Post_D(d, p, r, h, h'))$;

Fig. 5: Pseudo code for delegate instantiation.

Delegate Invocation. Delegate invocations are handled just like method calls (Fig. 6). The invoker must ensure that the requires clause holds before the invocation and may assume the ensures clause after the invocation. This reasoning is justified by the refinement relationship between the specifications of the delegate and the underlying method, which is checked when the delegate is instantiated. The $\mathbf{havoc}$ statement assigns arbitrary values to the variables for the current heap $\mathbb{H}$ and the result of the invocation v. This is necessary to make the verifier "forget" any prior knowledge about the variables

that are potentially modified by the delegate invocation. Before the havoc, the heap of the prestate is saved since the ensures clause may refer to it.

$v := d(p) \;\equiv$
  assert $d \neq \mathbf{null} \wedge Pre_D(d, p, \mathbb{H})$;
  $h := \mathbb{H}$; **havoc** $\mathbb{H}, v$;
  assume $Post_D(d, p, v, h, \mathbb{H})$;

Fig. 6: Pseudo code for delegate invocation. D is the static type of the delegate instance d.

Expose. Our methodology extends the **expose** statement of the Boogie methodology to automatically expose and un-expose dependent delegates (Fig. 7). We first discuss the parts we adopted from the Boogie methodology and then explain the extensions.

expose (o **for** T) { $\mathcal{S}$ } $\equiv$
  assert $o \neq \mathbf{null} \wedge o.inv = T$;
  assert $o.owner.obj \neq \mathbf{null} \Rightarrow \neg o.owner.obj.inv <: o.owner.typ$;
  #**foreach** $D \in dependents(T)$ {
    let $Dep_D := \{\, d \mid d.target = o \wedge d.inv = D \,\}$;
    foreach $d \in Dep_D$ { $d.inv :=$ **object** ;}
  }
  $o.inv :=$ **super**(T) ;
  $\mathcal{S}$;
  assert $(\forall$ **object** $p \bullet p.owner = \langle o, T \rangle \Rightarrow p.inv =$ **typeof**$(p))$;
  assert $Inv_T(o)$;
  $o.inv := T$;
  #**foreach** $D \in dependents(T)$ {
    foreach $d \in Dep_D$ {
      #**foreach** E **such that** $D <: E$ { **assert** $Inv_E(d)$;}
      $d.inv := D$;
    }
  }

Fig. 7: Pseudo code for **expose**. The extensions for delegates are highlighted by a shaded background.

The **expose** statement of the Boogie methodology implements a protocol that guarantees that owners are exposed before the objects they own, and that an object is exposed for a subclass before it is exposed for the superclass. Besides fields of o declared in T, the protocol allows $Inv_T(o)$ to depend on fields of objects owned by

$\langle o, T\rangle$ and on fields of o that are inherited from a superclass of T. In both cases, the protocol ensures that o is exposed for T before $Inv_T(o)$ is potentially violated by a field update.

In the pseudo code for $\mathbf{expose}\ (o\ \mathbf{for}\ T)$, this protocol is implemented as follows. First, we assert that o is non-null and valid for T, that is, has already been exposed for T's subclasses. Next, we assert that o's owner is sufficiently exposed. Finally, o is exposed by setting $o.inv$ to T's direct superclass, $\mathbf{super}(T)$. After the body of the $\mathbf{expose}$ block is executed, we assert that all objects owned by o in the type frame of T are fully valid. Then we un-expose o by setting $o.inv$ back to T. This update is guarded by an assertion that $Inv_T(o)$ holds (to maintain program invariant P1).

The automatic exposing of dependent delegates is done as follows. For each delegate type D in the dependent clause of T, we determine the set Dep_D of all delegate instances whose target is o and that are valid for D. These are the delegate instances whose invariants are potentially violated by assigning to fields of o declared in class T. We expose each of these delegate instances by setting its inv field to $\mathbf{object}$. The automatic un-exposing is done analogously. For each delegate instance that has been previously exposed, that is, is in one of the sets Dep_D, we assert its delegate invariant and set its inv field back to D. While non-delegate objects are exposed for one class at a time, the automatic exposes for a delegate instance d goes in one step from D to $\mathbf{object}$ and back. Therefore, it is necessary to assert all invariants that are declared in D and D's supertypes when d is un-exposed.

An important virtue of our methodology is that it causes no verification overhead for programs that do not use delegates, and very little overhead for programs whose delegates do not have invariants. In particular, the $\#\mathbf{foreach}$ loops in Fig. 7 can be unrolled statically by a compiler using the dependent clause of class T. If T does not have a dependent clause, the pseudo code for $\mathbf{expose}\ (o\ \mathbf{for}\ T)$ is identical to the Boogie methodology without delegates.

Disabling Delegates. As explained in Sec. 3, a delegate instance is disabled by exposing it and by making it un-owned, that is, setting its owner object to $\mathbf{null}$. The statement $\mathbf{disable}(D\ \mathbf{for}\ o)$ (Fig. 8) disables all delegate instances that are attached to a target object o and valid for delegate type D. Since disabling a delegate instance d changes its state, d's owner has to be sufficiently exposed.

$\mathbf{disable}(D\ \mathbf{for}\ o)\ \equiv$
  $\mathbf{assert}\ o \neq \mathbf{null}\ ;$
  $\mathbf{assert}\ o.owner.obj \neq \mathbf{null} \Rightarrow \neg o.owner.obj.inv <: o.owner.typ\ ;$
  $\mathbf{foreach}\ d\ \mathbf{such\ that}\ d.target = o \wedge d.inv <: D\ \{$
    $d.inv := \mathbf{object}\ ;\ d.owner := \langle \mathbf{null}, \_ \rangle\ ;$
  $\}$

Fig. 8: Pseudo code for $\mathbf{disable}$.

4.3 Soundness

Soundness of our methodology means that it is justified to assume program invariants P1–P4 when proving the assertions introduced by the methodology. In the following, we sketch the proofs of these program invariants. The proofs run by induction over the sequence of states of an execution of a program that is well-formed: that is, syntactically correct, type correct, and all invariants are admissible. The induction base is trivial since there are no allocated objects in the initial program state. For the induction step, we assume that the program invariant holds before the next statement s to be executed, and show that s preserves it.

Program Invariant P1. For non-delegate objects, P1 is guaranteed by the Boogie methodology. The soundness proof [11] remains valid because creation, exposing, and modification (that is, disabling) of delegate instances are guarded by the same proof obligations as the corresponding operations on non-delegate objects.

We proceed by proving P1 for delegate instances. That is, we show that statement s preserves the implication $o.inv <: T \Rightarrow Inv_T(o)$ for any delegate instance o and any type T. We consider all cases where s manipulates the state of an object; we omit all other cases for brevity.

Delegate Instantiation. Instantiation of a delegate D does not change the state of existing delegate instances. It remains to prove that the implication is established if o is the new delegate instance. After the instantiation, we have $o.inv = D$. The pseudo code asserts $Inv_E(o)$ for all E where $D <: E$, in particular, for $E = T$. Therefore, the implication holds.

Expose. The statement $\mathbf{expose}\ (x\ \mathbf{for}\ S)$ changes the inv field of x as well as each delegate instance $d \in Dep_D$, but nothing else. Since admissible delegate invariants do not refer to inv fields (Def. 1), $Inv_T(o)$ cannot be affected by these state changes. It remains to show that the implication is preserved if $o = x$ or $o = d$.

In both cases, the first update of inv preserves the implication by making its left-hand side stronger. By the induction hypothesis, the body of the $\mathbf{expose}$ block preserves the implication. Setting $x.inv$ from $\mathbf{super}(S)$ back to S affects the implication only if $T = S$. However, since $Inv_S(x)$ is asserted before the update of $x.inv$, the implication is preserved. Setting $d.inv$ from $\mathbf{object}$ back to D affects the implication only if $T = E$ for some supertype E of D. Again, since $Inv_E(d)$ is asserted for all such E before the update of $d.inv$, the implication is preserved.

Field Update. Consider an update $x.f := e$, where f is declared in a class F. We may assume that f is different from inv and $target$, which must not be directly assigned to. According to Def. 1, an invariant of T that mentions f must be an invariant for G, where $G <: F$ and T is mentioned in G's dependent clause ($T \in dependents(G)$). The update of $x.f$ is guarded by the assertion $\neg x.inv <: F$. By $G <: F$, we get $\neg x.inv <: G$. For o, T's invariant for G may depend on $x.f$ only if $o.target = x$. Moreover, we may assume $\mathbf{typeof}(o) <: G$, otherwise the desugared invariant holds trivially. Therefore, by contraposition on $P4$, this implies $\neg o.inv <: T$, and, thus, the left-hand side of the implication is $false$.

Disable. The statement $\mathbf{disable}(D\ \mathbf{for}\ x)$ changes the inv and $owner$ fields of each delegate instance where $d.inv <: D$ and $d.target = x$, but nothing else. Since admissible delegate invariants do not refer to inv and $owner$ (Def. 1), $Inv_T(o)$ cannot be affected by these modifications. It remains to show that the implication is preserved if $o = d$. This is trivially the case since the update of $d.inv$ makes the left-hand side of the implication stronger. □

Program Invariant P2. This program invariant is a consequence of the protocol that owners are exposed before the objects they own, the block structure of $\mathbf{expose}$, and the fact that newly created objects and delegate instances are fully valid when the constructor terminates. Since these arguments are identical for objects and delegate instances, the soundness proof from the Boogie methodology [11] remains valid. □

Program Invariant P3. The $owner$ field of an object is modified when an object or delegate instance is created and when a delegate instance is disabled. The proof is trivial for all three cases: (1) Newly created objects do not have any delegate instances attached. Therefore, the property holds trivially. (2) When a delegate is instantiated, its owner is set to the owner of its target, which establishes the property for the new instance. (3) When a delegate instance d is disabled, $d.inv$ is set to $\mathbf{object}$. Therefore, the left-hand side of the implication becomes $false$. □

Program Invariant P4. We prove that statement s preserves the implication of P4 for any object o, delegate instance d, class T, and delegate type D. It suffices to consider all statements s that modify the inv field.

Object Creation. Newly created objects do not have any delegate instances attached. Therefore, the implication trivially holds.

Delegate Instantiation. The instantiation $e := \mathbf{new}\ E(x.m)$ does not change the state of existing objects. We have to show that the implication is preserved for $e = d$. We may assume $d.inv <: D$, $D \in dependents(T)$, and $d.target = o$, otherwise the left-hand side of the implication is $false$. The instantiation establishes $e.inv = E$. By $d.inv <: D$ and $e = d$, we get $E <: D$.

From $D \in dependents(T)$ we conclude that D is mentioned in a dependent clause of T. By the visibility requirement (Sec. 3.2), D must declare or inherit an invariant for T. Since $E <: D$ and invariants are inherited, E also declares or inherits this invariant. By Def. 1, we know that E must also be mentioned in a dependent clause of T, that is, we have $E \in dependents(T)$. The pseudo code for delegate instantiation (Fig. 5) asserts $x.inv <: T$. Since $x = d.target = o$, this implies the right-hand side of the implication.

Expose. The statement $\mathbf{expose}\ (x\ \mathbf{for}\ S)$ changes the inv field of x as well as each delegate instance e in one of the Dep_A, but nothing else. This case is trivial if $o \neq x$ because in that case neither $o.inv$ nor $d.inv$ is changed.

For $o = x$, setting $d.inv$ to $\mathbf{object}$ makes the left-hand side of the implication stronger and, therefore, preserves the implication. Setting $o.inv$ to $\mathbf{super}(S)$ affects the implication only if $S = T$. The proof of this case is very similar to the proof for delegate instantiation. Again, we may assume $d.inv <: D$, $D \in dependents(T)$, and

$d.target = o$. Let $E = d.inv$; consequently, we have $E <: D$. Like for instantiation, we conclude $E \in dependents(T)$ and, by $S = T$, $E \in dependents(S)$. Since $d.target = o$ and $o = x$, we know $d \in Dep_E$ such that $d.inv$ is set to $\mathbf{object}$. Since D is a delegate type, this makes the left-hand side of the implication $false$.

By the induction hypothesis, the body of the $\mathbf{expose}$ block preserves the implication. The un-exposing after the body precisely un-does the modifications of the inv fields performed before the body. Therefore, it preserves the implication.

Disable. The statement $\mathbf{disable}(E\ \mathbf{for}\ x)$ modifies the inv field of each delegate instance where $e.inv <: E$ and $e.target = x$, but nothing else. The setting of $e.inv$ to $\mathbf{object}$ only makes the left-hand side of the implication stronger. □

5 Related Work

Eiffel agents [7] are similar to C# delegates, but more general since they allow the programmer to decide for each parameter, including the target, whether the actual argument is provided when the agent is instantiated (closed parameter) or when it is invoked (open parameter). Adapting our methodology to agents would require agent invariants that depend on all closed parameter objects. The corresponding visibility requirement might be too restrictive for certain applications of agents.

The work closest to ours is the version of the Boogie methodology described by Leino and Müller [11]. Besides the ownership-based object invariants that we also use in this paper, their work also supports visibility-based object invariants. Like our delegate invariants, visibility-based invariants may depend on fields of peers, provided that a visibility requirement is met. However, Leino and Müller's work requires programs to explicitly expose all objects whose visibility-based invariants are potentially affected by a field update. In general, without explicit references to the dependent peers, this obligation is hard to live up to. Our methodology exposes dependent delegate instances automatically when their target is exposed. Like the Boogie methodology, our work supports ownership transfer, but we omitted details due to space limitations.

The friendship methodology by Barnett and Naumann [3] simplifies the verification of visibility-based invariants by introducing ghost state to keep track of all dependent invariants of an object. This ghost state facilitates the exposing of dependent objects. The friendship methodology can handle implementations of function objects such as the Command pattern. Whereas the friendship methodology is very general, our methodology exploits the special syntactic structure of delegates to expose dependent delegate instances automatically, which removes the need to explicitly keep track of dependent invariants and, thus, reduces the annotation overhead.

Jacobs's version of Spec#, SpecLeuven [9], permits sound reasoning about delegates. Delegate instances own their target objects, which prevents these objects from being owned by other objects and, in particular, from being used in other delegate instances. Our solution does not impose this restriction.

Leino and Schulte [12] use history constraints to verify object invariants that are neither ownership-based nor visibility-based. If a history constraint guarantees that the state of an object o only evolves in ways that does not affect a dependent object invariant then there is no need to expose the dependent object before modifying o. We expect this approach to be a useful complement of our methodology, but not all targets

have strong history constraints. For instance, Leino and Schulte's methodology cannot handle our $TapeDrive$ example, because $IsLoaded$ is not a monotonic property.

Visible state semantics require invariants to hold in the pre- and post-states of all method executions. When invariants are allowed to depend on several objects such as delegate invariants, one needs a way of determining which invariants are potentially affected by a field update. These are exactly the invariants that our methodology exposes automatically when a target is exposed. Our methodology can be adapted to a visible state semantics using visibility-based invariants as described by Müller *et al.* [13].

The work by Börger *et al.* [4] has captured the semantics of delegates in an ASM model. This work has been fundamental in understanding the delegate construct. However, it is not suggestive how to use an ASM model to verify programs modularly.

6 Conclusions

We have presented a methodology for specifying and verifying C# delegates. Our methodology uses delegate invariants to express properties of the target object and allows one to reason about delegate invariants in a sound and modular way.

Our methodology requires significantly less specification and verification overhead than other techniques that could handle the Command pattern. This simplification is possible because delegates are essentially a stylized Command pattern, for which we can build special support into the verification methodology. We expect that similar methodologies can be developed for other design patterns, provided that the components of a design pattern are marked as such or that the idiom is supported by a special language construct.

Our methodology solves one of the two challenges related to function objects reported earlier [10], namely how to verify invocations of function objects. The other challenge is how to specify and verify invokers of function objects. We plan to address this challenge in future work. We also plan to implement our methodology into the Spec# programming system.

C# 2.0 delegates are more expressive than the delegates of C# 1.0, which we considered in this paper. For instance, C# 2.0 provides anonymous delegates, which may refer to local variables of the method body enclosing their declaration. Extending our methodology to C# 2.0 delegates is future work.

Acknowledgments. We are grateful to Rustan Leino for very helpful discussions and suggestions, in particular, his idea to use delegate subtypes. Thanks also to Bart Jacobs for his comments. Müller's work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project during his stay at ETH Zurich.

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2004.
3. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer-Verlag, 2004.

4. E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2-3):235–284, 2005.
5. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE*, pages 258–267. IEEE Computer Society Press, 1996.
6. C# language specification. ECMA Standard 334, June 2005.
7. Eiffel analysis, design and programming language. ECMA Standard 367, June 2005.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
9. B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, 2007.
10. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
11. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
12. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In R. de Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 80–94. Springer-Verlag, 2007.
13. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.