Master Thesis

Verification of design patterns

Author(s):
Hofer, Simon

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005763250 →

Verification of Design Patterns

Simon Hofer

Master Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

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

March 2009

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

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Spec# programming system is an automatic verification system. Verifying programs in Spec# works well for programs that use hierarchical object collaborations. However for programs with non-hierarchical object collaborations writing specifications can be complex, or the programs cannot be verified at all. Since programmers use design patterns to implement object collaborations, we have focused on specific design patterns to address the concrete problems.

In this thesis we have focused on the visitor pattern. In a visitor pattern the structure is usually hierarchical. But decoupling operations from the structure as separate objects, visitor objects, introduces non-hierarchical collaboration. In the current Spec# system visitor subclasses that refine the specifications of their base classes cannot be verified, and specifications of visitors cannot be decoupled from the structure they visit. Our solution is a methodology for the visitor pattern and its implementation as an extension of Spec#. With the extended system we have not only solved both problems, but we have also simplified the specification process of implementations of the visitor pattern by adding new language constructs specific for the visitor pattern.

Contents

Chapter 1

Introduction

Automatic verification systems have reduced the cost for verifying programs for programmers considerably. Such a system is Spec#, which consists of a programming language called Spec#, a compiler, and a static verifier called Boogie. The Spec# language extends C# with an ownership model, non-null types, method contracts, and object invariants.

The challenge for automatic verifications systems is to execute the proofs without interaction of a programmer. Since the program and its specifications are the only input to the verifier, the way a program is specified affects the verification process considerably in terms of the performance of the verification, and also in terms of the expressiveness of the results.

Spec# defines a methodology to verify programs sound and modularly. The methodology is built around an ownership model to control aliasing, which simplifies the specification and verification of programs with strictly hierarchical object collaborations. However for programs with non-hierarchical object collaborations writing specification can be complex or the programs can not be verified at all.

Since general optimizations for non-hierarchical object collaborations are hard to do, our approach is to focus on specific non-hierarchical object collaborations. As programmers use design patterns to implement object collaborations, designing specific methodologies for design patterns, and extending the Spec# system with implementations of these methodologies, can improve the verification of programs that implement these design patterns.

More precisely our focus was on the visitor pattern. In a visitor pattern the structure is usually hierarchical. But decoupling operations from the structure as separate objects, visitor objects, introduces non-hierarchical collaboration. Therefore the goal was to design a methodology for the visitor pattern, and extend the Spec# system with an implementation of the methodology, such that not only the verification of programs which implement the visitor pattern is improved, but also the specifications of implementations of the visitor pattern. With the extended Spec# system we want to be able to decouple the specifications for a visitor from the structure it visits, and we want to verify visitor subclasses that refine the specifications of their base classes. Both things are not possible in the current Spec# system.

Chapter 2

Background

2.1 The Spec# Programming System

The Spec# programming system[4] consists of a programming language called Spec#, a compiler, and a static verifier called Boogie[1].

The Spec# language is an extension of C# with an ownership model, non-null types, method contracts, object invariants, loop invariants, and assert statements. Other annotations and functionality is provided by a library delivered as part of the Spec# programming system in the form of C# attributes and static methods.

The Spec# compiler generates MSIL programs from Spec# programs retaining verification conditions in the form of annotations and calls to the Spec# library. The generated programs can be run on the standard .NET runtime. Additionally to the conventional C# compiler, the Spec# compiler generates runtime checks for the verification conditions. A verification condition that fails at runtime triggers an exception. Furthermore the compiler performs checks specific for the Spec# language. The checks include admissibility checks for invariants, method contracts, and the ownership model. Also non-null types are checked by the compiler with type checks and control-flow analysis.

Boogie is a static program verifier that proves the verification conditions of Spec# programs or programs written in BoogiePL[6], its internal language. Internally Boogie translates MSIL programs to BoogiePL programs before verification. For the verification, Boogie extracts the first-order verification conditions from the BoogiePL programs, by generating weakest preconditions[3] for each method, and proves them with one of its theorem provers. Currently the default theorem prover is Z3, but optionally Simplify can be used as well.

2.1.1 Spec#

The semantics of the Spec# language are defined by its methodology[2]. The Spec# methodology is designed around an ownership model, in which an object has at most one owner and the owner cannot be changed after it has been set. An important part of the methodology deals with object invariants. Since invariants do not have to hold all the time, the methodology encodes whether an object is valid as part of its state. A **valid object** refers to an object which invariants have to hold. An object which invariants can be broken is called an **exposed object**. Additionally all owned objects of a valid object are valid as well. An object is a peer of another object if it shares the same owner. If an object is **peer valid**, its peers are valid as well. An object is **consistent** if

it is valid and its owner is exposed. Similar to peer validity the peers of a **peer consistent** object are consistent as well.

Transforming a valid object to an exposed object, called **unpacking**, requires that the object is peer consistent, and ensures that the object is exposed. The reverse process, **packing** an object, requires that all of its invariants hold and that all of its owned objects are consistent, and ensures that the object is peer consistent. Since the unpack and pack command are often used in conjunction around a block of statements, the language introduces the expose statement. An **expose statement** for an object o around a block B

```
expose(o) B
```

Listing 2.1: Expose statement

is equivalent to the sequence found in literature: unpack o, execute B, and pack o.

The **default contracts** of a method require that the target object and all arguments are peer consistent and ensure that the target object, all arguments, and the return value are peer consistent. These default contracts guarantee that the invariants of the collaborating objects hold at the beginning and end of the method and that the collaborating objects can be updated, since they can be unpacked.

Pure methods are methods which have no side effects and can be used in method contracts. Pure methods are annotated with the C# attribute [*Pure*]. The default contracts for pure methods require and ensure peer validity instead of peer consistency for the collaborating objects.

Frame conditions are special postconditions, which specify what objects and fields a method may change. The default frame condition of a non-pure method allows the method to change all fields of its target object and all objects in the ownership domain of the target object. The **modifies clause** can be used to specify deviations from the default frame condition.

To know which references point to owned objects and which to peer objects, the fields, properties, and return values can be annotated with the C# attributes [*Rep*] or [*Peer*]. The [*Rep*] attribute denotes references to owned objects, and the [*Peer*] attribute denotes references to objects with the same owner as the corresponding target object.

If the type of a reference is a non-null type, it must not be a *null* reference, unless for field of objects that are being constructed. The type of a reference is a non-null type if an exclamation mark is appended to the name of the declared type. Here is an example of a local variable declaration of a non-null string.

```
string! s = "I␣am␣non-null";
```

Listing 2.2: Non-null string

2.1.2 BoogiePL

A BoogiePL program defines a set of global variables, constants, functions, axioms, procedures and implementations. The type system of BoogiePL defines the built-in types: integer (int), references (ref), boolean (bool), name (name), array, and any (any).

A **global variable** is defined by a name and type and can be used in functions, axioms, procedures, and implementations.

A **function** refers to a mathematical function which is defined by a name, a sequence of parameters and a type.

A **constant** refers to a mathematical function without parameters, a single function value and is defined by a name and a type.

The properties of functions and constants can be postulated by **axioms**. For example if we define a function *square* with a single integer parameter, we can define that its function value is the square value of the parameter as follows.

```
function square(int) returns (int);
axiom (forall x: int :: square(x) == x*x);
```

Listing 2.3: Example of a function and axiom

A **procedure** refers to the signature of a procedure in a computer program. It is defined by a name, input- and output parameters, and pre- and postconditions.

An **implementation** of a procedure refers to the body of a procedure in a computer program. It is defined by a sequence of blocks, which themselves have labels and sequences of commands.

2.1.3 Spec# to BoogiePL Translation

For the translation of Spec# programs to BoogiePL programs, the Spec# type system, the ownership model and other parts of the Spec# program are encoded as functions, constants and axioms.

The heap that holds all objects at runtime is modeled as global variable $\$Heap$. The type of the heap is a multi-dimensional array defined as $[ref, < x > name]x$, for simplicity we will refer to that type as $Heap$. The first dimension of the array maps references to objects. The second dimension maps field names to field values, where fields are represented as unique constants and field values have type any. For example a field access of the form $o.f$, where the field f is declared in class C and o is a local variable, is translated to $\$Heap[o, C.f]$.

Method signatures are transformed to procedures and method bodies are transformed to non-deterministic procedure implementations that retain the semantics.

For example the simple Spec# program

```
class Math {
    public int Square(int x)
        ensures result == x*x;
    {
        return x*x;
    }
}
```

Listing 2.4: Sample Spec# program

translates to the following simplified BoogiePL program, where we have omitted all functions, constants, and axioms.

```
procedure Math.Square$System.Int32(this: ref, x$in: int)
  returns ($result: int);
  ... // target object is peer consistent
  ... // frame condition
  modifies $Heap;
  // user-declared postconditions
```

```
  ensures $result == x$in * x$in;

implementation Math.Square$System.Int32(this: ref, x$in: int)
  returns ($result: int)
{
  var x: int stack0i: int;

  entry:
    x := x$in;
    goto block3162;

  block3162:
    goto block3383;

  block3383:
    stack0i := x * x;
    $result := stack0i;
    return;
}
```

Listing 2.5: Simplified BoogiePL program

2.2 Visitor Pattern

The main concept of the visitor pattern is to decouple operations from the structure they are applied to, in order to gain flexibility for adding new operations. But since this is only a concept, different programmers implement the pattern differently. As such, there exist many different variants of the visitor pattern.

In all common variants the operation is implemented as a separate class, called visitor, and in separate methods, called visit methods, for the structural types.

The main difference between different variants, also described by Gamma et al.[8], concerns the responsibility for the traversal of the structure. Essentially there are three possibilities: Either the structure is responsible for the traversal, we will refer to that variant as the structure controlled variant, the visitor is responsible for the traversal, we will refer to that variant as the visitor controlled variant, or finally there is a separate iterator which is responsible for the traversal. But as we have looked at different implementations of the visitor pattern from different open source projects such as NUnit, NRefactory, and Cecil, and of course we have looked at the Spec# compiler, and we have not found an implementation of the visitor pattern with an iterator for the traversal, we will only look at the first two variants.

Another difference concerns the mechanism which guarantees that for each type of structural object a different visit method on the visitor is called. The most common way to implement the mechanism, also described by Gamma et al.[8] and found in most implementations that we have studied, is to add a method, a so called accept method, to the structural classes, which when passed a visitor as argument dispatches to the corresponding visit method based on the structural type. This mechanism, calling a different method based on the types of the participating objects, is called double dispatch. For the structure and visitor controlled variant we will use that mechanism. Another common mechanism is often found for visitors of abstract syntax trees in compilers, for example in the Spec# compiler. The type is encoded as a separate field upon which value the corresponding visit method is selected. We will also introduce that variant and refer to

it as the direct visitor variant. Bishop[5] describes a similar mechanism which uses the already encoded runtime type information of objects instead of an additional field and reflection to call the corresponding visit method.

For the discussion about the visitor pattern we will use an example of an abstract syntax tree that represents expressions built of additions, subtractions, and constant values. The corresponding classes are defined as follows.

```
abstract class Expression {}

abstract class BinaryExpression : Expression {
    private Expression! left;
    public Expression! Left { get { return left; } }
    private Expression! right;
    public Expression! Right { get { return right; } }
}

class Add : BinaryExpression {}

class Subtract : BinaryExpression {}

class Literal : Expression {
    public int Value { get; set; }
}
```

Listing 2.6: Abstract syntax tree

As a first step we will implement a visitor which operation is to evaluate such an expression to a value.

2.2.1 Visitor Controlled Variant

Let us start with the visitor controlled variant. To be able to decouple the operation from the structural classes, we first add an accept method to the structural classes, to which we can pass a visitor object to initiate the execution of the operation that the visitor represents.

Later in the visitor class, we name it $EvalVisitor$, we want to be able to process the objects for each type with a different visit method. A method $VisitAdd$ to process objects of type Add, a method $VisitSubtract$ to process objects of type $Subtract$, and a method $VisitLiteral$ to process objects of type $Literal$.

Therefore we declare an abstract accept method in class $Expression$, and override the method in each of the concrete subclasses Add, $Subtract$, and $Literal$ to call the visit method corresponding to the type of the target object. The structural class are extended with the accept methods as follows.

```
abstract class Expression {
    public abstract void Accept(EvalVisitor! v);
}

abstract class BinaryExpression : Expression { ... }

class Add : BinaryExpression {
    public override void Accept(EvalVisitor! v) {
```

```
        v.VisitAdd(this);
    }
}

class Subtract : BinaryExpression {
    public override void Accept(EvalVisitor! v) {
        v.VisitSubtract(this);
    }
}

class Literal : Expression {
    ...
    public override void Accept(EvalVisitor! v) {
        v.VisitLiteral(this);
    }
}
```

Listing 2.7: Abstract syntax tree with accept methods

After we have implemented the double dispatch mechanism, we are ready to implement the visitor, with its visit methods *VisitAdd*, *VisitSubtract*, and *VisitLiteral*. Each visit method should evaluate the parameter expression. For *VisitAdd* and *VisitSubtract* we therefore evaluate the parameters subexpressions first, before we combine these results to the result of the parameter expression. To evaluate the subexpressions, we call the method *Accept* on the *Left* and *Right* subexpressions, and by storing the result of the evaluation always in the same place, in our case we use the field *total* of the visitor, we know where we can find the results after the calls. The resulting implementation of the visitor looks as follows.

```
class EvalVisitor {

    private int total;
    public int Total { get { return total; } }

    public void VisitAdd(Add! e) {
        int l = 0, r = 0;
        e.Left.Accept(this);
        l = Total;
        e.Right.Accept(this);
        r = Total;

        total = l + r;
    }

    public void VisitSubtract(Subtract! e) {
        int l = 0, r = 0;
        e.Left.Accept(this);
        l = Total;
        e.Right.Accept(this);
        r = Total;

        total = l - r;
    }

    public void VisitLiteral(Literal! e) {
```

```
        total = e.Value;
    }

}
```

Listing 2.8: class *EvalVisitor* for the visitor controlled variant

The reason why we call this variant visitor controlled is that the accept method calls, which initiate the traversal of the subexpressions, are located in the visit methods.

2.2.2 Structure Controlled Variant

Let us advance to the structure controlled variant. Again we have to add an accept method to the structural classes. We add an abstract accept method to class *Expression*, and override the method for the classes *Add*, *Subtract*, and *Literal*. But this time we include the traversal of the subexpressions directly in the accept methods, hence the name structure controlled variant. For the accept method in *Add* and *Subtract* we therefore call the accept method on the *Left* and *Right* subexpressions before dispatching to the visit method. Thus the structural classes are extended with the following accept methods.

```
abstract class Expression {
    public abstract void Accept(EvalVisitor! v);
}

abstract class BinaryExpression : Expression { ... }

class Add : BinaryExpression {
    public override void Accept(EvalVisitor! v) {
        Left.Accept(v);
        Right.Accept(v);
        v.VisitAdd(this);
    }
}

class Subtract : BinaryExpression {
    public override void Accept(EvalVisitor! v) {
        Left.Accept(v);
        Right.Accept(v);
        v.VisitSubtract(this);
    }
}

class Literal : Expression {
    ...
    public override void Accept(EvalVisitor! v) {
        v.VisitLiteral(this);
    }
}
```

Listing 2.9: Abstract syntax tree with accept methods

When writing the visitor, we have to consider that when a visit method is called for an expression, its subexpressions have already been visited. Therefore we have to use a data structure which can

store more than one of the intermediate values. To understand what structure we need, let us look more closely how the runtime deals with our previous implementation for the visitor controlled variant. For the visitor controlled variant, we used local variables to store the intermediate results while traversing the subexpressions. We were able to do that since the visitor is traversing the subexpressions recursively, meaning our local variables are still stored on the runtime stack in the stack frame of the visit method, while the recursive calls push and pop new stack frames from the stack to store their local values. We therefore can imitate this behavior by explicitly using a stack to store the intermediate values.

Therefore a visit method has to pop the intermediate results from its subexpressions from the stack, calculate its own result, and push the result back onto the stack. With this logic we implement the *EvalVisitor* as follows.

```
class EvalVisitor {

    private Stack<int>! values = new Stack<int>();
    public int Total { get { return values.Peek() } }

    public void VisitAdd(Add! e) {
        int l = 0, r = 0;
        l = values.Pop();
        r = values.Pop();

        values.Push(l + r);
    }

    public void VisitSubtract(Subtract! e) {
        int l = 0, r = 0;
        l = values.Pop();
        r = values.Pop();

        values.Push(l - r);
    }

    public void VisitLiteral(Literal! e) {
        values.Push(e.Value);
    }

}
```

Listing 2.10: class *EvalVisitor* for the structure controlled variant

2.2.3 Direct Visitor Variant

Finally let us look at the direct visitor variant. This variant is often used in compilers in the conjunction with abstract syntax trees. The idea is to represent different structural types as a single class when they share the same data. In that case the type is modelled as part of the data, as a field, to be still able to distinguish the different types.

For our expression example we can remove the classes *Add* and *Subtract* since they share the same data, and create objects of class *BinaryExpression* instead by encoding the different types as enumeration and field.

```
enum ExprType {
```

```
     Add, Subtract, Literal,
}

abstract class Expression {
    protected ExprType exprType;
    public ExprType ExprType { get { return exprType; } }
}

class BinaryExpression : Expression {
    // type field is ExprType.Add or ExprType.Subtract
    private Expression! left;
    public Expression! Left { get { return left; } }
    private Expression! right;
    public Expression! Right { get { return right; } }
}

class Literal : Expression {
   // type field is ExprType.Literal
   public int Value { get; set; }
}
```

Listing 2.11: Abstract syntax tree for direct visitor variant

Since we can now distinguish the type by inspecting the field, we do not need to create an accept method to use the double dispatch mechanism. We can implement a general visit method instead, which calls the corresponding visit method for the value of the type field. Other than that, we implement the different visit methods equivalently to the visitor controlled variant.

```
class EvalVisitor {

    public void VisitExpression(Expression! e) {
        switch(e.ExprType) {
            case ExprType.Add:
                VisitAdd((BinaryExpression)e); break;
            case ExprType.Subtract:
                VisitSubtract((BinaryExpression)e); break;
            case ExprType.Literal:
                VisitLiteral((Literal)e); break;
            default:
                break;
        }
    }

    private int total;
    public int Total { get { return total; } }

    public void VisitAdd(Add! e) {
        int l = 0, r = 0;
        Visit(e.Left);
        l = Total;
        Visit(e.Right);
        r = Total;

        total = l + r;
```

```
    }

    public void VisitSubtract(Subtract! e) {
        int l = 0, r = 0;
        Visit(e.Left);
        l = Total;
        Visit(e.Right);
        r = Total;

        total = l - r;
    }
    public void VisitLiteral(Literal! e) {
        total = e.Value;
    }

}
```

Listing 2.12: *EvalVisitor* for direct visitor variant

Chapter 3

Visitor Pattern: In Spec#

In this chapter we will pick up the abstract syntax tree example for the different variants, and discuss how it can be verified in the current Spec# system. In the first part we will discuss the ownership model with respect to the visitor pattern. In the second part we will discuss the specification and verification of the different variants, and finally extend the discussion to the verification of subclasses of the visitor and structure mutating visitors.

3.1 Ownership

Although the visitor pattern itself does not pose any restrictions to the kind of structures that can be visited, most implementations are used with trees or directed acyclic graphs, often in the form of a composite pattern. In the Spec# ownership model, the most convenient way to model trees and most directed acyclic graphs is to use Rep references for all children references. By using Rep references, we can already avoid some of the problems that would arise otherwise. For example the structure is guaranteed to be acyclic.

For our expression example this means that the *left* and *right* fields in class *BinaryExpression* have to be marked as Rep fields, and also the corresponding properties have to be annotated accordingly.

```
abstract class BinaryExpression : Expression {
    [Rep] private Expression! left;
    public Expression! Left { [Rep] get { return left; } }
    [Rep] private Expression! right;
    public Expression! Right { [Rep] get { return right; } }
}
```

Listing 3.1: Class *BinaryExpression* in Spec#

Another problem concerns the relationship between the structure and the visitor. Although a visitor might only read from the structure it visits, it usually has to update its own state to store the intermediate values. In that case the accept method cannot be pure and therefore the visitor has to be able to call non-pure methods on structural objects. In Spec#, this means that the target object and the arguments have to be peer consistent for an accept or visit method call. The general approach to achieve this is to expose the structural object that is being visited while traversing its substructure.

For the visitor controlled variant this means that we have to surround the accept method calls on the subexpressions within $VisitAdd$ and $VisitSubtract$ with an expose statement. For illustration the changes to $VisitAdd$ are as follows.

```
public void VisitAdd(Add! e) {
    int l = 0, r = 0;
    expose (e) {
        e.Left.Accept(this);
        l = Total;
        e.Right.Accept(this);
        r = Total;
    }

    total = l + r;
}
```

Listing 3.2: $VisitAdd$ with an added expose statement

Similarly we have to surround the accept method calls for the direct visitor and for the structure controlled variant with expose statements.

Furthermore if we look at the updated version of $VisitAdd$ more closely, we can see that since the default contracts for the calls to $Accept$ on $e.Left$ and $e.Right$ require that $e.Left$, $e.Right$, and the visitor are peer consistent, and we have exposed e, the visitor must not be peer with e, otherwise the visitor is not peer consistent. Therefore we generally have to specify that the visitor is not peer with the structure for a visit method call. We can specify this by adding the following precondition to the visit methods, where the target object is the visitor and e is the visiting expression.

```
requires Owner.Different(this, e);
```

Listing 3.3: Precondition for visit methods

The same reasoning also applies to the accept methods, therefore we add an equivalent precondition to the accept methods, where the target object is the expression and v the visitor.

```
requires Owner.Different(v, this);
```

Listing 3.4: Precondition for accept methods

3.2 Specification

To be able to write specifications for the $EvalVisitor$ for the different variants, we first have to define what the value of the evaluation of an expression is. Since it integrates the data of all nodes in an expression, a convenient way to describe it is a recursive mathematical function. Therefore we define the function $eval(e)$ for an expression e recursively as follows.

$$eval(e) = \begin{cases} eval(e.Left) + eval(e.Right) & \text{if e is an addition} \\ eval(e.Left) - eval(e.Right) & \text{if e is a subtraction} \\ e.Value & \text{if e is a literal} \end{cases}$$

To verify our visitor pattern in Spec#, we have to find a way to implement this mathematical function. In Spec#, the construct to represent a mathematical function are pure methods.

Since the mathematical function only depends on the structural classes, we implement $eval(e)$ as pure method of the structural objects. As we have subclass relationship between the structural classes, we can also split the different cases of the function to different pure methods of the concrete classes. To make the connection between the different methods, we can use abstract pure methods in the abstract classes, which are overridden by the pure methods in the concrete classes. The updated structural classes with additional pure methods can be defined as follows.

```
abstract class Expression {
    ...

    [Pure][Reads(ReadsAttribute.Reads.Everything)]
    public abstract int eval();
}

abstract class BinaryExpression : Expression { ... }

class Add : BinaryExpression {
    ...

    [Pure][Reads(ReadsAttribute.Reads.Everything)]
    public override int eval()
        ensures result == e.Left.eval() + e.Right.eval();
    {
        return e.Left.eval() + e.Right.eval();
    }
}

class Subtract : BinaryExpression {
    ...

    [Pure][Reads(ReadsAttribute.Reads.Everything)]
    public override int eval()
        ensures result == e.Left.eval() - e.Right.eval();
    {
        return e.Left.eval() - e.Right.eval();
    }
}

class Literal : Expression {
    ...

    [Pure][Reads(ReadsAttribute.Reads.Everything)]
    public override int eval()
        ensures result == Value;
    {
        return Value;
    }
}
```

Listing 3.5: Representation of $eval(e)$ as pure method of *Expression*

We were able to verify the visitor pattern with this representation. But the problem with this representation is that the mathematical function, which is specific to the visitor and the operation it represents, is now part of the specification of the structure. Although our intention was to

decouple the operation from the structure.

An attempt to solve this is to implement $eval(e)$ as pure method of the visitor. This can be done as follows.

```
[Pure][Reads(ReadsAttribute.Reads.Everything)]
public int eval(Expression! e)
    ensures e is Add ==> result == eval(((Add)e).Left) + eval(((Add)e).Right);
    ensures e is Subtract ==> result == eval(((Subtract)e).Left)
                                        - eval(((Subtract)e).Right);
    ensures e is Literal ==> result == ((Literal)e).Value;
{
    if (e is Add)
        return eval(((Add)e).Left) + eval(((Add)e).Right);
    else if (e is Subtract)
        return eval(((Add)e).Left) - eval(((Add)e).Right);
    else if (e is Literal)
        return ((Literal)e).Value;

    return 0;
}
```

Listing 3.6: Representation of $eval(e)$ as pure method of $EvalVisitor$

But to see why this representation does not work in Spec#, let us look at its translation to BoogiePL. For the static verification this method is translated to a Boogie function and axiom. A simplified version of the translation looks as follows.

```
function #EvalVisitor.eval([ref,<x>name]x, ref, ref) returns (int);
axiom forall $Heap: [ref,<x>name]x, this: ref, e$in: ref ::
    $IsNotNull(this, Visitor) && $IsNotNull(e$in, Expression) &&
    && ... // this is peer valid
    && ... // e$in is peer valid
    ==> ($IsNotNull(e$in, Add)
         ==> #EvalVisitor.eval($Heap, this, e$in) ==
             #EvalVisitor.eval($Heap, this,
                    $Heap[e$in, BinaryExpression.left])
             + #EvalVisitor.eval($Heap, this,
                    $Heap[e$in, BinaryExpression.right]))
        && ($IsNotNull(e$in, Subtract)
            ==> #EvalVisitor.eval($Heap, this, e$in) ==
                #EvalVisitor.eval($Heap, this,
                    $Heap[e$in, BinaryExpression.left])
                - #EvalVisitor.eval($Heap, this,
                    $Heap[e$in, BinaryExpression.right]))
        && ($IsNotNull(e$in, Literal)
            ==> #EvalVisitor.eval($Heap, this, e$in) ==
                $Heap[e$in, Literal.value]);
```

Listing 3.7: Boogie translation of the pure method

The problem with this representation is that the Boogie function also has the visitor as parameter, since the visitor is the target object of the method, although the original mathematical function only refers to the structural objects. This problem becomes visible if we need to prove that $eval(e)$ always represents the same value for a specific object e in a specific region of the program, when

the visitor in fact is changed in that region. This is essentially what we have when we want to prove that $eval(e.Left)$ stored in the local variable l does not change during the call to the accept method on $e.Right$ in $VisitAdd$ for the visitor controlled variant.

Using a static pure method does not work either, since the current Spec# system represents static methods as methods of class objects. Class objects are special objects which have no owner and thus may be modified from anywhere. In the translated Boogie function the class object is also a parameter, and since it can be modified from anywhere, we have the same problems as for the pure method of the visitor.

To conclude the discussion about the mathematical function, we have seen that to implement $eval(e)$ in Spec# we have to add pure methods to the structural classes. We will now discuss for each variant what specifications we have to write to specify the operation that the visitor represents. For convenience we will still use the mathematical function $eval(e)$ in the method contracts, but the function could be replaced by pure methods and pure method calls $e.eval()$.

3.2.1 Visitor Controlled Variant

Let us start with the visitor controlled variant. A representative for the correctness of our visitor is the following client, which asserts that the calculated value is actually the value that the expression represents.

```
void Run(Expression! e) {
    Visitor v = new Visitor();
    e.Accept(v);
    assert v.Total == eval(e);
}
```

Listing 3.8: Client of class $EvalVisitor$

The entry point for visiting an expression with a visitor is usually the accept method, since this entry point is well defined as opposed to a visit method call, for which there might exist multiple applicable visit methods.

To make sure that we have enough specifications to reason about the previous assertion, we have to add specifications to the visit and accept methods. First we have to add a postcondition to all visit methods that states that each visit method stores the calculated result $eval(e)$, where e is the parameter of the visit method, as the property $Total$.

```
ensures Total == eval(e);
```

Listing 3.9: Postcondition for visit methods

Secondly, we also have to specify that after the execution of an accept method the property $Total$ still refers to the result. Therefore we add the following postcondition to the accept methods.

```
ensures v.Total == eval(this);
```

Listing 3.10: Postcondition for accept methods

Proving the assertion with the help of the postcondition from the accept method call should be straight forward. The same applies for the proofs of the postconditions of the accept methods, since we can assume the postcondition for the corresponding visit method call.

The harder part is to prove the postconditions of the visit methods, since we have to use the mathematical definition of $eval(e)$ to prove that we calculate this recursive function. Now since

we are calculating one recursion step per visit method and each visit method represents a case of our function, the verifier can substitute $eval(e)$ in a proof of the postcondition of a visit method with the corresponding definition of the function. In Boogie this substitution corresponds to applying an axiom from the translation of the pure methods once, if we represent the function as pure methods. For the postcondition of $VisitAdd$ we then can show that $Total == eval(e.Left) + eval(e.Right)$, for which we can assume the postconditions of the accept method calls that specify when the property $Total$ holds which value from which subexpression. Similar reasoning applies for $VisitSubract$ and $VisitLiteral$.

What we can see from this example is that we have to write the same specifications in many different places. The postconditions of all visit methods plus the postcondition of all accept methods are all used to specify the same thing, namely that the calculated value is stored as the property $Total$ of the visitor.

3.2.2 Structure Controlled Variant

Now let us do the same for the structure controlled variant. The equivalent client looks as follows.

```
void Run(Expression! e) {
    Visitor v = new Visitor();
    e.Accept(v);
    assert v.Total == eval(e);
}
```

Listing 3.11: Client of class $EvalVisitor$

To have enough information to reason about this assertion we have to specify that the result is stored on top of the stack in the postconditions of the accept and visit methods. Therefore we add the following postcondition to the visit methods

```
ensures Total == eval(e);
```

Listing 3.12: Postcondition for visit methods

and the following postcondition to all accept methods.

```
ensures v.Total == eval(this);
```

Listing 3.13: Postcondition for accept methods

Again proving the assertion of the client and the postcondition of the accept methods is straight forward with these specifications under the assumption that we have specifications for the stack.

To prove the postcondition of the visit methods we have the problem that we need to know in the prestate of the visit methods where on the stack the different values from the subexpressions are stored. Therefore we have to add preconditions to the visit methods which state what results we expect in which positions. For $VisitAdd$ and $VisitSubtract$ we therefore add the following preconditions.

```
requires values[0] == eval(e.right);
requires values[1] == eval(e.left);
```

Listing 3.14: Precondition for $VisitAdd$ and $VisitSubtract$

Since the visit methods are called from the accept methods, we have to satisfy these preconditions within the accept methods. For a call to $VisitAdd$ we therefore have to look at the accept method declared in class *Add*. The problem for proving these preconditions is that our previous specifications only state which element is on top of the stack. What happens to the rest of the stack is not specified. Therefore we have to extend our specifications.

Within the accept method we know that all elements that we put on the stack during the accept method calls on the subexpressions should be popped by the subsequent visit method call. Additionally the visit method should push its own result back onto the stack. For the accept methods we therefore have to specify that the stack grows by one with respect to the prestate of the method execution and all elements that have already been on the stack in the prestate are not touched.

```
ensures v.values.Count == old(v.values.Count) + 1;
ensures ∀i ∈ [0 .. (v.values.Count − 2)] • v.values[i] == old(v.values[i]);
```

Listing 3.15: Postcondition for accept methods

For the visit methods $VisitAdd$ and $VisitSubtract$ the equivalent post condition looks differently. We pop the two subexpression results from the stack, calculate the sum of these results, and push that result back onto the stack. The stack size therefore decreases by one and again, all elements that have already been on the stack are not touched. This results in the following postconditions.

```
ensures values.Count == old(values.Count) - 1;
ensures ∀i ∈ [0 .. (values.Count − 2)] • values[i] == old(values[i]);
```

Listing 3.16: Postcondition for $VisitAdd$ and $VisitSubtract$

The postcondition for $VisitLiteral$ is the same as for the accept method, since we have no subexpressions and only have to push the value of the literal onto the stack. The rest of the stack is untouched.

```
ensures values.Count == old(values.Count) + 1;
ensures ∀i ∈ [0 .. (values.Count − 2)] • values[i] == old(values[i]);
```

Listing 3.17: Postcondition for $VisitLiteral$

In the end we can see that we have to add complex specifications for the structure controlled variant. Other than that we also have to write the same specifications in many different places.

3.2.3 Direct Visitor Variant

Let us also write specifications for the direct visitor variant. The equivalent client looks as follows.

```
void Run(Expression! e) {
    Visitor v = new Visitor();
    v.VisitExpression(e);
    assert v.Total == eval(v)
}
```

Listing 3.18: Client of class $EvalVisitor$

To have enough information to reason about this assertion we essentially have to write the same specifications as for the visitor controlled variant, with the difference that instead of the different accept methods we have the method $VisitExpression$. For all visit methods including $VisitExpression$ we therfore add the postcondition

```
ensures Total == eval(e);
```

Listing 3.19: Postcondition for visit methods

To prove that the method *VisitExpression* is correct we first have to make sure that for all inputs it calls a corresponding visit method. For this reason we first have to have a mapping of the enumeration types to classes. One way to do this is to specify the reverse mapping as object invariants. For class *BinaryExpression* we can specify that its type is either *ExprType.Add* or *ExprType.Subtract* by the following invariant.

```
invariant type == ExprType.Add || type == ExprType.Subtract;
```

Listing 3.20: Invariant for class *BinaryExpression*

Similarly we add an invariant to class *Literal* to specify that *Literal* maps to *ExprType.Literal*.

```
invariant type == ExprType.Literal;
```

Listing 3.21: Invariant for class *Literal*

If the resulting mapping from types to classes is unique, we can deduce whether the casts for the arguments of the different visit method calls are sound statically. Furthermore by proving the postcondition of *VisitExpression* we can guarantee that for each value of the type field a corresponding visit method is called.

Proving the postconditions for the different visit methods works almost the same as for the visitor controlled variant. The only difference is that the meaning of an *Expression* object is not determined uniquely by its runtime type but by its type field. We therefore have to add preconditions to the visit methods that imitate the checks that are done by the typechecker for the visitor controlled variant. Therefore we add the following preconditions to the visit methods.

```
requires e.ExprType == ExprType.Add;
```

Listing 3.22: Precondition for *VisitAdd*

```
requires e.ExprType == ExprType.Subtract;
```

Listing 3.23: Precondition for *VisitSubtract*

```
requires e.ExprType == ExprType.Literal;
```

Listing 3.24: Precondition for *VisitLiteral*

As we can see the direct visitor variant is very similar to the visitor controlled variant, also the specifications differ only slightly. In general the direct visitor variant can be seen as an optimization of either the visitor controlled variant or the structural controlled variant. Therefore, and because the variant only applies for specific applications, we will not incorporate the direct visitor variant for the design of a methodology of the visitor pattern.

3.3 Visitor Subclassing

In practice, the visitor class for which the structural classes implement the double dispatch mechanism is often an abstract class. The purpose of the abstract visitor class is only to declare the

signatures of the different visit methods. Concrete visitors are implemented as subclasses of that abstract visitor class. With this technique implementing a new operation on the structure is equivalent to creating a new concrete subclass of the abstract visitor. Since the mechanism for calling the visit methods is already defined for the abstract visitor class, no changes have to be made to the structural classes.

For our abstract syntax tree example, we use the visitor controlled variant, we can write such an abstract visitor class as follows.

```
abstract class Visitor {
    public abstract void VisitAdd(Add! e);
    public abstract void VisitSubtract(Subtract! e);
    public abstract void VisitLiteral(Literal! e);
}
```

Listing 3.25: Abstract visitor class

We then change the parameter types from $EvalVisitor$ to $Visitor$ in the different accept methods. As a next step we change the class $EvalVisitor$ to a subclass of class $Visitor$.

```
class EvalVisitor : Visitor {

    private int total; public int Total { get { return total; } }

    public override void VisitAdd(Add! e)
        ensures Total == eval(e);
    { ... }

    public void VisitSubtract(Subtract! e)
        ensures Total == eval(e);
    { ... }

    public void VisitLiteral(Literal! e)
        ensures Total == eval(e);
    { ... }

}
```

Listing 3.26: Class $EvalVisitor$ as subclass of the abstract visitor

With these changes our postconditions for the accept methods are no longer valid, since we were referring to the property $Total$ which is specific to class $EvalVisitor$. But since we need these postconditions to prove the postconditions from the visit methods, we need to preserve the information in the postconditions of the accept methods. For this reason we have to break modularity. As such we change the postconditions of the accept methods to the following postcondition.

```
ensures v is EvalVisitor ==> ((EvalVisitor)v).Total == eval(this);
```

Listing 3.27: Postcondition of accept methods

But since Boogie is optimized for modular verification, we cannot prove this postcondition in the accept methods. The problem is that Boogie uses the method contracts from the visit methods in the declared type for its proofs, which is $Visitor$, although the calls to the visit methods are dynamically bound. Therefore we cannot verify visitor subclasses in Spec#, when the visit methods refine the method contracts.

3.4 Structure Mutating Visitors

Up to this point we have only looked at visitors which do not modify the structure. Let us therefore look at a visitor that modifies the structure during traversal. We again look at an example based on the abstract syntax tree structure from before. But this time we remove the class *Subtract* for simplicity and focus on the visitor controlled variant. We then implement a visitor which operation is to remove all *Literal* objects with value zero from the abstract syntax tree, while preserving the original value of the expression.

The structural classes are then defined as follows with the assumption that our visitor will be called $NonZeroVisitor$.

```
abstract class Expression {
    public abstract void Accept(NonZeroVisitor! v);
}

class Add : Expression {
    [Rep] private Expression! left;
    public Expression! Left { [Rep] get; set; }
    [Rep] private Expression! right;
    public Expression! Right { [Rep] get; set; }

    public Add([Captured] Expression! left, [Captured] Expression! right)
        ensures this.Left == left && this.Right == right;
    {
        this.Left = left;
        this.Right = right;
    }

    public override void Accept(NonZeroVisitor! v) {
        v.VisitAdd(this);
    }
}

class Literal : Expression {
    public int Value { get; set; }

    public Literal(int value)
        ensures this.Value == value
    {
        this.Value = value;
    }

    public override void Accept(NonZeroVisitor! v) {
        v.VisitLiteral(this);
    }
}
```

Listing 3.28: Abstract syntax tree

We have also added ownership information and postconditions for the constructors. The meaning of the [*Captured*] annotation in Spec# is that the objects that are passed as corresponding arguments have to be unowned in the prestate of a call and might become owned in the poststate. We need to add these annotations since *Add* objects own their $Left$ and *Right* subexpressions,

as denoted by the Rep references. An assignment to *Left* or *Right* therefore changes the owner of the source object and this is only possible if the source object is unowned, since Spec# has no ownership transfer.

To implement the *NonZeroVisitor*, we travers the abstract syntax tree with our visitor recursively and remove all zero *Literal* objects from the tree. If we have removed a *Literal* object from a node of the tree, which in our case can only be an *Add* object, we have to propagate the changes recursively to the root of the tree. Meaning if an *Add* object remains only with a single subexpression, we can replace the *Add* object directly by its single subexpression. If an *Add* object has no subexpressions anymore, we can remove the whole *Add* object from the tree and propagate the change to the root.

If we are not bound to an ownership model, we can implement the class *NonZeroVisitor* as follows.

```
class NonZeroVisitor {
    private bool isZero;
    private bool isChanged;
    private Expression nonZero;
    public Expression NonZeroExpression { get { return nonZero; } }
    public override void VisitAdd(Add! e) {
        e.Left.Accept(this);
        if (isZero) {
            isZero = false;
            isChanged = false;
            e.Right.Accept(this);
            if (!isChanged) {
                nonZero = e.Right;
                isChanged = true;
            }
        } else {
            if (isChanged) {
                e.Left = nonZero;
                isChanged = false;
            }
            e.right.Accept(this);
            if (isZero) {
                isChanged = true;
                nonZero = e.Left;
                isZero = false;
            } else if (isChanged) {
                e.Right = nonZero;
                isChanged = false;
            }
        }
    }
    public override void VisitLiteral(Literal! e) {
        if (e.Value == 0) {
           isZero = true;
        } else {
           isZero = false;
        }
    }
}
```

Listing 3.29: Class *NonZeroVisitor*

The resulting expression without zero *Literal* objects is stored as private field *nonZero*, which is equivalent to the public property *NonZeroExpression*.

The problem with this implementation concerning ownership is that we are assigning the field *nonZero* to *e.Left* or *e.Right* in method *VisitAdd* when we have to replace the add object by its left respectively right subexpression. That means, we have to make sure that the values stored in the field *nonZero* are unowned. But since we are assigning objects to the field *nonZero* that are already part of the original abstract syntax tree, and therefore already have an owner, this is not possible.

As we can see from this implementation already, in Spec# we can only implement a restricted set of mutations on the structure with a visitor pattern. The mutations have to preserve the owners of the structural objects.

As an alternative we can copy the abstract syntax tree into a new tree during traversal. An equivalent visitor that copies the structure is implemented as follows.

```
class NonZeroVisitor {
    private Expression nonZero;
    public Expression NonZeroExpression { get { return nonZero; } }

    public void VisitAdd(Add! e) {
        Expression l = null, r = null;
        e.Left.Accept(this);
        l = nonZero;
        if (e.left != e.right)
            e.Right.Accept(this);
        r = nonZero;

        if (l != null && r != null) {
            nonZero = new Add(l, r);
        } else if (l != null) {
            nonZero = l;
        } else if (r != null) {
            nonZero = r;
        } else {
            nonZero = null;
        }
    }

    public void VisitLiteral(Literal! e) {
        if (e.value == 0) {
            nonZero = null;
        } else {
            nonZero = new Literal(e.Value);
        }
    }
}
```

Listing 3.30: Class *NonZeroVisitor* that duplicates the abstract syntax tree

Again, the duplicated expression without zero literals is stored as private field *nonZero*, which is equivalent to the public property *NonZeroExpression*.

The problem that we have to overcome with this implementation to satisfy the ownership model, is that we have to make sure that the object that is stored in the field *nonZero* is unowned. Moreover since we store the value of *nonZero* locally in variable *l* while calling the accept method on the subexpression *e.Right* in *VisitAdd* and *VisitSubtract*, we have to make sure that the object remains unowned after the call. To do so we can specify that the object has not existed before the method execution of *VisitAdd*, respectively *VisitSubtract*, by adding the following postconditions to all visit methods

```
ensures nonZero.IsNew;
```

Listing 3.31: Postcondition for visit methods

and to all accept methods.

```
ensures v.nonZero.IsNew;
```

Listing 3.32: Postcondition for accept methods

So far we have discussed and solved the problems with ownership. Let us continue with specifying the operation that the visitor represents.

We can define a mathematical function of type boolean that specifies whether an expression contains zero literals or not as follows.

$$noZero(e) = \begin{cases} noZero(e.Left) \wedge noZero(e.Right) & \text{if e is an addition} \\ e.Value \neq 0 & \text{if e is a literal} \\ true & \text{if e is null} \end{cases}$$

In combination with the function $eval(e)$ from before, we can add the following postconditions to the visit methods

```
ensures noZero(nonZero)
ensures eval(nonZero) == eval(e)
```

Listing 3.33: Postconditions for visit methods

and the corresponding postconditions to the accept methods

```
ensures noZero(v.nonZero)
ensures eval(v.nonZero) == eval(this)
```

Listing 3.34: Postconditions for accept methods

to specify that the visitor duplicates the abstract syntax tree into a new tree that represents the same value without zero literals.

The verification of these method contracts works equivalently to the verification of the method contracts of our previous evaluation visitor. As such the main challenges for structure mutating visitors concern the ownership model, for which we have found a representation when we duplicate the whole structure.

Chapter 4

Visitor Pattern: Methodology

In this chapter we introduce a methodology for the visitor pattern that is based on the Spec# methodology. The methodology that we introduce does not directly solve the problems that we have introduced in the last chapter. It rather formalizes the visitor pattern, such that we can improve the specification language and the verification of implementations of the visitor pattern with the Spec# system in a next step.

It is important to know that our methodology is an optional extension of the Spec# methodology. As such we introduce annotations to specify whether an implementation of the visitor pattern should follow our methodology. If an implementation of the pattern is not annotated, it is not affected by the rules and transformation that we describe.

Previously we have introduced three different variants of the visitor pattern. For the methodology, we decided to focus on the visitor controlled variant.

We will not cover the structural controlled variant, since it is less flexible in the sense of traversal and less modular since the structure still contains logic of the operations.

We will not cover the direct visitor variant since it is a very specific solution, which is therefore often not the first choice for applications of the visitor pattern.

This chapter is structured as follows. We will first introduce annotations and terms for the visitor pattern. In a next step we will discuss the different properties and rules of the pattern, before we list a formal definition of the admissibility rules and the translation function of the new constructs specific for the visitor pattern.

4.1 Definition

Before we describe the methodology for the visitor pattern, we have to define the scope of a visitor pattern. For our methodology we are always looking at a single instance of the visitor pattern. A single instance means that all structural classes are derived form one single structural class. The same holds for the visitor classes. All visitor classes are derived from one single visitor class.

To be able to identify the different classes, methods and parameters which are characteristic for the visitor pattern, we introduce annotations in the form of C# attributes. As a simplification we omit the *typeof* operator to convert a class to a type object for classes that are passed to arguments of the attribute constructors.

The first annotation is used to describe the structural classes. Later on we have to know which visitor classes corresponds to the structural classes, since the structural classes have to implement the dispatch mechanism for the visitor classes. Therefore we introduce the attribute $[Visitable(V)]$ with the meaning that the class it annotates is a structural class that implements the mechanism to dispatch to visitor objects of type V. Moreover we refer to such a class as visitable class for type V.

The mechanism to dispatch to visitor objects is implemented by a special method, the accept method. The attribute $[Accept(V)]$ describes that the annotated method is an accept method that dispatches to visitor objects of type V. We call such a method an accept method for type V.

We also have to know which parameter of an accept method corresponds to the visitor that it dispatches to. Therefore we introduce the attribute $[AcceptSubject]$ that annotates the corresponding parameter. We call the object that the parameter represents the accept subject.

To illustrate the annotations for the structural classes, here is an annotated version of the abstract syntax tree example.

```
[Visitable(EvalVisitor)] abstract class Expression {
    [Accept(EvalVisitor)]
    public abstract void Accept([AcceptSubject] EvalVisitor! v);
}

[Visitable(EvalVisitor)] abstract class BinaryExpression : Expression {
    private Expression! left;
    public Expression! Left { get { return left; } }
    private Expression! right;
    public Expression! Right { get { return right; } }
}

[Visitable(EvalVisitor)] class Add : BinaryExpression {
    [Accept(EvalVisitor)]
    public override void Accept([AcceptSubject] EvalVisitor! v) { ... }
}

[Visitable(EvalVisitor)] class Subtract : BinaryExpression {
    [Accept(EvalVisitor)]
    public override void Accept([AcceptSubject] EvalVisitor! v) { ... }
}

[Visitable(EvalVisitor)] class Literal : Expression {
    public int Value { get; set; }
    [Accept(EvalVisitor)]
    public override void Accept([AcceptSubject] EvalVisitor! v) { ... }
}
```

Listing 4.1: Abstract syntax tree example with annotations

Let us continue with the visitor classes. Visitor classes declare different methods, called visit methods, which process structural objects of a specific type.

To be able to describe the properties of the visit methods of a visitor, we have to know which type of structural objects the visitor may visit. The attribute $[Visitor(S)]$ describes that the annotated class is a visitor class that is able to visit objects of type S. We call such a class a visitor for type S.

The visitor class declares different visit methods, where each visit method is responsible for visiting objects of a specific type. The attribute $[Visit(S)]$ describes that the annotated method is a visit method that can visit objects of type S. We call such a method a visit method for type S.

Moreover we have to know which parameter of the visit method refers to the structural object. We introduce the attribute $[VisitSubject]$ to annotate the corresponding parameter. We call the object that the parameter represents the visit subject.

To illustrate the annotations of the visitor classes, here is an annotated version of the class $EvalVisitor$ from the expression example.

```
[Visitor(Expression)] class EvalVisitor {

    private int total;
    public int Total { get { return total; } }

    [Visit(Add)]
    public void VisitAdd([VisitSubject] Add! e) { ... }

    [Visit(Sub)]
    public void VisitSubtract([VisitSubject] Subtract! e) { ... }

    [Visit(Literal)]
    public void VisitLiteral([VisitSubject] Literal! e) { ... }

}
```

Listing 4.2: Class $EvalVisitor$ with annotations

4.2 Basic Methodology

Before we describe the methodology with formal admissibility rules and a translation function, we explain the rules in this section with respect to a single visitor class. We will show the extensions for visitor subclasses in the next section. We will make references to the formal rules where appropriate.

Let us begin with the visitable classes.

4.2.1 Visitable Class

The most important property of visitable classes concerning the functionality of the visitor pattern is the entry point for clients to initiate the traversal of a structure with a visitor. This entry point is the accept method. Clients can be parties from outside of the visitor pattern or the visitor itself when initiating the traversal of a substructure within a visit method. Therefore the most important property of a visitable class for type V is that it declares an accept method for type V. Moreover since we are looking at a single instance of the pattern and we want to make sure that there is a single entry point for starting the traversal with a visitor of type V, a visitable class for type V has to provide a single accept method for type V (Rule 2).

4.2.2 Double Dispatch Mechanism

An accept method implements the dispatch mechanism that calls the appropriate visit method for its target object. The question is therefore how do we know which visit method is appropriate for a specific accept method.

The purpose of the different visit methods is that a visitor can process visitable objects of different types differently. The accept method therefore has to decide based on the type of its target object which visit method it calls. To be able to make this decision uniquely, we have to remove ambiguity from the visitor by requiring that the visitor only declares one visit method for the exact same declared type of the visit subject (Rule 8).

Moreover we know that if there exists a visit method in the visitor for a type which is equivalent to the type of the target object of the accept method, the accept method has to call that visit method. To force this behavior, we define that for all applicable visitor and visitable objects the method contracts of an accept method have to be equivalent to the method contracts of the visit method for the type which is equivalent to the type of the target object (Rule 5).

But what happens if there exist no such visit method. In that case we have two distinguish two situations.

First, if the base class of our visitable class is also a visitable class for the same visitor, we can treat the target objects in the same way as objects of the base class. Since we have defined before that each visitable class has a single accept method (Rule 2), this implies that the accept method has to override the accept method from its base class. Therefore the accept method has to refine the method contracts from the overridden method, meaning the preconditions of the overridden method may only be weakened and the postconditions may only be strengthened. As a consequence the overriding accept method also inherits the specifications from the visit method corresponding to the overridden accept method.

Secondly, if the base class is not a visitable class, there is no visit method that we can call from the accept method. If our visitable class is an abstract class this means that we have to declare the accept method as abstract method. If the visitable class is a concrete class, we have a problem, since this means that some objects of a visitable type cannot be visited. To prevent this, we require that for each concrete visitable class the visitor declares a visit method which visit subject is applicable for objects of the visitable class (Rule 7).

4.2.3 Visit Methods

Since we have defined that the method contracts of the accept methods represent the method contracts of the corresponding visit method (a consequence from Rule 5), we have to make sure that this requirement does not break the behavioral subtyping rules for method refinement for the accept methods. To achieve that we translate the behavioral subtyping rules from the accept methods to the visit methods. This means that a visit method with a visit subject of type S has to refine all visit methods with a visit subject of a base type of S, since a visit method for type S may correspond to an accept method in S or subclass of S and a visit method for a base type of S may correspond to an accept method in a base class of S (Rule 12).

4.2.4 Defines Clauses and Speconly Methods

Defines clauses are used to represent mathematical functions with respect to a single parameter which type is a visitable type. A single mathematical function is represented by one or more defines clauses. Each clause defines the function for a different visitable type.

If we go back to our abstract syntax tree example, we have defined the mathematical function to represent the evaluation value of an expression as follows.

$$eval(e) = \begin{cases} eval(e.Left) + eval(e.Right) & \text{if e is an addition} \\ eval(e.Left) - eval(e.Right) & \text{if e is a subtraction} \\ e.value & \text{if e is a literal} \end{cases}$$

This function can be represented by the following defines clauses.

```
defines abstract int eval(Expression e);
defines override int eval(Add e): eval(e.Left) + eval(e.Right);
defines override int eval(Sub e): eval(e.Left) - eval(e.Right);
defines override int eval(Literal e): e.Value;
```

Listing 4.3: $eval(e)$ as defines clauses

The first defines clause, the abstract clause, specifies that for all objects of type *Expression* there exists a definition of the function *eval*. The requirement for abstract clauses is that the corresponding visitable class, in our case *Expression*, is an abstract class. The overriding clauses define the function for the different concrete subclasses of *Expression*. Of course the abstract defines clause has to be overridden for all concrete subclasses, which in our case are *Add*, *Sub*, and *Literal*, otherwise the function is not defined for all objects of type *Expression*.

Moreover, the defines clause is part of the visit method specification which visit subject type corresponds to the parameter type of the defines clause. But since we do not implement visit methods for all visitable types for which we want to add defines clauses, as in our case we have no visit method for *Expression*, we allow programmers to write visit methods which only provide specifications. Such methods are called speconly methods and are characterized by the modifier *speconly*.

With speconly methods and the above defines clauses we can write the specifications of our class *EvalVisitor* as follows.

```
[Visitor(Expression)] class EvalVisitor {

    private int total;
    public int Total { get { return total; } }

    [Visit(Expression)]
    speconly void Visit([VisitSubject] Expression e);
        defines abstract int eval(Expression e);
        ensures Total == eval(e);

    [Visit(Add)]
    public void VisitAdd([VisitSubject] Add e)
        defines override int eval(Add e): eval(e.Left) + eval(e.Right);
        ensures Total == eval(e);
    { ... }

    [Visit(Sub)]
    public void VisitSubtract([VisitSubject] Sub e)
        defines override int eval(Sub e): eval(e.Left) - eval(e.Right);
        ensures Total == eval(e);
    { ... }
```

```
    [Visit(Literal)]
    public void VisitLiteral([VisitSubject] Literal e)
        defines override int eval(Literal e): e.Value;
        ensures Total == eval(e);
    { ... }

}
```

Listing 4.4: Specifying class $EvalVisitor$

As we can see speconly methods not only have the purpose to specify defines clauses, but can also be used to write method contracts. Method contracts might seem unnecessary for speconly methods in the first place, but since we have defined refinement rules for the method contracts of visit methods, which also include speconly methods, this will force us to refine the contracts in other visit methods.

To be able to verify the postconditions in the above example, the verifier has to be provided with a representation of the defines clauses. Thus a function defined in defines clause is translated to a Boogie function. To be able to reason about such a Boogie function, each definition from a defines clause is translated to an axiom that defines the substitution that the verifier may perform if the types of the parameter of the defines clause and the argument to the Boogie function match.

For example to prove the postcondition $Total == eval(e)$ in $VisitAdd$, the verifier might perform a substitution with the definition of $eval$ for type Add to get $Total == eval(e.Left) + eval(e.Right)$.

This representation brings us to a next requirement for defines clauses.

A defines clause may only override an abstract clause. If we would allow arbitrary overrides, the function would not be defined uniquely for certain types of objects. This is a problem since we are reasoning modularly and could confuse the different definitions.

For example if we would extend our visitor with a visit method for type $BinaryExpression$ that defines the function $eval$ as follows.

```
[Visit(BinaryExpression)]
public void VisitBinaryExpression([VisitSubject] BinaryExpression e)
    defines override int eval(BinaryExpression e): 0;
    ensures Total == eval(e);
{
    total = 0;
}
```

Listing 4.5: Visit method for $BinaryExpression$

We could falsely deduce from the postcondition of a call to $VisitBinaryExpression$ with an argument object of type Add that $Total == eval(e.Left) + eval(e.Right)$, since the postcondition states that $Total == eval(e)$ and we know e is of type Add.

4.3 Visitor Subclassing

If we create a subclass of the visitor class, the subclass inherits the capability to visit the same visitable classes as its base class, since its objects can be passed as accept subject to the accept method of the same structural objects as well (Rule 10). As such the double dispatch mechanism is designed for its base class and therefore also the set of visit methods that are called by the

accept methods. Thus the only way to extend the behavior is to override visit methods from the base class. We enforce this behavior by disallowing the declaration of new visit methods in visitor subclasses (Rule 9).

Previously we have required that the method contracts of the accept method are equivalent to the method contracts of the corresponding visit method for all visitor and visitable objects (Rule 5). Another reason why this rule is important is not only to ensure the correct visit method is called from the accept method, but also to ensure that we provide the specifications of the dispatched visit method to the clients of the accept method. Therefore we have to retain this rule for visitor subclasses.

Again to make sure that the specifications of the visit methods do not break the behavioral subtyping rules, we have to translate the behavioral subtyping rules to the visit methods of the visitor subclass with respect to the visit subject type. As a result we define the behavioral subtyping rules for the visit methods in two dimensions, namely the visitor types and the visitable types (Rule 12).

4.4 Admissibility Rules

After we have described less formally what an admissible visitor pattern is, we provide a formal definition in the form of admissibility rules. As such we split the rules into rules for visitable classes, accept methods, visitor classes, visit methods, and finally into rules for the new constructs, which are defines clauses and speconly methods.

4.4.1 Visitable Class

A visitable class S for type V

```
[Visitable(V)] class S { ... }
```

is admissible if

The visitor class for which S implements the dispatch mechanism can visit objects of type S.

Rule 1. There exists a class S_{base}, where $S <: S_{base}$, such that V is an admissible visitor class (4.4.3) for type S_{base}.

S provides a single entry point for visitor objects of type V.

Rule 2. S declares, overrides, or inherits exactly one method, which is an admissible accept method (4.4.2) for type V.

Subclasses of S may also be visited by visitors of type V.

Rule 3. All classes derived from S are admissible visitable classes for type V.

4.4.2 Accept Method

An accept method *accept* for type V,

```
[Accept(V)] R accept([AcceptSubject] V v, P p)
    requires P_accept(this, v, p);
    ensures Q_accept(this, this', v, v', p, p');
```

where *accept* is declared in class S, is admissible if

An accept method may declare multiple parameters of which exactly one parameter is the visitor to which it dispatches.

Rule 4. Exactly one parameter is declared as accept subject and the declared type of the parameter is V.

The method contracts of the accept method are equivalent to the method contracts of the corresponding visit method in V or classes derived from V for the same applicable visitor, visitable, and argument objects. For an implementation of the methodology this means that the method contracts from the accept methods can be derived from the method contracts of the visit methods.

Rule 5. For all visit methods *visit* for type S, where *visit* is declared in $V_{sub}<:V$, the declared types of the parameters of *accept* and *visit* except for the visit and accept subject are equal, and the return type of *accept* and *visit* are equal. Furthermore the following relationship between the pre- and postconditions of *accept* and *visit* holds

$$
\begin{aligned}
&\forall\ s,\ v,\ p \bullet \\
&\quad typeof(s) <: S\ \wedge\ typeof(v) <: V_{sub}\ \wedge typeof(p) <: P \\
&\quad \Rightarrow (P_{accept}(s,\ v,\ p) \Leftrightarrow P_{visit}(v,\ s,\ p)) \\
&\qquad \wedge\ (Q_{visit}(v,\ v',\ s,\ s',\ p,\ p') \Leftrightarrow Q_{accept}(s,\ s',\ v,\ v',\ p,\ p'))
\end{aligned}
$$

4.4.3 Visitor Class

A visitor class V for type S

```
[Visitor(S)] class V { ... }
```

is admissible if

S provides the double dispatch mechanism for visitors of type V or a base type of V.

Rule 6. There exists a type V_{base}, where $V<:V_{base}$, such that S is an admissible visitable class (4.4.1) for type V_{base}.

V is able to visit all objects of type S.

Rule 7. For all concrete classes $S_{sub}<:S$, V declares, overrides or inherits an admissible visit method *visit* (4.4.4) for type S_{sub} or a base type of S_{sub}, where *visit* is not a speconly method.

V contains no duplicate visit methods.

Rule 8. For all classes $S_{sub}<:S$, V declares, overrides or inherits at most one visit method for type S_{sub}.

V is a subclass of a visitor class implies that it does not declare new visit methods.

Rule 9. The base class of V is not a visitor for type S, or all visit methods for type $S_{sub}<:S$ override a visit method for type S_{sub}.

Subclasses of V may also visit objects of type S.

Rule 10. All classes derived from V are admissible visitor classes for type S.

4.4.4 Visit Method

A visit method *visit* for type S,

```
[Visit(S)] R visit([VisitSubject] S s, P p)
    requires P_visit(this, s, p);
    ensures Q_visit(this, this', s, s', p, p');
```

where *visit* is declared in class V, is admissible if

A visit method may declare multiple parameters, of which exactly one parameter is the visitable object that is visited.

Rule 11. Exactly one parameter is declared as visit subject and the declared type of the parameter is S.

The method contracts of *visit* have to refine the method contracts of all visit methods $visit'$ which visit subject type is a base type of S. For an implementation of the methodology we can enforce this behavior by the language semantics similar to the behavioral subtyping rules, in that the actual postcondition of *visit* is defined as the conjunction from its own postcondition and the postconditions from the methods $visit'$. Likewise the preconditions can be combined by disjunction.

Rule 12. For all visit methods $visit'$ of type S_{base} declared in class V_{base}, where $V <: V_{base}$ and $S <: S_{base}$, the declared types of the parameters of *visit* and $visit'$ except for the visit subject are equal, and the return type of *visit* and $visit'$ is equal. Furthermore the following relationship between the pre- and postconditions of *visit* and $visit'$ holds

$$
\begin{aligned}
&\forall\ v,\ s,\ p \bullet \\
&\quad typeof(v) <: V\ \wedge\ typeof(s) <: S\ \wedge\ typeof(p) <: P \\
&\quad \Rightarrow (P_{visit}(v,\ s,\ p) \Rightarrow P_{visit'}(v,\ s,\ p)) \\
&\qquad \wedge\ (Q_{visit'}(v,\ v',\ s,\ s',\ p,\ p') \Rightarrow Q_{visit}(v,\ v'\ s,\ s',\ p,\ p'))
\end{aligned}
$$

The visitor is not peer with the target object. This constraint has to be implied by the precondition of the visit method, and therefore by Rule 5 is also implied by the preconditions of the accept methods.

Rule 13. The precondition of *visit* satisfies

$$
\begin{aligned}
&\forall\ v,\ s,\ p \bullet \\
&\quad typeof(v) <: V\ \wedge\ typeof(s) <: S\ \wedge\ typeof(p) <: P \\
&\quad \Rightarrow (P_{visit}(v,\ s,\ p)\ \Rightarrow\ v.owner\ \neq s.owner)
\end{aligned}
$$

4.4.5 Speconly Methods

Speconly methods provide a way to write visit methods which do not provide an implementation. Their main purpose is to be able to specify defines clauses for visitable types for which we do not want to provide any implementation. As such speconly methods should not be called.

Rule 14. Speconly methods must not be called.

We introduce concrete speconly methods that define a speconly method initially, and overriding speconly methods to extend the specifications of a speconly method from a base visitor class. The meaning is equivalent to virtual methods and overriding methods with the difference that speconly methods have no implementation.

Concrete Speconly Method

A method m declared in class C with the following signature

```
speconly R m(P p);
    requires P_m(this, p);
    ensures Q_m(this, this', p, p');
```

is admissible if

Rule 15. m does not hide another member declared in class C or a base class of C

Rule 16. m's method contracts are admissible.

Overriding Speconly Method

A method m declared in class C with the following signature

```
speconly override R m(P p);
    requires P_m(this, p);
    ensures Q_m(this, this', p, p');
```

is admissible if

Rule 17. The base class of C declares or inherits a speconly method with the same name, same parameter types, and same return type.

Rule 18. m's method contracts are admissible and refine the method contracts of the overridden member.

4.4.6 Defines Clause

As we have already indicated there are different types of defines clauses. The types are abstract, concrete, and overriding defines clauses. Defines clauses with the same name represent different definitions of the same function for the declared parameter types. Since not all combinations of defines clauses with the same name form an admissible function, we define rules that have to be satisfied by an admissible defines clause.

Abstract Clause

An abstract defines clause is used to introduce a new function. As such it indicates that for all objects of its parameter type the function has a definition. Since it does not provide a definition itself, its parameter type has to be an abstract class, and the visit method it is specified for is a speconly method.

An abstract defines clause

```
defines abstract R f(S s);
```

defined in method *visit* in class V is admissible if

Rule 19. *visit* is an admissible speconly method and an admissible visit method for objects of type S.

Rule 20. S is an abstract class.

The next two rules specify that there is no second defines clause that introduces a function with the same name.

Rule 21. *visit* defines exactly one clause with the same name.

Rule 22. In class V and in all transitive base classes of V, there exists no visit method $visit'$, which is not *visit* and which declares an abstract or concrete defines clause with the same name.

All concrete subclasses of the parameter type of the abstract defines clause define the function.

Rule 23. For all concrete classes $S_{sub}<:S$ there exists a visit method $visit'$ in class V which declares an overriding defines clause with the same name and with parameter type S_{sub} or a base type of S_{sub}.

Concrete Clause

Similar to an abstract defines clause, a concrete defines clause introduces a new function. With the difference that a concrete defines clause also defines the function, and therefore cannot be redefined to avoid ambiguity as we have already explained before.

A concrete defines clause

```
defines R f(S s): D(s)
```

defined in method *visit* in class V is admissible if

Rule 24. *visit* is an admissible visit method for objects of type S.

The next two rules specify that there is no second defines clause that introduces a function with the same name.

Rule 25. *visit* defines exactly one clause with the same name.

Rule 26. In class V and in all transitive base classes of V, there exists no visit method $visit'$, which is not *visit* and which declares an abstract or concrete defines clause with the same name.

Rule 27. $D(s)$ is an admissible Spec# expression of type R, each access expression starts with s and all method calls are pure.

Overriding Clause

Overriding defines clauses are used to define a function introduced by an abstract defines clause for subtypes of the parameter type of the abstract defines clause. To avoid ambiguity, an overriding defines clause is only allowed to declare a function for its parameter type if the function is not already defined for its parameter type or a base type of its parameter type.

An overriding defines clause

```
defines override R f(S s): D(s)
```

defined in method *visit* in class V is admissible if

Rule 28. *visit* is an admissible visit method for objects of type S.

Rule 29. *visit* defines exactly one clause with the same name.

There exists a corresponding abstract defines clause which parameter type is a base type of S.

Rule 30. In class V there exists a visit method $visit'$ for type S_{base}, where $S{<:}S_{base}$ and $S_{base}{\neq}S$, which declares an abstract defines clause with the same name and return type R.

There exists no corresponding concrete or overriding defines clause which parameter type is a base type of S.

Rule 31. All defines clauses with the same name declared in a visit method in class V for type S_{base}, where $S{<:}S_{base}$ and $S_{base}{\neq}S$, are abstract.

Rule 32. $D(s)$ is an admissible Spec# expression of type R, each access expression starts with s and all method calls are pure.

Function Reference

A function that is defined by defines clauses may be used in specifications of visit methods, we call such a use a function reference. A function reference has to pass the type checks for its argument and function value.

A reference to a function that is declared in a defines clause

```
f(E)
```

in method *visit*, which is declared in class V, is admissible if

Rule 33. v is an admissible visit method and V is an admissible visitor class.

Rule 34. The reference is part of an expression within a postcondition, a defines clause, an assertion, an assumption, or a loop invariant of a visit method.

Rule 35. A visit method in V declares a function, such that the type of E is a subtype of the parameter type of the function, and the expected type of the context of the reference equals the type of the function.

4.5 Translation Function

With the introduction of new language constructs we have to modify the translation function of Spec# to BoogiePL programs to incorporate the new constructs. As such we have to define the translation of visit methods, speconly methods and defines clauses.

4.5.1 Visit Method

Additionally to normal methods, we also have to translate the defines clauses of a visit method. The translation of a visit method is defined as

```
Tr⟦ [Visit(S)] R Visit(S s)
        requires P;
        ensures Q;
        defines D;
```

```
        modifies M;
    { B } ⟧ =

    Tr⟦ defines D ⟧

    Tr⟦ R Visit(S s)
            requires P;
            ensures Q;
            modifies M;
        { B } ⟧
```

Listing 4.6: Translation function for visit methods

4.5.2 Speconly Method

Since a speconly method cannot be called, the defines clause is the only construct that has to be translated for speconly methods. The method contracts do not have to be translated since they are already encoded in the method contracts from other visit methods by the admissibility rules. The translation of a speconly method is defined as

```
Tr⟦ [Visit(S)] speconly R Visit(S s)
        requires P;
        ensures Q;
        defines D; ⟧ =

    Tr⟦ defines D ⟧
```

Listing 4.7: Translation function for speconly methods

4.5.3 Defines Clause

Additionally to the different clauses introduced for the admissibility rules we also came up with an extended version of the concrete and the overriding defines clause. We introduce these clauses here since the admissibility rules are almost exactly the same as for the standard clauses. The important differences are part of the translation.

Essentially the difference to the standard clauses is that the extended clauses allow to split function declarations into different cases distinguished by conditions. For example for a visitable class T with field $value$ of type integer, the function $value(t)$ declared as

```
defines int value(T t):
    when t.value >= 0: t.value;
    when t.value < 0: -t.value;
    default: 0;
```

Listing 4.8: Extended defines clause

is defined as $t.value$ if $t.value >= 0$ is true, otherwise if $t.value < 0$ is true the function is defined as $-t.value$, and finally for all other cases the function is defined as 0.

For the translation of defines clauses, we assume that there exists a translation function $Tr[E, h]$, which translates an expression E with respect to the heap as local variable h.

The Boogie function $\$IsNotNull(o, T)$ is the translation of the Spec# expression o *is* T.

Abstract Clause

An abstract defines clause introduces a new function. Since it does not provide any definition, the translation of an abstract defines clause is only a Boogie function. The parameter type specified in the abstract defines clause was only important for the admissibility rules and is lost during the translation.

The translation of an abstract defines clause is defined as

$Tr[\![$ `defines abstract R name(T t);` $]\!] =$

```
function name(t: ref, h: Heap) returns (R);
```

Listing 4.9: Translation function for abstract defines clauses

Concrete Clause

Additionally to the abstract defines clause the concrete defines clause also contains a definition that we have to translate. As we translate the concrete defines clause equivalently to the abstract defines clause to a Boogie function, we can translate the definition in the defines clause to an axiom that describes the Boogie function.

The translation of a concrete defines clause is defined as

$Tr[\![$ `defines R name(T t) : C(t);` $]\!] =$

```
function name(t: ref, h: Heap) returns (R);
axiom (forall h: Heap, o: ref ::
       $IsNotNull(t, T) ==> name(o, h) == Tr[[C(o), h]]);
```

Listing 4.10: Translation function for concrete defines clauses

Extended Concrete Clause

The translation of an extended concrete defines clause is defined as

$Tr[\![$ `defines R name(T t):`
`when` C_1`(t): `D_1`(t);`
`...`
`when` C_n`(t): `D_n`(t);`
`default: `$D_{default}$`(t);` $]\!] =$

```
function name(t: ref, h: Heap) returns (R);
axiom (forall h: Heap, o: ref ::
        $IsNotNull(o, T)
        ==> (  (Tr[[C1(o), h]] ==> name(o, h) == Tr[[D1(o), h]])
               && ...
               && (!Tr[[C1(o), h]] && ... && !Tr[[Cn−1(o), h]] && Tr[[Cn(o), h]]
                   ==> name(o, h) == Tr[[Dn(o), h]])
               && (!Tr[[C1(o), h]] && ... && !Tr[[Cn(o), h]]
```

```
                    ==> name(o, h) == Tr⟦D_default(o), h⟧)  );
```

Listing 4.11: Translation function for extended concrete defines clauses

Overriding Clause

Overriding defines clauses define the function that is introduced in an abstract defines clause. Since the translation of the abstract defines clause already translates to a Boogie function, we only have to translate the definition in the defines clause to an axiom that describes the Boogie function.

The translation of an overriding defines clause is defined as

```
Tr⟦ defines override R name(T t) : C(t); ⟧ =

    axiom (forall h: Heap, o: ref :: $IsNotNull(o, T)
           ==> name(o, h) == Tr⟦C(o), h⟧);
```

Listing 4.12: Translation function for overriding defines clauses

Extended Overriding Clause

The translation of an extended overriding defines clause is defined as

```
Tr⟦ defines override R name(T t):
        when C_1(t): D_1(t);
        ...
        when C_n(t): D_n(t);
        default: D_default(t); ⟧ =

     axiom (forall h: Heap, o: ref ::
            $IsNotNull(o, T)
            ==> (  (Tr⟦C_1(o), h⟧ ==> name(o, h) == Tr⟦D_1(o), h⟧)
                   && ...
                   && (!Tr⟦C_1(o), h⟧ && ... && !Tr⟦C_{n-1}(o), h⟧ && Tr⟦C_n(o), h⟧
                       ==> name(o, h) == Tr⟦D_n(o), h⟧)
                   && (!Tr⟦C_1(o), h⟧ && ... && !Tr⟦C_n(o), h⟧
                       ==> name(o, h) == Tr⟦D_default(o), h⟧)  ));
```

Listing 4.13: Translation function for extended overriding defines clauses

Function Reference

As we have translated defines clauses to Boogie functions, function references in the specification of visit methods have to be translated to references of these Boogie functions.

The translation of a reference to a function that is declared in a defines clause is defined as

```
Tr⟦ name(E) ⟧ =

    name(Tr⟦E⟧)
```

Listing 4.14: Translation function for function references

4.6 Conclusion

In this chapter we have defined a methodology for the visitor pattern. We have specified how the different method contracts of the visit methods and accept methods are related. We have introduced the defines clause that can be used for recursive specifications of the visit methods, and we have introduced speconly methods to write visit methods which only provide specifications in the form of defines clauses and method contracts.

In the next chapter we will use the methodology to extend the Spec# system such that we can improve the specification and verification of Spec# programs that implement the visitor pattern.

Chapter 5

Visitor Pattern: Implementation for Spec#

Our implementation of the methodology for the visitor pattern is based on the Spec# programming system. As such it is an extension of Spec# and only affects the verification, if the new language constructs and pattern specific attributes are used.

To explain the details of our implementation, we will first explain how the pattern has to be annotated to use the described methodology. Then we will describe the syntax of the newly created language constructs and finally show how we implemented the methodology. As part of the implementation of the methodology, we will describe how the different admissibility rules and the translation function are implemented.

Finally we explain how runtime checks for the visitor pattern, and especially for the introduced defines clauses work.

5.1 Annotations

In the methodology we have introduced several attributes to annotate the different methods, classes, and parameters of the visitor pattern. Since these attributes contain redundancy, we decided to restrict our implementation to a small set of attributes from which all methods, classes, and parameters that are important for the pattern can be deduced.

First of all we drop the $Visitable(Type\ t)$ and $Visitor(Type\ t)$ attributes, since this information is implicitly encoded in the context where the $Visit(Type\ t)$ and $Accept(Type\ t)$ attributes are used.

Secondly we remove the $VisitSubject$ and $AcceptSubject$ attributes. We expect that the corresponding parameters are the first parameter of the visit and accept methods respectively. Furthermore we have learned from studying implementations of the pattern that programmers rarely write more than one parameter for the visit and accept methods. The reason is that since the declared types of the parameters have to be the same for all visitors of a pattern instance, they have to be the least common denominator of all visitors. Hence the resulting loss of type information is usually avoided and the values are passed between the different visit rounds in fields instead of arguments.

Moreover since the type argument for the $Visit(Type\ t)$ and $Accept(Type\ t)$ attribute can be

deduced from the visit and accept subjects, we remove the parameters from these attributes. Additionally the attributes do not have to be specified for overriding methods if the overridden method already specifies the attribute.

To be able to use the remaining two attributes in Spec#, we have to define corresponding attribute classes that are derived from *System.Attribute* and which specify where and how the attributes can be applied and what functionality they provide. For convenience we added the two attribute classes to the Spec# library as part of the namespace Microsoft.Contracts. The two classes are defined as follows.

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false,
                Inherited = true)]
public sealed class AcceptAttribute : Attribute
{
    public AcceptAttribute() {}
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple = false,
                Inherited = true)]
public sealed class VisitAttribute : Attribute
{
    public VisitAttribute() {}
}
```

Listing 5.1: Spec# attributes for the visitor pattern

With our extended version of the Spec# system the visit and accept methods in a Spec# program can be annotated with [*Visit*] or [*Accept*] to use our implementation of the methodology.

5.2 Language Extensions

As already shown in the definition of the methodology, our implementation adds two new language constructs to Spec#. The first language construct is a new method modifier, speconly and the second is the defines clause. Additionally to the semantics of these constructs which have already been explained, here is a syntactical definition.

The grammar is based on the notation and specification of the ECMA standard for C#[9]. Italic symbols refer to non-terminals and typewriter symbols to terminals. Since there is no formal specification of the Spec# language and grammar, the *method-header* production, which we had to modify, is assumed to be defined for Spec# as

method-header:

attributes $_{opt}$ *method-modifiers*$_{opt}$ *return-type member-name*
type-parameter-list$_{opt}$ (*formal-parameter-list*$_{opt}$)
type-parameter-constraints-clauses$_{opt}$ *method-contracts*$_{opt}$

method-contracts:

method-contract
method-contracts method-contract

method-contract:

method-requires
method-ensures

method-modifies

Listing 5.2: Grammar for method signatures in Spec#

The definition of the productions for *method-requires*, *method-ensures*, *method-modifies* and other Spec# specific productions is not listed here, since it is not in the scope of our language changes.

The grammar is modified as follows. First, the *method-modifier* production is extended with the *speconly* keyword. Secondly, the *method-contract* production is extended with the *method-defines* production to incorporate the defines clause.

method−contract:

method-requires
method-ensures
method-modifies
method-defines

method-defines:

`defines` *defines-modifier*$_{opt}$ *defines-type identifier* `(` *type identifier* `) :`
defines-block

defines-type

return-type

defines-modifier:

`abstract`
`override`

defines-block:

defines-section
defines-when-sections$_{opt}$ *defines-default-section*

defines-section:

defines-when-definition

defines-when-sections:

defines-when-section
defines-when-sections defines-when-section

defines-when-section:

`when` *defines-when-condition* `:` *defines-when-definition* `;`

defines-default-section

`default :` *defines-when-definition* `;`

defines-when-condition:

expression

defines-when-definition:

expression

Listing 5.3: Grammar extensions for defines clauses

Although the above grammar accepts arbitrary types for the defines clauses, our implementation currently only works with non-object types.

5.3 Methodology

The entry point which has to be provided to use the implementation of our methodology is the *Accept* attribute. As soon as the compiler detects a method with an *Accept* attribute, it starts to process the pattern and its participating classes and methods. From the declaration class of the accept method it deduces the visitable types, from the accept subject type the visitor types and its visit methods. After the participating classes and methods are known to the compiler, it makes sure the program satisfies the admissibility rules. Certain rules have to be checked and others can be enforced.

The admissibility rules for the visitable and visitor classes can all be implemented as compiler checks. This is also the case for the rules for speconly methods, defines clauses, and for certain rules for visit and accept methods. We will therefore focus on the rules that we implemented differently.

5.3.1 Visit Method

The rules of interest for visit methods are Rule 12 and Rule 13. Both specify constraints for the method contracts.

Rule 13 specifies that the visitor and the visit subject must not be peers. Therefore we add the following precondition to all visit methods, where *this* refers to the visitor object and *s* refers to the visit subject.

```
requires Owner.Different(this, s);
```

Listing 5.4: Precondition for visit methods

To satisfy Rule 12 we rewrite the method contracts of the visit methods such that the rule is satisfied within the compiler. Similarly to the way the behavioral subtyping rules for method contracts are applied, where the postconditions from overridden methods are copied to its overriding methods, we copy the postconditions between visit methods. The postconditions are copied in the subtype direction of the visit subject types.

The resulting transformation of the method contracts looks as follows. Given a visit method declared in visitor class V for type S with the following signature

```
[Visit] public R Visit(S s)
    ensures Q_Visit(this, s);
    modifies M_Visit(this, s);
```

Listing 5.5: Visit method for type S

and visit methods $Visit_i$ declared in visitor class V for type S_i, where $S <: S_i$, with the following signature

```
[Visit] public R Visit_i(S_i s)
    ensures Q_Visit_i(this, s);
    modifies M_Visit_i(this, s);
```

Listing 5.6: Visit method for type S_i

The method signature of $Visit$ is rewritten to

```
[Visit] public R Visit(S s)
    ensures Q(this, s);
    // forall Visit_i {
    ensures Q_Visit(this, s);
    // }
    modifies M_Visit(this, s);
```

Listing 5.7: Rewritten visit method

The transformation is applied before applying the behavioral subtyping rules for method contracts. The method signatures of the methods $Visit_i$ refer to the original, untransformed definitions.

5.3.2 Accept Method

The rule of interest for accept methods is Rule 5. It describes the relation of the method contracts of accept methods to the method contracts of visit methods. Essentially it states that the method contracts of an accept method are equivalent to the corresponding visit method for any given visitor object.

Since in Spec# preconditions can only be written for non-overriding methods, we may only add preconditions to the initial declaration of the accept method and thus the precondition of all accept methods have to be the same. Translated to the visit methods this also means that for all visit methods the preconditions have to be the same. As such we decided to make sure the rule is satisfied when excluding the user declared preconditions, but to allow the programmer to violate the rule, since a violation is captured by verification when proving the precondition for a call to a visit or accept method.

To satisfy the rule without user declared preconditions, we only have to add an equivalent precondition of the precondition that we add to the visit methods, namely that the accept subject and the visitable object must not be peers. Therefore we add the following precondition to the accept methods, where *this* is the visitable object and v the accept subject.

```
requires Owner.Different(this, v);
```

Listing 5.8: Precondition for accept methods

To satisfy the rule for the postconditions, we first have to think of how programmers write specifications. Since the logic of a visitor is mostly implemented in the visit methods, it makes more sense to be able to write postconditions for the visit methods. Because of the tight coupling to the postconditions of the accept methods described in the admissibility rule, we can then derive the postconditions for the accept methods from the postconditions of the visit methods.

The resulting transformation of the method contracts looks as follows. Given an accept method declared in visitable class S with the following signature

```
[Accept] public R Accept(V v)
```

```
    modifies M_Accept(this, v);
```

Listing 5.9: Accept method in class S

and all visit methods $Visit_i$ declared in visitor class V_i<:V with the following signature

```
[Visit] public R Visit_i(S s)
    ensures Q_Visit_i(this, s);
    modifies M_Visit_i(this, s);
```

Listing 5.10: Visit method for type S

The method signature of Accept is rewritten to

```
[Accept] public R Accept(V v);
    // forall Visit_i {
    ensures v is V_i ==> Q_Visit_i((V_i)v, this);
    // }
    modifies M_Accept_i(this, v);
```

Listing 5.11: Rewritten accept method

The transformation is applied before applying the behavioral subtyping rules for method contracts. The visit methods $Visit_i$ refer to the original methods without the described transformation for visit methods.

The problem with the described rewrites is that Boogie cannot prove the new postcondition of the accept method for some cases. Because the method call to a visit method within the accept method is dynamically bound, Boogie only uses the method signature from the visit method of the class corresponding to the declared type of the target object for its proofs, and therefore is not able to prove its postcondition, which reflects the postconditions of all overrides of the visit method.

Let us look at this in more detail. A visit method declared in V, where the base type of V is not a visitor has the following method signature

```
public virtual R Visit(S s)
    ensures Q_V.Visit(this, s);
```

Listing 5.12: Visit method in class V

Overriding visit methods of the above visit method in classes V_i, where V_i<:V, have the following signature.

```
public override R Visit(S s)
    ensures Q_V_i.Visit(this, s);
```

Listing 5.13: Visit method in class V_i

If we write an accept method in class S that calls the above visit method, the previously described transformation of the method contracts will cause the method contracts of both $V.Visit$ and $V_i.Visit$ to be copied to the accept method. The resulting accept method looks as follows.

```
public R Accept(V v)
    ensures Q_Visit(v, this);
    ensures v is V_i ==> Q_V_i.Visit(v, this);
```

```
{
    v.Visit(this);
}
```

Listing 5.14: Rewritten accept method

But for modularity reasons, the translation of the visit method call to BoogiePL refers to the translated procedure of $V.Visit$, since the declared type of v is V. The corresponding procedure looks as follows.

```
procedure Visit$Virtual$(this: ref, s: ref)
    ...
    ensures Tr⟦ Q_{V.Visit}(this, s) ⟧;
```

Listing 5.15: Procedure for visit method in class V

The problem is that this procedure does not include the postconditions of the overriding methods $V.Visit_i$, and therefore we cannot prove the postcondition of the accept method.

To resolve this, we add the postconditions of the overriding methods $V.Visit_i$ also to the procedure of the overridden method $V.Visit$. But since we do not want to prove these postconditions for the implementation of $V.Visit$, we precede these postconditions with the keyword $free$, which means they do not have to be proved in the implementation. Thus the procedure is transformed as follows.

```
procedure Visit$Virtual$(this: ref, s: ref)
    ...
    ensures Tr⟦ Q_{V.Visit}(this, s) ⟧;
    // forall V_i <: V {
    free ensures $IsNotNull(this, V_i) ==> Tr⟦ Q_{V_i.Visit} ⟧;
    // }
```

Listing 5.16: Transformed procedure for visit method in class V

As we can see we still have to break modularity for the verification. But since the non-modular specifications are generated automatically, the user-defined specifications are modular.

5.4 Runtime Checks

Since our implementation rewrites the input program before the runtime checks are generated, the rewriting and copying of postconditions is already reflected by the generated runtime checks. The only exception where we have to generate additional runtime checks are defines clauses.

To create runtime checks for defines clauses, we have to calculate the function wherever it is referenced. Since each defines clause defines the function for a specific visitable type, we have to choose the definition of the function based on the type of its argument to calculate the value. If we do not want to inspect the type at runtime explicitly, we can transform each defines clause to a method of the class corresponding to its parameter type, relate the different methods by method overriding, and then at runtime dynamic binding implicitly ensures that the correct implementation is chosen.

If we compare this to our first attempt to verify the visitor pattern in Spec# from chapter 3, this representation is very similar. In fact if we use pure methods to represent the defines clauses and also transform the definitions of the clauses to postconditions, we have almost the same

representation. This relationship is a motivation to translate the defines clause entirely to pure methods, which are then not only used for runtime checks but also for the static verification. In our implementation this is exactly what we did. In the rest of this section we therefore show the translation of the different types of defines clauses to pure methods and compare the resulting translation of the pure methods to the translation function presented in our methodology. As a consequence of this translation we require the parameter types of the defines clauses to be non-null types.

But before we start describing the translation, we want to explain why defines clauses are still an improvement over writing pure methods directly in Spec#. The problem with representing the specifications of the visitor as pure methods of the structural classes is that it interferes with the concept of decoupling the operation from the structure. The defines clauses, which in our extended language represent the specifications, are decoupled from the structure. Only for the verification and the runtime checks, which are not directly visible to the programmer, we break with the concept. Therefore defines clauses really improve the specification language.

5.4.1 Abstract Clause

An abstract defines clause declared in method *visit* in class V with parameter type S is translated to an abstract pure method in class S.

$Tr_{runtime}[\![$ `defines abstract name(S s);` $]\!] =$

```
[Pure] public abstract R name();
```

Listing 5.17: Translation function for abstract defines clauses

The translation of the abstract pure method to BoogiePL results in a Boogie function with the same signature as the Boogie function described in the translation function for abstract defines clauses from the methodology.

5.4.2 Concrete Clause

A concrete defines clause with parameter type S declared in method *visit* in class V is translated to a virtual pure method in class S

$Tr_{runtime}[\![$ `defines R name(S s) : D(s);` $]\!] =$

```
[Pure] public virtual R name()
    ensures result == D(this);
{
    Tr_runtime[[ return D(this) ]]
}
```

Listing 5.18: Translation function for concrete defines clauses

In comparison to the translation function for concrete defines clauses from the methodology, a virtual pure method is translated to a similar function and axiom. Here is a simplified version of the function and axiom.

```
function #S.name(h: Heap, this: ref) returns (R)

axiom (forall $Heap: Heap, this: ref ::
```

```
    $IsNotNull(this, S) && ... // this is peer valid in $Heap
    ==> #S.name($Heap, this) == Tr[D(this), $Heap]);
```

Listing 5.19: Translation of the pure method to BoogiePL

The difference to the translation function for concrete clauses is that the axiom only applies if *this* is peer valid, which is the default precondition for pure methods. But since the function declared in the defines clause is intended for the use in method contracts, where the default contracts guarantee peer consistency of the participating objects, both translations are equivalent in practice.

5.4.3 Extended Concrete Clause

An extended concrete defines clause with parameter type S declared in method *visit* in class V is translated to a virtual pure method in class S.

$Tr_{runtime}[\![$ `defines R name(S s):`
`when` C_1`(s): D`$_1$`(s);`
`...`
`when` C_n`(s): D`$_n$`(s);`
`default: D`$_{default}$`(s);` $]\!] =$

```
[Pure] public virtual R name()
    ensures (C1(this) ==> result == D1(this))
            && (!C1(this) && ... && !Cn−1(this) && Cn(this)
                ==> result == Dn(this))
            && (!C1(this) && ... && !Cn(this)
                ==> result == Ddefault(this));
{
    Tr_runtime[[
      if (C1(this)) return D1(this);
      ...
      if (Cn(this)) return Dn(this);
      return Ddefault(this);
    ]]
}
```

Listing 5.20: Translation function for extended concrete defines clauses

5.4.4 Overriding Clause

An overriding defines clause with parameter type S_{sub} declared in method *visit* in class V is translated to a pure method in class S_{sub}.

$Tr_{runtime}[\![$ `defines override name(`S_{sub} `s) : D(s);` $]\!] =$

```
[Pure] public override R name()
    ensures D(this);
{
    Tr_runtime[[ return D(this) ]]
}
```

Listing 5.21: Translation function for overriding defines clauses

As a difference to the translation function for overriding defines clauses from the methodology, the translation of an overriding pure method creates a separate Boogie function. But the representation is still equivalent since the translation adds an axiom which relates the Boogie function to the function generated for the overridden method. Of course also an axiom for the definition in the defines clause is added.

```
function #S_sub.name(h: Heap, this: ref) returns (R)

axiom (forall $Heap: Heap, this: ref ::
       this != null && $typeof(this) <: S_sub
       ==> #S.name($Heap, this) == #S_sub.name($Heap, this));

axiom (forall $Heap: Heap, this: ref ::
       $IsNotNull(this, S_sub) && ... // this is peer valid in $Heap
       ==> #S_sub.name($Heap, this) == Tr[D(this), $Heap]);
```

Listing 5.22: Translation of the pure method to BoogiePL

5.4.5 Extended Overriding Clause

An extended overriding defines clause with parameter type S_{sub} declared in method *visit* in class V is translated to a pure method in class S_{sub}

$Tr_{runtime}[\![$ `defines override R name(S_sub s):`
`when C_1(s): D_1(s);`
`...`
`when C_n(s): D_n(s);`
`default: D_default(s);` $]\!] =$

```
[Pure] public override R name()
    ensures (C_1(this) ==> result == D_1(this))
            && ...
            && (!C_1(this) && ... && !C_{n-1}(this) && C_n(this)
                ==> result == D_n(this))
            && (!C_1(this) && ... && !C_n(this)
                ==> result == D_default(this));
{
    Tr_runtime[[
      if (C_1(this)) return D_1(this);
      ...
      if (C_n(this)) return D_n(this);
      return D_default(this);
    ]]
}
```

Listing 5.23: Translation function for extended overriding defines clauses

Chapter 6

Conclusions

We conclude this thesis by looking at related and future work, and finally by recapitulating the results from our work.

6.1 Related Work

Design patterns are often used to illustrate verification techniques in publications. But to our knowledge very little work has been done for the design of specific methodologies for design patterns.

6.1.1 jStar

Distefano and Parkinson[7] introduce jStar, a static verifier which methodology is based on abstract predicates families, a way to define the behavior of a system by user defined predicates, in combination with separation logic. In their paper they demonstrate the methodology applied to different design patterns including the visitor pattern. Although we were inspired by the notion of user defined predicates when designing defines clauses for the visitor pattern, their approach differs in the sense that programmers have to write predicates for the structure and the dispatch mechanism too.

6.1.2 Pattern Enforcing Compiler

PEC[12] stands for Pattern Enforcing Compiler and is an extended Java compiler. It provides additional features to check design patterns during compilation. To be able to use the pattern specific features, the pattern specific classes have to implement special interfaces provided by the library that is delivered with PEC. Design pattern specific checks are implemented as static compiler checks and unit tests. As such its approach is different to ours since it is not sound, and it only checks the general behavior and object structure described by the pattern.

6.1.3 Design Verification Patterns

Knudsen et al.[10] introduce a formalism for design patterns based on communicating sequential processes (CSP), object Z, and duration calculus. CSP is a language to specify patterns of inter-

action in concurrent systems, it is used to formalize the abstract behavior of the pattern. Object Z is an extension of Z introducing object orientated constructs. Duration calculus is used for embedded systems to reason about the timing behavior of a system. For each design pattern they formalize the behavior of the pattern in a verification pattern that gives the proof obligations for a corresponding implementation of a pattern. In contrast to our approach, their approach is mainly used to verify that the pattern is implemented correctly, as opposed to simplify the verification of programs that use the pattern.

6.1.4 Design Pattern Formalizations

Taibi[14] and Mikkonen[13] describe formalizations for design patterns. The most important concept of both approaches is to describe the relationship between objects over time. For our purpose the formalizations were too different from the Spec# methodology to be useful, and neither provides a formalization of the visitor pattern.

6.2 Future Work

Our extended Spec# system already supports the verification of important implementations of the visitor pattern. Future work for the visitor pattern could focus on more general structures, such as cyclic structures, different verification techniques, or other variants of the visitor pattern.

As our solution is specific for the visitor pattern, designing and implementing similar methodologies for other design patterns could simplify the verification of even more programs. Interesting design patterns for the verification are the composite pattern, and most behavioral patterns such as the observer, and iterator pattern. Creational patterns are less interesting for verification since they do not involve complex object collaboration. The factory method, abstract factory, and prototype pattern can already be verified in Spec#. The property of the singleton pattern that a singleton class can only be instantiated once, cannot be verified by the current Spec# system, but since static invariants[11] are not implemented in Spec# yet, we have not covered it in this thesis.

6.3 Conclusions

This thesis presents a methodology for the visitor pattern, and an extension of the Spec# system based on the designed methodology.

With the design of a methodology we have presented a formalism of the visitor pattern that not only helps to verify the design pattern in Spec#, but also formalizes the properties of the visitor pattern.

We have extended the Spec# system with new language constructs, namely defines clauses and speconly methods, and we have improved the specification and verification of implementations of the visitor pattern.

More precisely, with the extended Spec# system we were able to verify important implementations of the visitor pattern with Spec#. Two characteristic implementations are the evaluation, and the simplification of an expression represented by an abstract syntax tree, for which the Spec# programs can be found in Appendix A.

Moreover we have simplified the specification for implementations of the visitor pattern, since we have reduced the amount of redundancy required to specify implementations of the pattern, and

since we have extended the specification language with the defines clause as a convenient way to specify the operation of a visitor.

Additionally our solution no longer requires programmers to write non-modular specifications for implementations of the visitor pattern, and finally with the extended Spec# system it is now possible to verify implementations of the visitor pattern, where subclasses of the visitors refine the specifications of their base classes.

Appendix A

Spec# Examples

To illustrate what programs can be verified we present an implementation of the visitor pattern with two characteristic visitor classes that we were able to verify. The structural classes represent abstract syntax trees for expressions with addition and constants. The structural classes and the abstract visitor class that defines the interface of the different visitors are defined as follows.

```
abstract class Expression {
    [Accept]
    public abstract void Accept(Visitor! v)
        modifies this.*, v.*;
}

class Add : Expression {
    [Rep] public Expression! left;
    [Rep] public Expression! right;

    public Add([Captured] Expression! left, [Captured] Expression! right)
        ensures this.left == left && this.right == right;
    {
        this.left = left;
        this.right = right;
    }

    public override void Accept(Visitor! v) {
        v.VisitAdd(this);
    }
}

class Literal : Expression {
    public int value;

    public Literal(int value)
        ensures this.value == value;
    {
        this.value = value;
    }

    public override void Accept(Visitor! v) {
```

```
        v.VisitLiteral(this);
    }
}

abstract class Visitor {
    [Visit]
    speconly void Visit(Expression! e);

    [Visit]
    public abstract void VisitAdd(Add! e);
        modifies this.*, e.*;

    [Visit]
    public abstract void VisitLiteral(Literal! e);
        modifies this.*, e.*;
}
```

A.1 Evaluation Visitor

The first visitor class represents visitor objects that evaluate a given expression to a value during traversal. The visitor class and a client of the visitor is defined as follows.

```
class EvalVisitor : Visitor {
    public int total;

    speconly override void Visit(Expression! e);
        defines abstract int eval(Expression! e);
        ensures eval(e) == old(eval(e));
        ensures total == eval(e);

    public override void VisitAdd(Add! e)
        defines override int eval(Add! e): eval(e.left) + eval(e.right);
    {
        int l = 0, r = 0;
        expose(e) {
            e.left.Accept(this);
            l = total;
            if (e.left != e.right) {
                e.right.Accept(this);
            }
            r = total;
        }

        total = l + r;
    }

    public override void VisitLiteral(Literal! e)
        defines override int eval(Literal! e): e.value;
    {
        total = e.value;
    }
}
```

```
class Client {

    void Run(Expression! e)
        modifies e.*;
    {
        EvalVisitor v = new EvalVisitor();
        e.Accept(v);
        assert v.total == EvalVisitor.eval(e);
    }

}
```

A.2 Non-Zero Visitor

The second visitor class represents visitor objects that duplicate a given expression during traversal and remove all literal objects with value zero. The visitor class and a client of the visitor is defined as follows.

```
class NonZeroVisitor : Visitor {
    public Expression newExpression;

    speconly override void Visit(Expression! e);
        defines abstract int eval(Expression! e);
        defines abstract bool noZero(Expression! e);
        ensures eval(e) == old(eval(e));
        ensures newExpression != null ==> newExpression.IsNew;
        ensures newExpression != null ==> eval(newExpression) == eval(e);
        ensures newExpression != null ==> noZero(newExpression);
        ensures newExpression == null ==> eval(e) == 0;

    public override void VisitAdd(Add! e)
        defines override int eval(Add! e): eval(e.left) + eval(e.right);
        defines override bool noZero(Add! e): noZero(e.left) && noZero(e.right);
    {
        Expression l = null, r = null;
        expose (e) {
            e.left.Accept(this);
            l = newExpression;
            if (e.left != e.right)
                e.right.Accept(this);
            r = newExpression;
        }

        assert l != null ==> eval(l) == eval(e.left);
        assert r != null ==> eval(r) == eval(e.right);

        if (l != null && r != null) {
            newExpression = new Add(l, r);
        } else if (l != null) {
            newExpression = l;
        } else if (r != null) {
```

```
            newExpression = r;
        } else {
            newExpression = null;
        }
    }

    public override void VisitLiteral(Literal! e)
        defines override int eval(Literal! e): e.value;
        defines override bool noZero(Literal! e): e.value != 0;
    {
        if (e.value == 0) {
            newExpression = null;
        } else {
            newExpression = new Literal(e.value);
        }
    }
}

class Client {

    void Run(Expression! e)
        modifies e.*;
    {
        NonZeroVisitor v = new NonZeroVisitor();
        e.Accept(v);
        assert v.newExpression != null
               ==> NonZeroVisitor.noZero(v.newExpression);
    }

}
```

Bibliography

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *LNCS*, volume 4111. Springer, 2006.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 2004.

[3] 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*. ACM, 2005.

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

[5] Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly, 2007.

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

[7] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for java. In *OOPSLA, 2008*. ACM, 2008.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[10] John Knudsen, Anders P. Ravn, and Arne Skou. Design verification patterns. In *Formal Methods and Hybrid Real-Time Systems*, volume 4700. Springer, 2007.

[11] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *LNCS*, volume 3582. Springer, 2005.

[12] Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity. A pattern enforcing compiler (PEC) for java: Using the compiler. In *APCCM '05: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*. Australian Computer Society, Inc., 2005.

[13] Tommi Mikkonen. Formalizing design patterns. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, 1998.

[14] Toufik Taibi and Fathi Taibi. Formal specification of design patterns and their instances. In *Computer Systems and Applications, 2006. IEEE International Conference*. ACM, 2006.