Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs

K. Rustan M. Leino [0] and Peter Müller [1]

[0] Microsoft Research, Redmond, WA, USA, leino@microsoft.com
[1] ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch

Manuscript KRML 189, 17 September 2009.

Abstract. Spec# is a programming system for the development of correct programs. It consists of a programming language, a verification methodology, and tools. The Spec# language extends C# with contracts, which allow programmers to document their design decisions in the code. The verification methodology provides rules and guidelines for how to use the Spec# features to express and check properties of interesting implementations. Finally, the tool support consists of a compiler that emits run-time checks for many contracts and a static program verifier that attempts to prove automatically that an implementation satisfies its specification. These lecture notes teach the use of the Spec# system, focusing on specification and static verification.

0 Introduction: What is Spec#

The Spec# programming system was built as a research effort to gain experience in programming with specifications, focusing on how one can specify object-oriented programs and how the specifications can be enforced dynamically and statically [2]. These lecture notes give a tutorial account of how to write and specify programs in Spec#. The aim here is for the specifications to be detailed enough that programs can be verified statically, using the Spec# static program verifier. The verifier checks that programs satisfy their specifications and that they do not lead to run-time errors. It assumes sequential execution; that is, it does not check for concurrency errors such as data races and deadlocks, and it might miss errors caused by insufficient synchronization of threads.

The verifier is run like the compiler—in fact, it can be turned on to run at "design time", in the background as the programmer types in the program [0]. Akin to the way a compiler performs separate compilation, the Spec# program verifier performs *modular verification*, which means that it can be applied to pieces of a program separately. A programmer interacts with the program verifier only by supplying program text and specifications and by receiving error messages, analogously to how a programmer interacts with the compiler. The goal of this tutorial is to provide the user with an understanding of the concepts that underlie the Spec# specifications, which will also help in deciphering the error messages.

The specifications of a program must describe the steady state of data structures and must account for the changes that such data structures undergo. This can be done in various ways. The way Spec# does this is to impose a programming discipline, a

methodology, that guides how specifications and programs are written. This methodology is closely tied to the specification constructs provided by the language, for example the **invariant** declaration. Through our experience, we have found that programming problems that fit the methodology can be specified and verified with ease; however, we have also found that it is too easy to fall outside the boundaries of what the methodology permits. For this reason, it is much easier to verify programs when they are designed with specification in mind from the start.

In the literature, the methodology used by Spec# has been called the *Boogie methodology*, since the Spec# programs are verified using a tool called Boogie. In retrospect, this is a bit confusing, because the Boogie language and tool are also used in applications that are unrelated to Spec#. To reduce confusion in this tutorial, we will use the words *Spec# methodology* and *program verifier*.

It is specifically not a goal of this tutorial to justify the Spec# methodology, only to explain how it is used. Also, this tutorial is not a reference manual; it is more of a cookbook for how to handle common situations. We conclude many sections in this tutorial with notes on advanced features that we cannot describe in detail here and with suggestions for further reading. These notes are marked with a "steep ascent" sign. We focus on static verification of programs, but occasionally add remarks that pertain to the dynamic checks that the Spec# compiler emits. Finally, we have done our best to explain what Spec# is today, which in many small ways differs from what is described in research papers that we and our colleagues have written; for example, the research papers use many variations of syntax, describe solutions to specification problems that have not been implemented in the Spec# programming system, spend many words and theorems justifying the soundness of the approach, sometimes show specifications in full detail whereas Spec# uses a host of defaults, and do not mention additional features and automation that is available in Spec#.

Installing and Using Spec#. The Spec# binaries (and sources) and installation instructions are available from http://specsharp.codeplex.com/. We recommend using Z3 [5] as the theorem prover for the Spec# program verifier; it can be installed from http://research.microsoft.com/projects/z3/. The Spec# installation requires Visual Studio. Once installed, Spec# can be used within Visual Studio or from the command line.

All examples presented in this tutorial are available online [11]. They compile and verify with the latest Spec# version (v1.0.21125), which requires Visual Studio .NET 2008. To try an example `File.ssc` from the command line, first compile the program to a library:

```
ssc /t:library /debug /nn File.ssc
```

and then run the program verifier

```
SscBoogie File.dll
```

The compiler option `/debug` produces a file `File.pdb` with debug information, which is needed by the program verifier. The `/nn` switch makes non-null types the default (see Section 1.0) and is used for all examples in this tutorial. Rather than running the verifier

separately, it is also possible to invoke it from the compiler by adding the `/verify` switch (which in some cases has the side effect of giving more detailed source locations in error messages). The `/help` option of the compiler and the verifier displays a list of all available options.

To run the examples inside Visual Studio, create a new Spec# project (from File → New → Project) and edit its project properties (by right-clicking on the name of the project in the Solution Explorer and then choosing Properties) as follows. In the section "Configuration Properties", set "ReferenceTypesAreNonNullByDefault", "RunProgramVerifier", and "RunProgramVerifierWhileEditing" to true. The compiler's parser and type checker and the program verifier will now run automatically in the background while you are editing your code. The compiler and verifier show any errors they detect using red or green squigglies at the location of each error. Hoving with the mouse over such a squiggly displays the error message. All error messages are also shown in the Error List (whose pane is made visible by View → Error List). To compile your program into executable code, build the project (using Build → Build Solution).

1 Basics

In this section, we go through a number of small examples that illustrate the syntax of some familiar specification constructs, as well as the use of some less familiar constructs. It also serves as an incomplete summary of features that Spec# adds to its basis, C#. We assume a basic familiarity with C-like syntax, like that found in Spec#, C#, Java, or C++. We also assume a basic familiarity with object-oriented concepts (for example, classes, instances, fields) and how these are represented in Java-like languages.

1.0 Non-null Types

One of the most frequent errors in object-oriented programs is to dereference the null reference. To eradicate this error, Spec#'s type system distinguishes between *non-null types* and *possibly-null types*. In this tutorial, we assume non-null types to be the default. In this mode, the type **`string`** is the type of all proper string objects, whereas the type **`string`**? includes the string objects plus **`null`**.

The class `NonNull` in Fig. 0 declares three string fields, two with a non-null type and one with a possibly-null type. On entry to a constructor, fields have zero-equivalent values, in particular, fields of reference type initially hold the null reference. Each constructor of a class is responsible for initializing non-null fields with non-null values.[0] If the class does not explicitly declare a constructor, a default constructor is implicitly added by the compiler. In that case, non-null fields have to be initialized using a field initializer in the declaration of the field. The constructor of class `NonNull` initializes `aString` through a field initializer and `anotherString` through an assignment inside the constructor body. It leaves `maybeAString` un-initialized.

[0] There are actually two kinds of constructors in the language, those that, explicitly or implicitly, call a constructor of the superclass and those that instead call another constructor of the same class (using **`this`**`(...)`). Here and throughout, we only discuss the first kind of constructor; in several ways, the other kind operates more like a method than a constructor.

```
class NonNull {
  string aString = "Hello";
  string anotherString;
  string? maybeAString;

  public NonNull() {
    anotherString = "World";
  }

  public int GetCharCount() {
    return aString.Length + maybeAString.Length;  // type error
  }
}
```

Fig. 0. An example using non-null and possibly-null types. The body of `GetCharCount` does not type check because it dereferences a possibly-null reference, `maybeAString`.

The Spec# type checker does not allow possibly-null references to be dereferenced. For instance, without further information, it flags the call `maybeAString.Length` in Fig. 0 as a type error. There are several ways to make this code type check. First, we could guard the call by a conditional statement (**if**) or expression. For instance, we could write the second summand as

```
(maybeAString != null ? maybeAString.Length : 0)
```

Second, we could add a specification (for instance, an inline assertion, see below) that expresses that `maybeAString` holds a non-null value. Third, we could convey the information to the type checker through a type cast. For example, we could have written

```
((string)maybeAString).Length
```

which casts `maybeAString` from the possibly-null type **string**? to the non-null type **string**. Here, since the type of the target is **string**, the dereference is permitted. The correctness of the cast is checked by the program verifier.[1]

A synonym for the non-null type **string** is **string**!.[2] The type cast can therefore also be expressed as (**string**!). In cases where the programmer only intends to cast the nullity aspect of the type, Spec# allows the cast to be written simply as `(!)`. Thus, the call could also have been written as `((!)maybeAString).Length`.

[1] As for other type casts, the compiler emits a run-time check for the cast. For this type cast, the run-time check will include a comparison with **null**.

[2] If the compiler is run in the mode where class names by default stand for the corresponding possibly-null type, then one routinely uses **string**! to denote the non-null type. Also, regardless of the compiler mode used, both inflections `?` and `!` are useful in the implementations of generic classes: if `T` is a type parameter constrained to be a reference type, then the naked name `T` stands for the actual type parameter (which might be a possibly-null type or a non-null type), `T?` stands for the possibly-null version of the type, and `T!` stands for the non-null version of the type.

1.1 Method Contracts

One of the basic units of specification in Spec# is the method. Each method can include a *precondition*, which describes the circumstances under which the method is allowed to be invoked, and a *postcondition*, which describes the circumstances under which the method is allowed to return. Consequently, an implementation of the method can assume the precondition to hold on entry, and a caller of the method can assume the postcondition to hold upon return. This agreement between callers and implementations is often known as a method *contract* [16].

Consider the method `ISqrt` in Fig. 1, which computes the integer square root of a given integer `x`. It is possible to implement the method only if `x` is non-negative, so

```
int ISqrt(int x)
  requires 0 <= x;
  ensures result*result <= x && x < (result+1)*(result+1);
{
  int r = 0;
  while ((r+1)*(r+1) <= x)
    invariant r*r <= x;
  {
    r++;
  }
  return r;
}
```

Fig. 1. A Spec# program that computes the positive integer square root of a given number. To try this example in Spec#, include this method in a class declaration like "`class Example { ... }`".

the method uses a **`requires`** clause to declare an appropriate precondition. The method also uses an **`ensures`** clause to declare a postcondition. This postcondition uses the keyword **`result`**, which refers to the value returned by the method.[3] The program verifier enforces preconditions at call sites and postconditions at all normal (that is, non-exceptional) exit points.[4] Note that a non-terminating method execution does not reach an exit point and, therefore, trivially satisfies its postcondition. The Spec# verifier does not check for termination.

Syntactically, a method can use any number of **`requires`** and **`ensures`** clauses, in any order. The effective precondition is the conjunction of the **`requires`** clauses and the

[3] Like **`value`** in C# (and Spec#), **`result`** is a context-sensitive keyword. In particular, **`result`** is reserved only in postconditions; elsewhere, `result` is just an ordinary identifier.

[4] Pre- and postconditions are also enforced dynamically by compiler-emitted checks. Through these dynamic checks, a Spec# programmer benefits from contracts even if the program verifier is never applied. If an entire program is successfully verified by the program verifier, then the dynamic checks are guaranteed never to fail and could therefore, in principle, be removed. With one exception—the **`assume`** statement, which we explain in Section 1.2—the dynamic checks form a subset of the checks performed by the program verifier.

effective postcondition is the conjunction of the **ensures** clauses. Other specification constructs in Spec# can be used cumulatively in a similar way. As Fig. 1 suggests, method contracts are declared between the method type signature and the method body; if the method has no body, as for abstract methods and interface methods, the contract follows the semicolon that for such methods ends the method type signature.

A postcondition is a *two-state predicate*: it relates the method's pre-state (the state on entry to the method) and the method's post-state (the state on exit from the method). To refer to the pre-state, one uses the **old** construct: **old**(E) refers to the value of expression E on entry to the method.[5] For example, consider the Counter class in Fig. 2. The postcondition of method Inc uses **old** to say that the final value of x is to be strictly

```
class Counter {
  int x;

  public void Inc()
    ensures old(x) < x;
  {
    x = 2*x;  // error
  }
}
```

Fig. 2. A simple class with an instance field that is modified by a method. The method implementation fails to establish the postcondition in the case that x was initially less than or equal to zero; thus, the program verifier reports a "possible postcondition violation" error.

greater than its initial value.[6] Regardless of **old**, an in-parameter mentioned in a method contract always refers to the value of the parameter on entry (in other words, the fact that the language allows in-parameters to be used as local variables inside the method body has no effect on the meaning of the contract), and an **out** parameter always refers to the value of the parameter on exit; **ref** parameters, which are treated as copy-in copy-out, are sensitive to the use of **old**.

The Counter example in Fig. 2 also shows that contracts operate at a level of abstraction that is not available directly in the code: the contract promises that x will be

[5] The **old** construct can be mentioned only in postconditions, not in, for example, inline assertions or loop invariants, see below. When those specifications need to refer to the pre-state value of an expression, one has to save it in an auxiliary local variable and use that variable in the specification.

[6] As a matter of consistent style, we prefer the operators < and <= over > and >=. This lets inequalities be read (by a human) from left to right, as if they were placed in order along a number line. For example, `0 <= x && x < N` "shows" x to lie between 0 and less than N; compare this to the form `x >= 0 && x < N`, which does not give the same visual queue. A common mistake is to write the negation of this condition as `0 < x || x >= N`. If all inequalities are turned the same way, the correct negation, `x < 0 || N <= x`, shows x as lying "left" of 0 or "right" of N. Though we use and advocate this style, it has no effect on the operation of the program verifier.

incremented, but does not let callers know by how much; the method body is thus free to make the amount of increment a private implementation decision. Note that the implementation in Fig. 2 does not live up to the postcondition; this could be fixed by adding a precondition such as `0 < x`.

With one major exception, expressions that are given as part of contracts (like the condition given in a **requires** clause) are like any other expressions of the language. For example, they must type check and be of an appropriate type (**bool** in the case of **requires**). The major exception is that expressions in contracts are restricted to be *side-effect free* (*pure*). For example, the declaration **requires** `x++` is not allowed. The reason for this restriction is that contracts are supposed to describe the behavior of the program, not to change it. In particular, whether run-time contract checking is enabled (during testing) or disabled (in production code to increase performance) must not influence the behavior of the program.

1.2 Inline Assertions

While method contracts indicate conditions that are expected to hold on method boundaries, the **assert** statement can be used in code to indicate a condition that is expected to hold at that program point. Unlike a method contract, which spells out a contract between a caller and an implementation, the **assert** statement only provides a form of redundancy in the code—after all, the asserted condition is supposed to be a logical consequence of the surrounding code. This redundancy can be useful, because it gives a programmer a way to check his understanding of the code. In particular, the program verifier will attempt to prove that the assert does indeed always hold. We shall see several uses for this familiar statement throughout this tutorial.

For example, adding the statement

```
assert r <= x;
```

anywhere between the declaration of `r` and the **return** statement in Fig. 1 will cause the program verifier to check that `r` is bounded by `x`.

Sometimes a programmer expects a condition to hold somewhere in the code, but the condition cannot be proved as a logical consequence of the surrounding code and specifications. For example, it may be that the condition follows from a pattern of calls to methods that have not been given strong enough formal specifications. In such cases, using an **assert** would cause the program verifier to issue a complaint. More appropriate in such a case is to use an **assume** statement, which (like the **assert**) generates a run-time check but (unlike the **assert**) is taken on faith by the program verifier. There are good uses of **assume**, but one needs to be aware that it trades static checking for dynamic checking. Hence, by writing down a condition that does not hold—an extreme example would be **assume false**—the program verifier will be satisfied and the incorrect assumption will not be revealed until the condition fails at run-time. Good engineering will pay special attention to **assume** statements during testing and manual code inspections.

The compiler and type checker also pay some attention to **assert** and **assume** statements. For example, the type checker considers inline assertions for its non-null analy-

sis. For instance, the type error in method `GetCharCount` (Fig. 0) can be prevented by adding **`assume`** `maybeAString != ` **`null`** before the **`return`** statement.

1.3 Loop Invariants

A loop may prescribe an infinite number of different iteration sequences. Clearly, it is not feasible to reason about every one of these individually. Instead, the loop's iterations are reasoned about collectively via a *loop invariant*. The loop invariant describes the set of states that the program may be in at the beginning of any loop iteration.

The program verifier treats loops as if the only thing known at the beginning of an iteration is that the loop invariant holds. This means that loop invariants must be sufficiently strong to rule out unreachable states that otherwise would cause the program verifier to generate an error message. For example, the condition `r <= x` holds on every loop iteration in Fig. 1, but this loop invariant by itself would not be strong enough to prove that the method establishes its postcondition. The program verifier enforces the loop invariant by checking that it holds on entry to the loop (that is, before the first iteration) and that it holds at every *back edge* of the loop, that is, at every program point where control flow branches back to the beginning of a new iteration.

The loop in method `ISqrt` in Fig. 1 is a bit of a special case and coincides with the form of the simple loops usually employed in teaching material. What's special is that the loop has only one exit point, namely the one controlled by the loop guard, which is checked at the beginning of each loop iteration. In this special case, one can conclude that the loop invariant and the negation of the loop guard hold immediately after the loop. (In the case of `ISqrt`, this condition is exactly what is needed to prove the method postcondition. In many other cases, this condition is stronger than needed for the proof obligations that follow the loop, in the same way that an inductive hypothesis in mathematics is usually stronger than the theorem proved.) In the general case of a loop with multiple exits, one cannot conclude that the loop invariant holds immediately following the loop, but it is still true that the loop invariant holds at the beginning of every iteration, including at the beginning of the last iteration, the (partial) iteration in which control flow exits the loop.

For example, consider the method in Fig. 3, which performs a linear search, backwards. Note that the loop invariant `0 <= n` holds at the beginning of every loop iteration, but it does not always hold after the loop, and ditto for the loop invariant with a quantifier. The program verifier explores all possible ways through the loop body to determine what may be assumed to hold after the loop.

Speaking of quantifiers, the example shows both an existential quantifier (**`exists`**) and a universal quantifier (**`forall`**). Each bound variable in a quantifier must be given a range with an **`in`** clause. Here, the range is a half-open integer range; for example, the range `(0: a.Length)` designates the integers from `0` to, but not including, `a.Length`. The program verifier currently supports only integer ranges in quantifiers, as well as ranges over array elements that can be converted into quantifiers over integer ranges. For example, the postcondition can equivalently be written as

```
ensures result == exists{int x in a; x == key};
```

```
bool LinearSearch(int[] a, int key)
  ensures result == exists{int i in (0: a.Length); a[i] == key};
{
  int n = a.Length;
  do
    invariant 0 <= n && n <= a.Length;
    invariant forall{int i in (n: a.Length); a[i] != key};
  {
    n--;
    if (n < 0) {
      break;
    }
  } while (a[n] != key);
  return 0 <= n;
}
```

Fig. 3. A linear search that goes through the given array backwards. The example illustrates a loop with multiple exit points. In addition, the example illustrates the use of several **`invariant`** declarations, which are equivalent to conjoining the conditions into just one **`invariant`** declaration, and the use of quantifier expressions. Note that the interval `(x: y)` is half open, that is, `i in (x: y)` says that `i` satisfies `x <= i && i < y`.

Quantified expressions are not confined to use in contracts, but can also be used in code. For example, one could implement the linear-search method with a single line:

```
  return exists{int x in a; x == key};
```

However, the program verifier currently does not understand quantifiers in code, so it complains that it cannot prove the postcondition for this single-line implementation.[7]

Not all loop invariants need to be supplied explicitly. The program verifier contributes to the loop invariant in two ways beyond what is declared. First, it performs a simple interval analysis, which amounts to that inequality relations between a variable and a constant often do not need to be supplied explicitly. For example, for a basic loop like

```
  s = 0;
  for (int i = 0; i < a.Length; i++) {
    s += a[i];
  }
```

the program verifier infers the loop invariant `0 <= i`; together with the loop guard `i < a.Length`, the program verifier thus automatically verifies that this loop body al-

[7] As an implementation detail, the program verifier does not work directly on the source code, but on the bytecode emitted by the compiler. For contracts, the compiler also spills out some meta-data that helps the program verifier. But to the program verifier, a quantifier in code just looks like the loop that the compiler emits for it, and that loop does not have a loop invariant that would permit verification.

ways accesses the array within its bounds. As another example, the program verifier infers the loop invariant `0 <= r` for `ISqrt` in Fig. 1, though that condition is not needed to verify the method. The simple interval analysis does not understand values derived from the heap, for example, so it is not able to infer the loop invariant `0 <= n` in Fig. 3.[8] Second, the program verifier infers and limits what the loop modifies. For instance, it performs a simple syntactic analysis to infer that `ISqrt` does not modify `x`. We will have more to say about modifications in Section 1.4.

Recall that the program verifier does not check that programs terminate. If a programmer wants help in checking that a loop terminates, it is possible to manually insert such checks. For example, the program in Fig. 4 computes the value of a *variant function* (see, *e.g.*, [15]) at the beginning of the loop body and then checks, just before the end of the body, that the variant function is bounded and that the iteration has strictly decreased the variant function. The responsibility for that the manually inserted code actually does imply termination (for example, that all paths to the next loop iteration are considered) lies with the user.

The current version of the Spec# program verifier does not check for arithmetic overflow. Hence, for example, the error of computing `mid` in Fig. 4 as:

```
int mid = (low + high) / 2;  // potential overflow
```

is not checked. Similarly, any overflow in the multiplication in Fig. 2 is not detected.

1.4 Accounting for Modifications

It is important that a caller can tell which variables a method may modify. For illustration, consider class `Rectangle` in Fig. 5.

To inform its callers that only `X` and `Y` are modified, method `MoveToOrigin` uses a postcondition that specifies the values of `Dx` and `Dy` to be unchanged. Another way of accomplishing this is to use a **`modifies`** clause, like in the contract of method `Transpose`. If a method's **`modifies`** clause does not explicitly list some field of **`this`**, then the **`modifies`** clause implicitly includes **`this.`**`*`, which means that the method is allowed to modify any field of **`this`**. More precisely, unless the method has a **`modifies`** clause that designates a field of **`this`** or explicitly lists **`this.`**`*`, **`this.`**`**` (explained in Section 3.1), or **`this.`**`0` (explained below), then the method contracts gets an implicit **`modifies this.`**`*`. The default **`this.`**`*` is why the previous examples we have shown do not complain about illegal modifications.

There are subtle differences between using a postcondition to exclude some modifications (from the default **`this.`**`*`), like `MoveToOrigin` does, and using a **`modifies`** clause to allow certain modifications, like method `Transpose` does. The former allows temporary modifications inside the method body, whereas the latter does not. For instance, the code `f++; f--` is considered a side effect that needs to be accounted for in the **`modifies`** clause. Moreover, the former allows fields in superclasses and subclasses

[8] The program verifier also implements some more powerful domains for its abstract-interpretation inference [3], including the polyhedra abstract domain [4]. These can be selected with the program verifier's `/infer` option.

```
int BinarySearch(int[] a, int key)
  requires forall{int i in (0:a.Length), int j in (i:a.Length); a[i]<=a[j]};
  ensures -1 <= result && result < a.Length;
  ensures 0 <= result ==> a[result] == key;
  ensures result == -1 ==> forall{int i in (0: a.Length); a[i] != key};
{
  int low = 0;
  int high = a.Length;

  while (low < high)
    invariant 0 <= low && high <= a.Length;
    invariant forall{int i in (0: low); a[i] < key};
    invariant forall{int i in (high: a.Length); key < a[i]};
  {
    int variant = high - low;  // record value of variant function
    int mid = low + (high - low) / 2;
    int midVal = a[mid];

    if (midVal < key) {
      low = mid + 1;
    } else if (key < midVal) {
      high = mid;
    } else {
      return mid; // key found
    }
    assert 0 <= variant;  // check boundedness of variant function
    assert high - low < variant;  // check that variant has decreased
  }
  return -1;  // key not present
}
```

Fig. 4. A method that performs a binary search in array a. The precondition says the array is sorted, and the postconditions say that a negative result value indicates the key is not present and that any other result value is an index into the array where the key can be found. The example also illustrates a hand-coded termination check, which uses a variant function. Finally, the example uses the short-circuit implication operator ==>, which is often useful in specifications, but may also be used in code. As is suggested by the textual width of the operator, ==> has lower precedence than && and ||, and the if-and-only-if operator <==> has even lower precedence.

```
public class Rectangle {
  public int X, Y;
  public int Dx, Dy;

  public void MoveToOrigin()
    ensures X == 0 && Y == 0;
    ensures Dx == old(Dx) && Dy == old(Dy);
  {
    X = 0;  Y = 0;
  }

  public void Transpose()
    modifies Dx, Dy;
    ensures Dx == old(Dy) && Dy == old(Dx);
  {
    int tmp = Dx;  Dx = Dy;  Dy = tmp;
  }

  public void Disturb(Rectangle r)
    modifies r.*;
  {
    X = r.Y;  r.X = Y;
    Dx = min{Dx, r.Dx};
    r.Dy = max{X, Dy + r.Dy, 100};
  }

  public void CopyPositionTo(Rectangle r)
    modifies this.0, r.X, r.Y;
  {
    r.X = X;  r.Y = Y;
  }

  public Rectangle Clone()
  {
    Rectangle res = new Rectangle();
    res.X = X;
    res.Y = Y;
    res.Dx = Dx;
    res.Dy = Dy;
    return res;
  }
}
```

Fig. 5. An example that shows several ways of specifying modifications. Method `Disturb`, which performs some arbitrary changes to the rectangles **`this`** and `r`, includes uses of Spec#'s built-in **`min`** and **`max`**, here applied to lists of 2 and 3 elements, respectively.

to be modified, whereas the latter does not. We defer further discussion of subclass and virtual-method issues until Section 1.5.

Method `Disturb` in Fig. 5 obtains the license to modify fields of parameter `r` by including `r.*` in the **`modifies`** clause. In addition, it is allowed modifications of **`this`**`.*` by default, as usual.

Method `CopyPositionTo` modifies two fields of its parameter `r`, but does not modify any field of **`this`** (unless **`this`** and `r` happen to be the same). To specify that behavior, the method explicitly lists in its **`modifies`** clause the special form **`this`**`.0`. By itself, **`this`**`.0` does not refer to any field, but has the effect that the default **`this`**`.*` is not added. So if **`this`** and `r` happen to be the same object, `CopyPositionTo` may modify fields of **`this`** because it may modify fields of `r`. If **`this`** and `r` are different, the method may modify `r.*`, but not fields of **`this`**.

Method `Clone` illustrates that new objects may be modified without declaring these modifications in the **`modifies`** clause. In other words, a **`modifies`** clause constrains the modification only of those objects that were allocated in the pre-state of the method. Besides newly allocated objects, there are other objects whose modification is implicitly permitted. We discuss those in Section 2.0.

Spec# does not feature conditional **`modifies`** clauses (like in JML [8]), which would allow a method to include a modification term only in certain situations. Instead, the method must include all possible modifications in the **`modifies`** clause and then use **`ensures`** clauses to say when certain fields are not modified.

Array elements can also be listed in **`modifies`** clauses. In Fig. 6, method `Swap` affects only elements `i` and `j` of the given array. Method `Reverse` can change any and all elements of `b`, but must leave array `a` unchanged. The figure also shows a method `Caller`, which calls the other two methods and demonstrates some of the properties that the specifications of those methods allow the caller to conclude. Note how the **`assert`** statements let us confirm our understanding of what the program verifier does with the specifications.

To reason about the behavior of a loop, it is also important to have modifies information. In Spec#, loops do not have explicit **`modifies`** clauses; instead, they inherit the **`modifies`** clause of the enclosing method. For example, consider the following method:

```
void ContrivedModifications()
  requires 8 <= Dx;
  modifies X, Y;
{
  Y = 125;
  while (X < 27) {
    X += Dx;
  }
  assert 8 <= Dx;
  assert Y == 125;  // error reported here
}
```

The method's **`modifies`** clause grants the loop license to modify `X` and `Y`, but not `Dx`. Therefore, the program verifier knows that `8 <= Dx` remains true throughout the

```
public void Swap(int[] a, int i, int j)
  requires 0 <= i && i < a.Length;
  requires 0 <= j && j < a.Length;
  modifies a[i], a[j];
  ensures a[i] == old(a[j]) && a[j] == old(a[i]);
{
  int tmp = a[i];  a[i] = a[j];  a[j] = tmp;
}

public void Reverse(int[] a, int[] b)
  requires a.Length == b.Length && a != b;
  modifies b[*];
  ensures forall{int i in (0: a.Length); b[i] == a[a.Length-1-i]};
{
  int low = 0;
  int high = a.Length;
  while (low < high)
    invariant high + low == a.Length;
    invariant forall{int i in (0: a.Length), i < low || high <= i;
                     b[i] == a[a.Length-1-i]};
  {
    high--;
    b[low] = a[high];
    b[high] = a[low];
    low++;
  }
}

public void Caller(int[] a)
  requires 100 <= a.Length;
{
  int[] b = new int[a.Length];
  int x = a[57];
  int last = a.Length - 1;
  Reverse(a, b);
  assert x == a[57];         // Reverse leaves a unchanged
  assert b[last - 57] == x;  // this is where a[57] ends up
  Swap(b, 20, 33);
  assert b[20] == a[last - 33];  // b[20] and b[33] were swapped
  assert b[last - 57] == x;      // Swap leaves b[last-57] unchanged
}
```

Fig. 6. Examples that show the `modifies` clause syntax for array elements. The quantifier in the loop invariant of `Reverse` uses a filter expression, `i < low || high <= i`. Alternatively, for this universal quantifier, the filter could have been written as an antecedent of an implication `==>` in the quantifier's body.

method. Note, however, that no analysis is done to determine that this loop does not make use of its license to modify `Y`; hence, the program verifier assumes nothing about the value of `Y` after the loop, and an explicit loop invariant about `Y` is required in order to prove the last assertion in the example.

1.5 Virtual Methods

Calls to virtual methods are dynamically bound. That is, the method implementation to be executed is selected at run time based on the type of the receiver object. Which implementation will be selected is in general not known at compile (verification) time. Therefore, Spec# verifies a call to a virtual method `M` against the specification of `M` in the *static* type of the receiver and enforces that all overrides of `M` in subclasses live up to that specification [16, 14]. This is achieved through specification inheritance [6]: an overriding method inherits the precondition, postcondition, and **`modifies`** clause from the methods it overrides. It may declare additional postconditions, but not additional preconditions or **`modifies`** clauses because a stronger precondition or a more permissive **`modifies`** clause would come as a surprise to a caller of the superclass method.

Class `Cell` in Fig. 7 declares a virtual setter method for the `val` field. The override in subclass `BackupCell` is allowed to declare an additional postcondition. It also has to satisfy the inherited specification. In particular, method `BackupCell.Set` has to live up to the implicit **`modifies`** clause of `Cell.Set`. This example shows that for virtual methods, the default **`this`**`.*` is preferable over a more specific **`modifies`** clause such as **`this`**`.val` because the former allows subclass methods to modify additional fields declared in subclasses. Class `GrowingCell` attempts to implement a cell whose value can never decrease. Since callers of `Cell.Set` cannot anticipate the extra precondition, it is rejected by the Spec# compiler.

1.6 Object Invariants

The data associated with an object usually takes on many fewer values than the types of the fields would allow. For example, the implementation `Rectangle` in Fig. 5 may keep the width and height fields `Dx` and `Dy` as non-negative integers, even though their types would also admit negative values. Furthermore, the designer of the class may decide to set both `Dx` and `Dy` to `0` whenever the area of a rectangle is `0`. Such properties can be captured as *object invariants*. For example, class `Rectangle` may declare

```
invariant 0 <= Dx && 0 <= Dy;
invariant Dx == 0 || Dy == 0 ==> Dx == 0 && Dy == 0;
```

The object invariant is checked to hold at the end of each constructor of the class. The program verifier also checks that every update to a field maintains the object invariant. For example, with the invariants above, the program verifier checks that any value assigned to `Dx` is non-negative. However, it is not always possible to maintain object invariants with every assignment. For example, if `Dx` is changed from a positive value to `0` (or vice versa), then the second invariant above requires `Dy` to undergo a similar change; this means that there will be some point in the program when one of `Dx` and `Dy`

```
using Microsoft.Contracts;

public class Cell {
  [SpecPublic] protected int val;

  public virtual void Set(int v)
    ensures val == v;
  { val = v; }
}

public class BackupCell : Cell {
  [SpecPublic] protected int backup;

  public override void Set(int v)
    ensures backup == old(val);
  {
    backup = val;
    val = v;
  }
}

public class GrowingCell: Cell {
  public override void Set(int v)
    requires val <= v;  // error
  { base.Set(v); }
}
```

Fig. 7. An example illustrating specification inheritance. The precondition in `GrowingCell.Set` is rejected by the compiler. The custom attribute `[SpecPublic]` (declared in the `Microsoft.Contracts` namespace, which is conveniently included with a **`using`** declaration) allows a non-public field to be mentioned in public specifications. We discuss better ways to support information hiding in Section 7.

has been updated but the other has not yet been updated accordingly. To allow object invariants to be broken temporarily, Spec# includes a block statement **expose**. While an object is exposed, its invariants need not hold; instead, they are checked at the end of the **expose** block.

For example, if a `Rectangle` method wants to increase both the width and height by `10`, it would do the following:

```
expose (this) {
  Dx += 10;
  Dy += 10;
}
```

Without the **expose** statement, the program verifier would complain with the somewhat cryptic message "Assignment to field `Rectangle.Dx` of non-exposed target object may break invariant: ...". The **expose** statement is not always needed, however. For example, if instead of adding `10` to `Dx` and `Dy`, the method were to double each of the fields:

```
Dx *= 2;
Dy *= 2;
```

then no **expose** is needed, since each statement maintains the class invariants.

To explain what is going on, we say that an object is in one of two states: *mutable* or *valid*. When an object is in the mutable state, its object invariants need not hold and its fields can freely be updated. When an object is in the valid state, its object invariant is known to hold. Fields of a valid object are allowed to be updated only if the update maintains all invariants. An object starts off as mutable and remains mutable until the end of the constructor. After its construction, **expose** statements are used to temporarily change an object from valid to mutable.

We will have much more to say about object invariants in the rest of the tutorial. While things will get more complicated, the following basic intuitions will remain the same: object invariants describe the steady state of the data of an object and there are times when object invariants may be temporarily violated. An additional issue that we will encounter is that an object invariant can be enforced only if it is known to the program verifier, so in the modular setting where only some of the program's classes are visible, not all expressions are admissible as object invariants and more machinery is needed to check those object invariants that are admissible.

2 Working with Object Invariants

In this section, we take a deeper look at working with objects and their invariants.

2.0 Aggregate Objects and Ownership

Abstractly, objects provide certain state and functionality, but the implementation is rarely limited to the fields and methods of a single object. Instead, the fields of the object reference other objects, often of different classes, and those objects reference further objects, and so on. In other words, the implementation of a class usually builds on other classes. We say that the joint functionality provided by these objects combine into providing one *aggregate object*, and we say that the sub-objects or sub aggregate objects are *components* of the larger aggregate object.

In Spec#, fields that reference a component of the aggregate are declared with the `[Rep]` attribute, where "rep" stands for "representation". This makes it possible for the program text to distinguish between component references and other object references that a class may have.

To keep track of which objects are components of which aggregates, Spec# uses the notion of *object ownership*. We say that an aggregate object *owns* its component objects. For example, for an object `b` of type `Band` in Fig. 8, `b` is the owner of `b.gt`, as indicated by the `[Rep]` attribute.[9]

[9] Ownership describes the structure of objects and is used by the program verifier. However, no ownership information is kept at run time, so there are no dynamic checks that correspond to the static checks performed by the program verifier. Consequently, the only way to detect ownership-related errors is to apply the program verifier; at run time, such errors go undetected.

```
using Microsoft.Contracts;

class Band {
  int gigs;
  [Rep] Guitar gt;
  Accountant accnt;

  public void Play() {
    expose (this) {
      gigs++;
      gt.Strum();
    }
  }

  public Band() {
    gt = new Guitar(10);  // ...
  }

  // ...
}
```

Fig. 8. A simple example that shows a representation field `gt`, that is, a field that references a component of the enclosing aggregate object. Notice that the **`using`** declaration is needed in order to use `Microsoft.Contracts.Rep` unqualified. The example also shows another field, `accnt`, which references an object that is not a component of the aggregate. Finally, the example shows a typical method that operates on the aggregate object by calling methods on its components.

We explained before that an object is either in the mutable state (where its invariants need not hold) or in the valid state (where its invariants are known to hold), and that the **`expose`** statement is used to temporarily bring a valid object into the mutable state. To take ownership into account, we subdivide the valid state. If a valid object has no owner object or its owner is mutable, then we say that the object is *consistent*. This is the typical state in which one applies methods to the object, for there is no owner that currently places any constraints on the object. If the valid object does have an owner and that owner is in the valid state, then we say the object is *committed*. Intuitively, this means that any operation on the object must first consult with the owner. Figure 9 illustrates a typical heap configuration.

An implicit precondition of a method is that the target be consistent. This implies that all components of the target are committed. Thus, to operate on a component, the method body must first change the target into the mutable state, which implies that all of its component change from committed to consistent.

For example, method `Play` in Fig. 8 wants to invoke the `Strum` method on **`this`**`.gt`. Doing so without the **`expose`** statement would result in an error:[10]

[10] The line and column numbers in the error messages refer to the files on the tutorial web page [11], but we abbreviate the file names here.

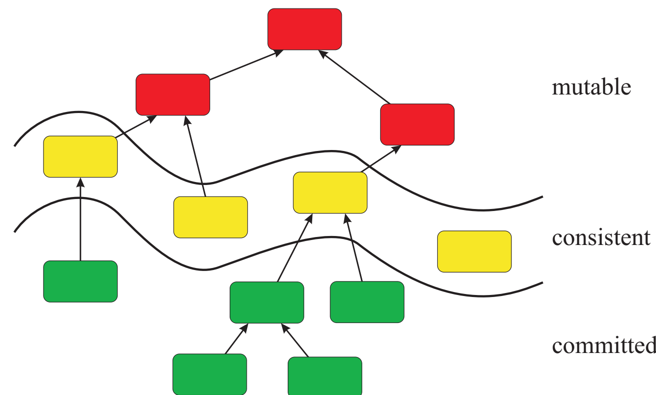

Fig. 9. A typical heap configuration. Objects are denoted by boxes; arrows point to an object's owner. Every heap has a thin layer of consistent objects that separate the mutable objects from the committed. The **expose** statement temporarily shifts this layer downward by making a consistent object mutable (and the objects it owns consistent).

```
Fig8.ssc(14,7): The call to Guitar.Strum() requires target object to be peer
consistent (owner must not be valid)
```

Most often when the program verifier produces this error, the program is either missing an **expose** statement or is trying to operate on an object whose owner is unknown. Sometimes, the latter is due to a missing `[Rep]` declaration.

Components of aggregate objects are implementation details that are not relevant to clients of the aggregate. Therefore, whenever a method may modify the state of an aggregate object, it is also allowed to modify its components, without mentioning those components in the **modifies** clause. For instance, since method `Play` may modify the state of its receiver (by its default **modifies** clause **this**.*), it may implicitly also modify the state of the `Guitar` object `gt`, which is a component of the `Band` aggregate. So, in summary, there are three cases in which a method has the permission to modify a field `o.f`: when `o.f` is listed (implicitly or explicitly) in the **modifies** clause, when `o` has been allocated during the execution of the method, and when `o` is a committed component of an object that may be modified by the method. To make use of the second option, postconditions sometimes contain `o.IsNew`, which expresses that an object `o` has been allocated by the method.

2.1 Ownership-based Invariants

An object invariant of an aggregate is allowed to constrain the state of its components. For example, the `Band` class in Fig. 8 may expect its guitarist to be reasonably good:

```
invariant 7 <= gt.Level;
```

Or perhaps it wants to relate the guitarist's participation with the band's number of gigs:

```
invariant gigs <= gt.PerformanceCount;
```

Not all expressions are admissible as object invariants. For example, there are restrictions on which subexpressions can be dereferenced. Spec# permits an object invariant if the program verifier has a way of checking that the invariant holds whenever the object is valid. One such way is to use ownership: An *ownership-based* invariant of an object may depend on the state of that object and its components. The program verifier can check ownership-based invariants by enforcing that the components of an aggregate object are changed only while the aggregate is mutable. So, “consulting with the owner” is done by exposing the aggregate.

The invariants above are admissible ownership-based invariants, because the field they dereference (`gt`) is a `[Rep]` field. If `gt` were not declared as `[Rep]`, then trying to include the invariants above in class `Band` would result in the following error, pointing to the dereference of `gt`:

```
Fig8.ssc(8,18): Expression is not admissible: it is not visibility-based[11],
and first access 'gt' is non-rep thus further field access is not admitted.
```

Suppose `Band` includes the invariant about `gt.PerformanceCount` above, and suppose the `Strum` method is specified to increment `PerformanceCount` by `1`. Then the **`expose`** statement in Fig. 8 correctly maintains the invariant, even though the update `gigs++` by itself may cause the invariant to be temporarily violated. On the other hand, if `gigs` were incremented *before* the **`expose`** statement, then the program verifier would flag an error, because invariants of valid objects must be maintained with every assignment.

We have explained that mutable objects are not subject to invariant checking, whereas each field update of a valid object must maintain the object’s invariants. But because of the principle that updates of component objects must first consult the owner, field updates are disallowed on committed objects. For example, if the body of method `Play` started with the update

```
gt.Level++;
```

then the program verifier would complain:

```
Fig8.ssc(12,5): Target object of assignment is not allowed to be committed
```

We mentioned that an implicit precondition of a method is that its target be consistent. This is important for modular verification: Knowing that the target’s owner is already exposed allows the body of `Guitar.Strum` to update `PerformanceCount` even if it is not aware of the `Band` invariant about `gt.PerformanceCount`. Moreover, it means that method `Play` can make any change to the fields of the `Band`, as long as it maintains the `Band` invariant, and it does not need to be concerned about the invariants declared in classes that may (directly or transitively) own the `Band` object.

From what we have seen so far, public methods might as well wrap their entire body inside an **`expose (this)`** statement. While that might be a good rule of thumb, it is not

[11] We discuss visibility-based invariants at the end of this subsection.

always appropriate. First, one might argue that trying to make **expose** blocks as small as possible is a good idea, because that makes it more clear where invariants might be temporarily violated. Second, changing **this** to mutable disables certain operations, in particular those that require **this** to be consistent. An example of that situation arises if a public method calls another public method on **this**.

Spec# also supports so-called *visibility-based* invariants, which allow the invariant expression to dereference fields that are not declared with `[Rep]`. However, there is another admissibility condition: A visibility-based invariant may dereference a field only if the declaration of the invariant is visible where the field is declared. This allows the static verifier to check for every field update that all objects whose visibility-based invariants depend on that field are exposed. Visibility-based invariants are useful to specify invariants of object structures that are not aggregates. The web site for this tutorial [11] contains an example of visibility-based invariants. Further details can be found in our research paper on object invariants [9].

In addition to ownership-based and visibility-based invariants, a dereference of a field `f` in an invariant is admissible on the grounds that the value of `f` does not change. For example, if `f` is a **readonly** field or is declared in an `[Immutable]` class, then any invariant can depend on `f`. More generally, the invariant is admissible if the `f` field belongs to a *frozen* object, one whose fields will never change again. Spec# already has a notion of immutable types, but we omit details here, because we are working on replacing it with an implementation of the more flexible notion of frozen objects [12].

2.2 Subclasses

A central facility provided by a subclass mechanism is the ability to extend a class with more state. This has an effect on invariants and ownership, so we now refine the notions we have introduced earlier.

A *class frame* is that portion of an object that is declared in one particular class, not its superclasses or subclasses. For example, an object of type `ResettingController`, see Fig. 10, has three class frames: one for `ResettingController`, one for `Controller`, and one for the root of the class hierarchy, **object**.

Each class frame can contain its own object invariants, which constrain the fields in that class frame. For example, class `Controller` declares an invariant that constrains `rate`, but the subclass `ResettingController` does not mention the superclass fields.

We refine the notions of mutable and valid to apply individually to each class frame of an object. For example, an object of type `ResettingController` can be in a state that is valid for class frames `ResettingController` and **object** and mutable for class frame `Controller`. We say an object is consistent or committed only when all its class frames are valid. In other words, our terms "consistent" and "committed" apply to the object as a whole, whereas "mutable" and "valid" apply to each class frame individually.

The **expose** statement changes one class frame of an object from valid to mutable. The class frame to be changed is indicated by the static type of the (expression denoting the) object. For example, in class `ResettingController`, `expose ((Controller)this)` exposes the `Controller` class frame of the target, whereas the statement `expose (this)` exposes the `ResettingController` class frame of the same object. Omitting the first of these **expose** statements leads to the following complaint:

```
using Microsoft.Contracts;
using System;

public class Controller {
  [Rep] protected Sensor s0;
  [Rep] protected Sensor s1;
  protected bool alarm;
  protected int rate;

  invariant s0.measurement != s1.measurement ==> alarm;
  invariant rate == (alarm ? 10: 2);

  public Controller() {
    s0 = new Sensor();
    s1 = new Sensor();
    rate = 2;
  }

  // ...
}

public class ResettingController : Controller {
  DateTime alarmTriggered;
  int clearedAlarms;
  invariant 0 <= clearedAlarms;

  public void CheckAlarm()  // call periodically
  {
    if (alarmTriggered.AddSeconds(5) <= DateTime.UtcNow &&
        s0.measurement == s1.measurement) {
      expose ((Controller)this) {
        alarm = false;
        rate = 2;
      }
      expose (this) {  // optional
        clearedAlarms++;
      }
    }
  }
}
```

Fig. 10. An example that shows a superclass and a subclass (we omitted the declaration of class `Sensor`). The invariant declared in each class is treated independently of the other. The invariants of `Controller` express that an alarm is triggered when the measurements from the two duplicate sensors are different. When an alarm has been signaled, the sampling rate of the sensor goes up. Class `ResettingController` clears an alarm in case the last divergence between the measurements was at least five seconds ago.

```
Fig10.ssc(32,9): Error: Assignment to field Controller.alarm of non-exposed
target object may break invariant: rate == (alarm ? 10: 2)
```

Fields of valid class frames can be updated (without an **expose** statement) provided the update maintains the invariant; for example, the second **expose** statement in Fig. 10 is not needed. As before, however, field updates are not allowed on committed objects.

Once an object has been fully constructed, only one class frame of an object is allowed to be mutable at a time. For example, if one tried to nest the two **expose** statements in Fig. 10, the program verifier would issue an error:

```
Fig10.ssc(34,17): Object might not be locally exposable
```

As a final adjustment to working with subclasses, we refine the notion of ownership: an owner is an (object, class) pair. As with **expose** statements, static types and enclosing contexts are used to indicate the class-component of this pair. For example, for an object o with a [Rep] field f, the owner of o.f is (o,C), where C is the class that declares f.

In the discussion above, an object invariant constrains the fields declared in the same class. Sometimes it is necessary to specify object invariants that relate fields from different class frames. For instance, class ResettingController might contain an invariant that requires alarmTriggered to have some default value whenever alarm is false. An invariant is allowed to mention a field from a superclass if that field is declared with attribute [Additive]. To update an additive field, one has to expose the object for the class declaring that field and all of its subclasses, which is done one class at a time with a special **additive expose** statement. This ensures that the class frame with the invariant is mutable when the field is updated. The tutorial web site contains an example with a additive fields. Additive fields are the default in many of our papers [1, 9, 10]. The difference between additive and non-additive fields is discussed in another publication [13].

2.3 Establishing Object Invariants

Each constructor of a class is responsible for initializing the fields of the class in accordance with the object invariant. A constructor starts off mutable for the object being constructed and the enclosing class. Therefore, fields of **this** declared in that class can be assigned to without exposing the receiver, as illustrated by the assignments in the constructor of Controller (Fig. 10). The class frame is changed to valid at the end of the constructor, which is therefore where the invariant is checked.[12]

[Rep] attributes on fields can be seen as special object invariants that constrain the owner of the object stored in the field. Any assignment to a [Rep] field must preserve that special invariant, even when the class frame containing the field is mutable. In the Controller example, the [Rep] fields s0 and s1 do not admit the value **null**, so the constructor must assign to them. In Spec#, typical constructors produce objects that are un-owned, so the right-hand sides of the assignments to s0 and s1 are un-owned Sensor objects. So how is the ownership relation instituted? Any assignment to a [Rep] field also has the effect of setting the owner of the right-hand side of the

[12] If the program verifier has occasion to report an error for the default constructor, the source location it uses is that of the name of the class.

assignment. Thus, Spec# automatically institutes the ownership relation when a [Rep] field is being assigned to. Such an assignment requires the right-hand side to have no owner or to already have the desired owner. That is, the assignment might set an owner of a previously un-owned object, but it won't automatically *change* an existing owner.

Until an object has been initialized for all of its class frames, it is not consistent (it has mutable class frames) and, thus, cannot be used as receiver or argument to methods that expect consistent arguments. We discuss how to work with partially-initialized objects in the following advanced remark.

Non-null types express a special kind of invariant that needs to be established by each constructor. The virtual machine initially sets all fields of a new object to zero-equivalent values, in particular, fields of reference types to **null**. So before the new object has been fully initialized, it would be unjustified to assume that non-null fields actually contain non-null values.

Until the initialization of an object is completed, we say that the object is *delayed*, meaning that it is in a raw state where we can rely neither on the non-nullness of its fields nor on its object invariants. Moreover, field values of delayed objects are themselves allowed to be delayed. By default, the **this** object is delayed inside constructors.

A delayed object is in general not allowed to escape from its constructor. However, sometimes it is useful to call methods on delayed objects or to pass a delayed object as an argument to a method call. This is permitted if the callee method or its parameter is marked with the attribute [Delayed]. The consequence of this choice is that the method is then not allowed to assume fields of non-null types to have non-null values, let alone assume the object invariant to hold.[a]

An alternative is to mark the constructor with the attribute [NotDelayed]. This requires that all non-null fields of the class be initialized before an explicit call of the superclass (aka *base class*) constructor, **base**. A constructor can make a **base** call to a [NotDelayed] superclass constructor only if it itself is marked as [NotDelayed]. A consequence of this design is that after the **base** call, the **this** object is fully initialized and no longer delayed. Therefore, it can be passed to methods without the [Delayed] attribute.

Fähndrich and Xia [7] describe the details of delayed objects. Examples for delayed objects and explicit **base** calls can be found on the tutorial web page.

[a] Any reference-valued parameter of a method, not just the receiver, can be marked with [Delayed]. However, there is a bug in the current version of the program verifier that makes the verification of methods with more than one [Delayed] parameter unsound.

3 Owners and Peer Groups

In this section, we explore more dimensions of ownership, especially sets of objects with the same owner, so-called *peers*.

3.0 Peers

The ownership model introduced so far allows aggregate objects to maintain invariants and to assume consistency of their components, for instance, when calling a method of a component. However, not all interacting objects are in an aggregate-component relationship. For example, a linked-list node n interacts with its successor node n.next,

but n.next is usually not thought of as a component of n. Rather, n and n.next have a more equal relationship, and both nodes may be part of the same enclosing aggregate object. Therefore, both nodes are more appropriately declared as *peers*, that is, objects with the same owner. This is accomplished by the attribute [Peer].

A general guideline is to use [Rep] wherever possible, because it strengthens encapsulation and simplifies verification. [Peer] is appropriate when two objects are part of the same aggregate (that is, the aggregate object has direct access to both objects) or when the objects are part of a recursive data structure (such as the nodes of a linked list).

Another guideline is to use [Rep] when the field references an object whose type or mere existence is an implementation detail of the enclosing class, and to use [Peer] when the field references an object that can also be accessed by clients of the enclosing class. For example, in a typical collection-iterator pattern, the iterator has a field that references the collection. This field is best marked as [Peer], because the collection is not an implementation detail of the iterator and clients of the iterator may also access the collection directly.

As another example illustrating the use of peer objects, consider the two classes in Fig. 11. The Dictionary class makes use of some unspecified number of Node objects, which by the [Peer] declaration on the next field are all peers. The Dictionary class maintains a reference to the first and last of the Node objects, and declares the head and tail fields as [Rep]. Note that, in general, a Dictionary object owns many more objects than the two that are referenced directly by its fields.

Let us consider how the peer relation is instituted. The situation is analogous to [Rep] attributes: if pr is a [Peer] field, then assignment o.pr = x automatically institutes a peer relation between o and x. Essentially, the assignment sets the owner of x to be the same as the owner of o. But, as for [Rep] fields, Spec# won't change an owner, so the operation requires the right-hand side to start off having no owner or already having the desired owner. Moreover, an assignment to a [Peer] field requires that the target object not be committed—one is allowed to add peers to an object o only at times when the invariant of o's owner need not be maintained. We will illustrate these rules with the two versions of an Insert method shown in Fig. 11.

The body of method InsertA first records the value of head in local variable h, and then sets head to a newly allocated Node. Since head is declared with [Rep], the assignment to **this**.head also sets the owner of the new Node to **this**. Then, since next is a [Peer] field, the assignment head.next = h sets the owner of h to be the same as the owner of head, which is **this**; in this case, the owner of h was already **this**, so the assignment to the [Peer] field is allowed and has no net effect on the ownership of h.

The **expose** statement in InsertA is required. First, the class invariant in Dictionary would be broken by the assignment to head if head were initially **null**. By using the **expose** statement, the code is allowed to temporarily violate the invariant; the assignment to tail restores the invariant. Second, the assignment of a [Peer] field requires the target object not to be committed; without the **expose** statement in InsertA, the program verifier would issue a complaint:

```
Fig11.ssc(23,7): Target object of assignment is not allowed to be committed
```

```
public class Dictionary {
  [Rep] Node? head = null;
  [Rep] Node? tail = null;
  invariant head == null <==> tail == null;

  public bool Find(string key, out int val) {
    for (Node? n = head; n != null; n = n.next) {
      if (n.key == key) { val = n.val;  return true; }
    }
    val = 0;  return false;
  }

  public void InsertA(string key, int val) {
    expose (this) {
      Node? h = head;
      head = new Node(key, val);  // new.owner = this;
      head.next = h;              // h.owner = head.owner;
      if (tail == null) {
        tail = head;              // head.owner = this;
      }
    }
  }

  public void InsertB(string key, int val) {
    expose (this) {
      Node n = new Node(key, val);
      if (head != null) {
        Owner.AssignSame(n, head);  // n.owner = head.owner;
        n.next = head;              // head.owner = n.owner;
      }
      head = n;                     // n.owner = this;
      if (tail == null) {
        tail = head;                // head.owner = this;
      }
    }
  }
}

class Node {
  public string key;
  public int val;
  [Peer] public Node? next;
  public Node(string key, int val) { this.key = key;  this.val = val; }
}
```

Fig. 11. An example showing a linked list of key-value pairs. A `Dictionary` object owns all the `Node` objects it reaches from head. The two variations of an `Insert` method illustrate two different ways to accomplish the same result. The comments in the code show the automatic and manual ownership assignments.

Method InsertA first updates head and then sets the next field of the new object. An alternative would be to first set the next field and then update head, as in:

```
Node n = new Node(key, val);
n.next = head;
head = n;
```

But this code fragment poses a problem: when the [Peer] field n.next is assigned to, Spec# will want to change the owner of the right-hand side, head, to the owner of n, but since head already has an owner (and it is not the owner of n, for n has no owner at this point), Spec# would have to do an ownership change, which it won't do. In cases like this, when one wants to change the owner of the target object, not of the right-hand side, one has to resort to a manual ownership assignment. This is illustrated in method InsertB, which uses the method Microsoft.Contracts.Owner.AssignSame. While the automatic ownership assignments take care of the case when the right-hand side is **null**, the manual ownership assignments do not; hence, the **if** statement in InsertB. The remaining automatic ownership assignments in InsertB, see the comments in the figure, have no net effect.

It is worth noting the direction of ownership assignments. Automatic ownership assignments always affect the right-hand side of the field assignment. In contrast, the Owner methods that can be used to manually change ownership change the owner of the first parameter. For illustration, see the first two comments in method InsertB.

3.1 Peer Consistency

In the Dictionary example above, the peer objects, of type Node, are designed together with the owner class, Dictionary. This need not always be the case. Sometimes, objects may naturally occur as related abstractions, but without a specific client context in mind. For example, a collection object may have a number of iterator objects. The collection and its iterators are programmed together, but they can be used in any client context, just like a collection by itself could be.

We show a simple collection and iterator example in Fig. 12. Each iterator holds a reference to the collection it is iterating over. The iterator does not own the collection—clients that use collections need the ability to acquire ownership; besides, a collection may have several iterators, and they cannot all own the collection. Instead, the field c is declared as [Peer]. Let us first explore why the field is declared [Peer] at all (as opposed to having no ownership attribute) and then, in the next subsection, explore how the peers are instituted.

Consider the GetCurrent method of the iterator. To determine if it has gone through all the elements in the collection, it compares its index, i, with the number of elements in the collection, Count. Then, it accesses the collection's array, which requires the index to be less than the length of that array. The correctness of this operation relies on the invariant Count <= a.Length, which holds of valid collection objects. But how do we know the collection to be valid?

One could add a precondition to GetCurrent, using the property IsConsistent, which is available in all objects and which yields whether an object is consistent:

```
public class Collection {
  [Rep] internal int[] a;
  public int Count;
  invariant 0 <= Count && Count <= a.Length;

  public int Get(int i)
    requires 0 <= i && i < Count;
    modifies this.0;
  { return a[i]; }

  public Iterator GetIterator() {
    Iterator iter = new Iterator();
    Owner.AssignSame(iter, this);
    iter.c = this;
    return iter;
  }

  // ...
}

public class Iterator {
  [Peer] internal Collection? c;
  int i = 0;
  invariant 0 <= i;
  public bool MoveNext() {
    i++;
    return c != null && i < c.Count;
  }
  public int GetCurrent() {
    if (c != null && i < c.Count) return c.a[i];
    else return 0;
  }
  public void RemoveCurrent()
    modifies this.**;
  {
    if (c != null && i < c.Count) {
      for (int j = i+1; j < c.Count; j++)
        invariant 0 < j && c != null && 0 < c.Count;
      {
        expose (c) { c.a[j-1] = c.a[j]; }
      }
      expose (c) { c.Count--; }
    }
  }
}
```

Fig. 12. A rudimentary collection and iterator example, illustrating the use of peer objects independent of any owning context. The example leaves out many common features of collections and iterators that are not the focus here. The access mode **internal** indicates that the field can be accessed by other classes in the same *assembly* (which is .NET speak for “module”).

```
public int GetCurrent()
  requires c != null && c.IsConsistent;
```

However, such a precondition reveals implementation details (one would have to declare field `c` as `[SpecPublic]`, change its access level, or use abstraction mechanisms).

Since it is quite common for objects to rely on the consistency of some of their peers, Spec# uses another approach. It adds an implicit precondition to every method that requires all in-bound parameters, including the target parameter, *and all their peers* to be consistent. When an object and all its peers are consistent, we say the object (and each of its peers) is *peer consistent*. Peer consistency is also an implicit postcondition of all out-bound parameters and return values.

Requiring peer consistency as the precondition of `GetCurrent` is more than we need for the example, but it has the advantage that `GetCurrent` does not explicitly need to name the associated collection in its precondition.

To summarize, because of the implicit precondition of peer consistency, all we need to do to verify method `GetCurrent` is to declare the field `c` as `[Peer]`. The peer consistency of **`this`** then implies the (peer) consistency of `c`, and thus the invariant of `c` can be assumed to hold and the array access `c.a[i]` can be verified to be within bounds.

The peer-consistency precondition makes it very easy to call methods of peer objects. Suppose the direct array access `c.a[i]` in method `GetCurrent` were replaced by a call `c.Get(i)`. The implicit precondition of `Get` is that the receiver, `c`, be peer consistent. Since the iterator **`this`** and its collection `c` are peers, the peer consistency of **`this`** (which is the implicit precondition of `GetCurrent`) implies the peer consistency of `c`. Note that it is not necessary (nor possible) to enclose a call to `c.Get(i)` in an **`expose`** `(`**`this`**`)` statement, whereas if `c` had been declared a `[Rep]`, it would have been necessary (and possible) to do so.

The peer-consistency precondition also allows a method to expose peer objects. For instance, method `RemoveCurrent` exposes the collection `c` to remove an element. Its **`modifies`** clause uses the wild-card **`this`**`.**`, which denotes all fields of all peers of **`this`**. A more usable example would add a postcondition to recover information about the state of the collection and the iterator.

Any time a method invokes a method on some object, say `x`, it needs to live up to the precondition of the callee, which includes establishing the peer consistency of `x`. If `x` is a parameter of the enclosing method or has been obtained from a **`new`** call or as the return value of some other method, then `x` is typically already known to be peer consistent. When `x` is obtained from a field of an object, say `o.f`, then peer consistency typically follows from the `[Rep]` or `[Peer]` attribute on the declaration of field `f`. Omitting such an attribute often causes the program verifier to issue a complaint like:

```
Sec3.1.ssc(12,5): The call to Demo.M() requires target object to be peer
consistent
```

because it cannot prove `x` and its peers to be valid. Peer consistency also includes not being committed. Therefore, if `f` is a `[Rep]` field, one needs to expose `o` before calling a method on `o.f`. Otherwise, the verifier reports an error such as:

```
Sec3.1.ssc(13,5): The call to Demo.M() requires target object to be peer
consistent (owner must not be valid)
```

Occasionally, it is useful to call methods on a receiver or with arguments that are not peer consistent. For instance, a method might want to expose its receiver and then, from inside an **expose** statement, call an auxiliary method on the receiver (see the tutorial web site for an example). Spec# provides two ways of avoiding the implicit precondition of peer consistency.

All implicit specifications can be turned off by marking a method with [NoDefaultContract]. However, often one would like to turn off implicit specifications more selectively.

Methods parameters can also be declared with the attribute [Inside]. The implicit precondition for an [Inside] parameter p of static type C says that p is exposed for C, and it says nothing about the peers of **this**. To mark **this** as [Inside], place the attribute on the method.

A consequence of using [Inside] is that the method cannot assume the object invariant of the [Inside] receiver or parameter, and the caller of the method cannot assume the object invariant to hold upon return. Instead, an [Inside] method must write explicit pre- and postconditions that explain which conditions are to hold on entry and exit.

3.2 Peer Groups

In the previous subsection, we motivated the notion of peer consistency and the use of that condition as an implicit method precondition. Let us now explore what peers and peer consistency mean when an object has no owner.

An object always has some (possibly empty) set of peers, regardless of whether or not the object has an owner. We say that an object always belongs to a *peer group*. When an owner is assigned, the entire peer group is assigned that owner, so the peer relation among the objects of the peer group is preserved. For example, for a [Rep] field rp, an assignment o.rp = x transfers the entire peer group of x into ownership by o. For a [Peer] field pr, an assignment a.pr = x merges the entire peer group of x into the peer group of a. (As we mentioned before, both of these kinds of ownership assignments require x to be un-owned or to already have the desired owner.) The use of peer groups and the implicit precondition of peer consistency mean that an object does not need to reveal to its clients what its peers are, which provides information hiding.

Peer groups may be enlarged and may be merged, but Spec# does not currently provide any way to break up a peer group.

The treatment of peer groups and ownership in Spec# also means that the order in which objects are made peers or assigned owners does not matter. For example, if one wants to establish a situation where o owns both a and b, one can first merge the peer groups of a and b and then set object o as the owner of the resulting peer group, or one can first set o as the owner of a and then merge the peer group of b into that of a. As an analogy, consider the process of going to dinner with some friends. One can either first gather a group of friends (analogy: peers) and then decide which restaurant (analogy: owner) to go to, or one can first decide which restaurant to go to and then find friends to come along.

4 Arrays

In this section, we explain how Spec# handles arrays, especially how non-null types and ownership work for array elements.

4.0 Covariant Array Types

C#, and therefore also Spec#, has covariant array types. That means that a variable of static type `T[]` can hold an array whose allocated type is `U[]`, where `U` is any subtype of `T`. The property that the elements of an array are indeed of the array's element type cannot be ensured by the static type system. Instead, it is enforced dynamically by the .NET virtual machine and also checked statically by the program verifier.[13]

For example, without further information, the method

```
void SetArrayElement(Controller[] a, int i)
  requires 0 <= i && i < a.Length;
  modifies a[i];
{
  a[i] = new Controller();  // possible error
}
```

will cause the program verifier to complain:

```
Sec4.0.ssc(6,5): RHS might not be a subtype of the element type of the array
being assigned
```

This warns about the possibility that a has allocated type, say, `ResettingController[]`, in which case it is not allowed to assign a `Controller` object into `a`.

To prevent this complaint, one has to convince the program verifier that array-element updates are correct, which usually boils down to showing that the allocated type of the array equals its static type, say `T[]`. In some cases, this can be determined without additional specifications, in particular if `T` is a **`sealed`** class or if `T` is a class that is internal to the assembly (*e.g.*, non-**`public`**) and the assembly does not define any subclasses of `T`. In other cases, one needs to write a specification that constrains the allocated type of the array.

For instance, in the example above, the following precondition takes care of the problem:

```
  requires a.GetType() == typeof(Controller[]);
```

The `GetType` method (which is defined for all references) returns an object (of type `System.Type`) that represents the allocated type of `a`, and the expression **`typeof`**`(T)`, where `T` denotes a type, returns a `System.Type` object that represents the type `T`. `GetType` may also be used in object invariants, which is useful when arrays are stored in fields.

[13] However, Spec# currently ignores co-variance errors that occur when an array with non-null element type is cast to an array of possibly-null elements. The compiler does not emit the necessary run-time check, and the verifier also ignores this issue.

4.1 Arrays of Non-Null Elements

Array types typically have the form `T?[]!`, meaning that the array itself is non-null, whereas the array elements are possibly-null instances of `T`. Either the `?` or the `!` can be omitted, depending on the defaults used by the compiler. Besides this common form, Spec# supports all other combinations of non-null and possibly-null, in particular, non-null element types as in `T![]!`.

Unlike fields of non-null types, whose initialization in the constructors of the class can be assured by syntactic definite-assignment rules, arrays of non-null elements are initialized by arbitrary code that follows the **new** allocation of the arrays. Until that initialization is completed, one cannot rely on the type of the array to accurately reflect the non-nullness of the array elements. For this reason, the Spec# type checker provides a special marker, in the form of a method `NonNullType.AssertInitialized`, which is used to indicate a program point where the initialization code has completed. The type checker will not give the array its declared non-null type until that point.

```
public void ExampleArrays() {
  string[] food = { "black-eyed peas", "red hot chili peppers", "cream" };
  WriteAll(food);

  string[] series = new string[3];
  series[0] = "The prime numbers start with 2, 3, 5, 7, 11, 13, 17";
  series[1] = "The Fibonacci numbers start with 0, 1, 1, 2, 3, 5, 8";
  series[2] = "The perfect numbers start with 6, 28, 496, 8128";
  NonNullType.AssertInitialized(series);
  WriteAll(series);

  string[] facts = new string[10];
  for (int n = 0; n < facts.Length; n++)
    invariant n <= facts.Length;
    invariant forall{int i in (0: n); facts[i] != null};
  {
    facts[n] = (n+1) + " ants is more than " + n + " elephants";
  }
  NonNullType.AssertInitialized(facts);
  WriteAll(facts);
}
public void WriteAll(string[] ss)
{
  foreach (string s in ss) {
    Console.WriteLine(s);
  }
}
```

Fig. 13. The `WriteAll` method takes a non-null array of non-null strings. Method `ExampleArrays` shows several ways of initializing arrays with non-null elements. Method `NonNullType.AssertInitialized` is declared in `Microsoft.Contracts`.

For illustration, Fig. 13 shows the initialization of three arrays. Array `food` is initialized in the same statement that allocates it, so the type checker can treat it as having type **string**[] immediately. Arrays `series` and `facts` are initialized by code sequences. Thus, before these arrays can be used as having type **string**[], the code must call `AssertInitialized`. At that call site, the program verifier checks that every array element is non-null.[14]

If a program tries to use the array element before the array has been given its declared type, the compiler will complain. For example, if the assignment to `series[2]` in Fig. 13 is replaced by `series[2] = series[1]`, the following type error results:

```
Fig13.ssc(13,17): Cannot store delayed value into non(or incompatibly)-
delayed location
```

despite the fact that the right-hand side of the assignment actually does have a non-null value at that time.

Also, if the code does not include a call to `AssertInitialized` for an array of non-null elements, the type checker complains:

```
Fig13.ssc(10,14): Variable 'series', a non-null element array, may not have
been initialized. Did you forget to call NonNullType.AssertInitialized()?
```

Perhaps confusingly, the source location mentioned in the error message points to where the array is declared, but this does not mean that `AssertInitialized` has to be called there.

4.2 Ownership of Arrays and Array Elements

Just like nullness, Spec# allows one to specify ownership independently for an array and its elements.

In Fig. 14, we show a class that uses an array of (possibly-null) `Step` objects. Method `AddStep` of the class queues up drawing steps and method `Apply` performs the work associated with these steps.

Ownership of Arrays. The last object invariant dereferences the array: `steps[i]`. To make this invariant admissible, the class declares the field `steps` to be `[Rep]`. Without the `[Rep]` attribute, the compiler's admissibility checker would report an error:

```
Fig14.ssc(11,39): Expression is not admissible: first access on array or
binding member must be rep.
```

Note that this invariant constrains the state of the array object, but not of the array elements. Therefore, this invariant does not require the array elements to be owned by the `DrawingEngine` object.

Arrays do not have object invariants. Therefore, they need not be exposed before an array element is updated. However, since an owning object might have an invariant

[14] At run time, `AssertInitialized` performs a dynamic check that the array elements are not **null**. The time needed to do so is proportional to the length of the array, but that is no worse than the time required to initialize the array in the first place.

```
using System;
using Microsoft.Contracts;

public class DrawingEngine {
  [Rep] [ElementsRep] Step?[] steps = new Step?[100];
  invariant 1 <= steps.Length;
  int cnt;
  invariant 0 <= cnt && cnt <= steps.Length;
  invariant forall{int i in (0: cnt); steps[i] != null};

  public void AddStep(byte op, int argX, int argY) {
    if (cnt == steps.Length) { EnlargeArray(); }
    expose (this) {
      Step s = new Step(op, argX, argY);
      steps[cnt] = s;
      cnt++;
    }
  }
  void EnlargeArray()
    ensures cnt < steps.Length;
  {
    expose (this) {
      Step?[] more = new Step?[2*steps.Length];
      Array.Copy(steps, 0, more, 0, steps.Length);
      steps = more;
    }
  }
  public void Apply() {
    for (int i = 0; i < cnt; i++) {
      Step? s = steps[i];
      assert s != null;
      expose (this) { s.DoWork(); }
    }
    cnt = 0;
  }
}

class Step {
  public byte op;
  public int argX, argY;
  public Step(byte op, int x, int y) {
    this.op = op;  argX = x;  argY = y;
  }
  public void DoWork() { /* ... */ }
}
```

Fig. 14. This example class uses an array of owned `Step` objects. The call to `DoWork` in method `Apply` requires `s` to be peer consistent. This information follows from the `[ElementsRep]` attribute on `steps`, which says that the array elements are owned by the `DrawingEngine` object. Note that we use an **`assert`** statement in method `Apply` to convince the type checker that `s` is non-null. The verifier can prove this assertion using the third object invariant, but the type checker does not consider object invariants.

that constrains the state of the array, array-element updates, like `steps[cnt] = s` in `AddStep`, require the array not to be committed (that is, require the owner to be mutable). The enclosing **expose** statement temporarily changes **this** from valid to mutable and thus, since `steps` is a `[Rep]` field, changes the array `steps` from committed to peer consistent. Without the **expose** statement, the program verifier would complain:

```
Fig14.ssc(17,7): Target array of assignment is not allowed to be committed
```

The **expose** statement could be wrapped around just the array-element update, but wrapping it as shown in Fig. 14 also works.

Similarly, by its implicit precondition, the call to `Array.Copy` in `EnlargeArray` requires its parameters to be peer consistent. The **expose** statement puts `steps` into the required state. As in `AddStep`, this particular **expose** statement is shown wrapped around several statements, not just the call statement that needs it.

Ownership of Array Elements. The call `s.DoWork()` in method `Apply` requires the `Step` object `s` to be peer consistent. As we have discussed in Section 3.1, for objects stored in fields, peer consistency typically follows from ownership attributes on the fields. Here, `s` is stored in an array, and we use the `[ElementsRep]` attribute on the field `steps` to express that every non-null element of the array is owned by the enclosing `DrawingEngine` object.

Without `[ElementsRep]` on `steps`, the program verifier would produce several error messages for method `Apply`, complaining about the effects of `DoWork` and about the lack of peer consistency at the call to `DoWork`:

```
Fig14.ssc(34,23): method invocation may violate the modifies clause of the
enclosing method
Fig14.ssc(34,23): The call to Step.DoWork() requires target object to be
peer consistent
Fig14.ssc(34,23): The call to Step.DoWork() requires target object to be
peer consistent (owner must not be valid)
```

With the `[ElementsRep]` attribute, the code exposes **this**, which makes `steps[i]` peer consistent. Moreover, `Apply` is then allowed to modify the elements of `steps` because they are components of the `DrawingEngine` aggregate.

Spec# also provides an attribute `[ElementsPeer]`, which expresses that the array elements are peers of the object containing the `[ElementsPeer]` field.

Spec# requires all elements of an array to have the same owner, even if that owner is not specified by an [ElementsRep] or [ElementsPeer] attribute. For an array arr, the call Owner.ElementProxy(arr) yields an artificial object that is a peer of the elements of arr. This artificial object exists even if arr contains all **null** elements. The element proxy of a new array is initially un-owned. It is set when the array is assigned to an [ElementsRep] or [ElementsPeer] field. The element proxy can be used to query and modify ownership information for arr's elements. For instance, the call Owner.AssignSame(Owner.ElementProxy(arr), **this**) makes the element proxy of arr—and thus all current and future elements of arr—a peer of **this**. Like all ownership assignments, this call requires the element proxy to be un-owned or to already have the desired owner. Analogously to updates of [Rep] or [Peer] fields, assignments to array elements, like steps[cnt] = s in AddStep, make the right-hand side of the assignment a peer of the array's element proxy.

5 Generics Classes

Instead of using arrays, it is often more convenient to use generic collection classes. In this section, we illustrate how to write clients of generic classes. We do not discuss how to implement generic classes, because the implementation of generic classes in the Spec# compiler and verifier still needs improvement.

Figure 15 shows another version of class DrawingEngine from Fig. 14, this time using the generic class List. The implementation based on List is significantly simpler. One reason for this is that we can use a list of non-null Step objects, which simplifies the specifications. The details of dealing with a partially-filled array are hidden inside the List class.

Ownership for generics is very similar to arrays, with two differences. First, for instances of generics, one can specify the owner individually for each generic type argument. This is done by passing the number of the type argument to the attributes [ElementsRep] and [ElementsPeer] (starting with 0, of course). For instance, declaring a field

```
[ElementsPeer(0)] Dictionary<K,V> dict;
```

adds implicit checks and assumptions to all operations on dict that values of type K are peers of **this**. When the number is omitted, like in the declaration of steps in Fig. 15, the attribute refers to all type arguments.

Second, there are no automatic owner assignment when objects are passed to operations of generic classes. For instance, method AddStep has to assign an owner to the new object s before passing it to List's Add method. Omitting this assignment leads to the following complaint from the verifier:

```
Fig15.ssc(13,7): Error: The call to System.····.List<Step!>.Add(Step! item)
requires item to be a peer of the expected elements of the generic object
```

6 Capturing Parameters

A standard way to construct an aggregate object is to construct the components inside the constructor of the aggregate. For example, the constructor of the Band class in Fig. 8

```
using System.Collections.Generic;
using Microsoft.Contracts;

public class DrawingEngine {
  [Rep] [ElementsRep] List<Step> steps = new List<Step>();

  public void AddStep(byte op, int argX, int argY) {
    expose (this) {
      Step s = new Step(op, argX, argY);
      Owner.AssignSame(s, steps);
      steps.Add(s);
    }
  }

  public void Apply() {
    foreach (Step s in steps) {
      expose (this) { s.DoWork(); }
    }
    steps = new List<Step>();
  }
}
```

Fig. 15. The `DrawingEngine` from Fig. 14, this time using the generic class `List`. The `Step` class is unchanged. Like for arrays, attribute `[ElementsRep]` indicates ownership for the elements of the collection. However, the owner has to be set explicitly before an object is stored in the list.

initializes its `gt` field to a `Guitar` object that it allocates. Sometimes, a component is provided by a client of the aggregate, either during construction or via a method. This is useful, because it allows the client to customize the component, for example by allocating it to be of a particular subclass.

6.0 Customizing Rep Fields

Consider again the `Band` class, this time with a constructor and a method that accept a `Guitar` that is to become a component of the `Band`, see Fig. 16. Since `gt` is declared as `[Rep]`, the assignment `gt = g` will set the owner of `g` to **`this`**, and the operation requires `g` to be un-owned. This precondition and license to modify an owner are obtained by declaring the parameter with `[Captured]`. Intuitively, the `[Captured]` attribute says that the parameter is passed in but does not "come back out". More precisely, `[Captured]` says that the callee has the right to take ownership of the object referenced by the parameter, and that a caller should not expect to be able to directly use the object after the call.[15]

[15] Spec# currently does not support an `[ElementsCaptured]` attribute that would allow a method to capture the elements of an array or a generic collection.

```
class Band {
  [Rep] Guitar gt;

  public Band([Captured] Guitar g)
  {
    gt = g;
  }

  public void ReplaceGuitar([Captured] Guitar g)
    requires Owner.Different(this, g);
    ensures Owner.Is(g, this, typeof(Band));
  {
    gt = g;
  }

  // ...
}
```

Fig. 16. An example that shows how clients can, via either a constructor or a method, supply an object that is to become a component of the Band aggregate. This allows a client to supply an appropriate Guitar object. The [Captured] attribute allows the callee to assign ownership to a parameter. The method Owner.Is yields whether its first argument is owned by the class frame specified by its second and third argument—here, (**this**,Band).

The [Captured] attribute affects a method's (or constructor's) precondition and modifies clause[16], but it has no effect on the postcondition. So, without further specification, the caller does not get to find out how the parameter is captured. This is usually satisfactory when captured into a [Rep] field, since the [Rep] field is usually an implementation detail of the class. Nevertheless, it is possible to write an explicit postcondition. For example, a postcondition gt == g will do. Another way to do it is to use the Owner.Is predicate as shown for method ReplaceGuitar in Fig. 16.[17]

Here is a possible client of the Band:

```
Guitar g = new BumperGuitar();
Band r = new Band(g);
r.Play();
g.Strum();  // error
```

This client decides to use a particular Guitar subclass, BumperGuitar, for the Band it constructs. Note that after calling the Band constructor, the caller still has a reference to

[16] In fact, there is no explicit way of listing an owner "field" in a **modifies** clause. Even the **modifies** clause term p.* does not give the right to modify the owner of p. So, the only way to obtain the license to modify the owner of a parameter object is to use the [Captured] attribute.

[17] Using Owner.Is in the postcondition of the constructor is more involved because of its actual parameter **this**, which may not have been fully constructed yet—a Band subclass constructor may have more work to do and, thus, the object is delayed (see Section 2.3).

the captured object g. However, the caller is not able to invoke a method on g, because the caller cannot be sure that g is peer consistent (in fact, it will be committed).

Here is another client, somewhat contrived:

```
Band r = new Band();
Guitar g = new BumperGuitar();
r.ReplaceGuitar(g);
r.Play();
expose (r) {
  g.Strum();
}
```

From the information in ReplaceGuitar's postcondition, one can conclude that the **expose** statement makes g peer consistent. Therefore, this client's call to g.Strum() verifies.[18]

One more thing remains to be explained about the example in Fig. 16, namely the reason for the precondition of method ReplaceGuitar. Without this precondition, the program verifier would complain about the assignment in the method:

```
Fig16.ssc(21,5): when target object is valid, it is not allowed to have the
same owner as RHS, because that would cause a cycle in the ownership relation
```

Or, if the assignment to gt occurred inside an **expose** statement, the program verifier would issue a complaint at the end of that **expose** statement:

```
Fig16.ssc(23,5): All of the object's owned components must be fully valid
```

The reason for these errors is the following scenario: The [Captured] attribute on parameter g entails the precondition of g having no owner. But this precondition still allows g to have peers. Suppose that, on entry to the method, g and **this** were peers. Then, the implicit ownership assignment that takes place when assigning to the [Rep] field **this**.gt would create the incestuous situation that **this** owns g and yet **this** and g are peers! This is disallowed and is the reason for these errors.

The predicate Owner.Different says that its two arguments are not peers. By using it in the precondition of ReplaceGuitar, one avoids the errors above.

[18] One can draw the same conclusion had the method's postcondition been gt == g. However, it is not possible to prove, after the call to r.Play(), that r.gt == g. This is because the implicit modifies clause of r.Play() is r.*, which allows r.gt to be modified. But the modifies clause does not permit modifying an owner, which is why one can conclude that g is still owned by r at the time of the **expose**.

In addition to `Different`, the `Owner` class has a method `Same`. For static program verification, these two predicates are each other's negation. However, there is an important and subtle difference in their run-time behavior. In principle, all contracts could be checked at run time, but to keep the overhead reasonable, Spec# omits certain run-time information and run-time checks, for example, ownership information. Consequently, predicates like `Owner.Same`, `Owner.Different`, and `Owner.Is` cannot be computed at run time. Instead, these predicates all return **`true`** at run time. As long as these predicates are used in positive positions (that is, not negated) in contracts, the program verifier will enforce stronger conditions than are enforced at run time. So, to make a choice between `Same` and `Different`, use the one that you can use in a positive position. It would be good if Spec# enforced this "positive position rule" for the predicates concerned, but the current version of Spec# does not implement any such check.

6.1 Customizing Peer Fields

It is also possible to capture parameters into `[Peer]` fields. Consider the example in Fig. 17, which shows two constructors that establish a peer relationship between the object being constructed and the object given as a parameter.

```
public class Iterator {
  [Peer] public Collection Coll;

  public Iterator([Captured] Collection c)  // captures 'c'
    ensures Owner.Same(this, c);
  {
    Coll = c;  // c.owner = this.owner;
  }

  [Captured]
  public Iterator(Collection c, int x)  // captures 'this'
    ensures Owner.Same(this, c);
  {
    Owner.AssignSame(this, c);  // this.owner = c.owner;
    Coll = c;                   // c.owner = this.owner;
  }

  // ...
}
```

Fig. 17. An example that shows two ways of setting up a peer relationship in a constructor. The first constructor captures the parameter into the peer group of the `Iterator` being constructed; the second constructs an `Iterator` in the same peer group as the parameter. Alternatively, either postcondition could have been written as (the stronger) **`ensures`** `Coll == c`.

Both constructors take a parameter `c` and ensure that, upon return, **`this`** and `c` are in the same peer group. A newly allocated object—that is, **`this`** on entry to a constructor—

starts off in a new, un-owned peer group. Like any other method, unless its specification says otherwise, the constructor is not allowed to change this ownership information for **this**, or for any other parameter. Therefore, a **new** expression typically returns a new object in a new, un-owned (but not necessarily singleton) peer group.

The first constructor in Fig. 17 declares that it will capture the parameter, c. The ownership relation (or, rather, the peer-group relation) is instituted automatically when the [Peer] field Coll is assigned, as we have described in Section 3.0 and as suggested by the comments in Fig. 17. This solution is not always the best, because it requires the caller to pass in an un-owned collection c. A caller may be in a situation where the collection already has an owner and the caller wants to create a new iterator for that collection.

For these reasons, the second constructor in Fig. 17 shows the more common way to set up a peer relationship in a constructor. The [Captured] attribute on the constructor, which is to be construed as applying to the implicit **this** parameter, says that the instigating **new** expression may return with the new object being placed in a previously existing peer group (with or without an assigned owner). Thus "capturing" **this** instead of c is usually a good idea, since ownership relations previously known to the caller are unaffected. Moreover, since the object being constructed starts off with no owner, the body of the constructor can easily live up to the precondition of the ownership assignment it will effect. However, the automatic ownership assignment that is performed with peer-field updates goes the wrong direction, so the body needs to use a manual ownership assignment as shown in Fig. 17.

In the example, each of the two constructors declares a postcondition that tells callers about the ownership of the new object, namely that it will be a peer of the parameter c. This kind of postcondition is common when a peer relationship is established, but uncommon when a parameter is captured into a [Rep] field. The reason is the same as the reason for choosing between [Rep] and [Peer]: [Rep] denotes something of an implementation detail, promoting information hiding and letting the class write invariants that dereference the [Rep] field, whereas [Peer] is used when clients have an interest in the object referenced.

7 Abstraction

When specifying the methods of a class, it is desirable to write the method contract in terms of entities (fields, methods, or properties) that can be understood by clients without violating good principles of information hiding. The compiler enforces the following rules: Entities mentioned in the precondition of a method must be accessible to all callers; this means that they must be at least as accessible as the method itself. Entities mentioned in the postcondition of a method must be accessible to all implementations of the method; this means that contracts of virtual methods (which can be overridden in subclasses) cannot mention private entities and can mention internal entities only if the method or its enclosing class is internal. These rules ensure that callers

```
public class Counter {
  int inc;
  int dec;

  [Pure][Delayed]
  public int GetX()
  { return inc - dec; }

  public Counter()
    ensures GetX() == 0;
  {}

  public void Inc()
    ensures GetX() == old(GetX()) + 1;
  { inc++; }

  public void Dec()
    ensures GetX() == old(GetX()) - 1;
  { dec++; }
}
```

Fig. 18. A simple example that uses pure methods as a form of abstraction. Abstractly, a `Counter` is a value that can be retrieved by `GetX()`. Concretely, the value is represented as the difference between the number of increment and decrement operations performed. All method contracts are written in terms of the pure method `GetX()`, not the private fields `inc` and `dec`.

understand the preconditions they are to establish and implementations understand the postconditions they are to establish.[19]

But then, what can be used in contracts when most fields of the class are private implementation details? The solution lies in *abstracting* over those details. For that purpose, Spec# provides pure methods, property getters, and model fields, which we explain in this section.

7.0 Pure Methods

A method that returns a value without changing the program state can provide a form of abstraction. Such methods are called *pure* and are declared as such by the attribute `[Pure]`. Pure methods are not allowed to have side effects.

The program in Fig. 18 represents a counter that can be incremented and decremented. The current value of the counter is retrieved by the method `GetX()`, which is declared as `[Pure]`. The specifications of the constructor and methods are given in terms

[19] Spec# does not enforce similar restrictions on object invariants. So when a client exposes an object, it might not understand the condition that has to hold at the end of the **expose** block. However, since clients typically do not modify inaccessible fields, the program verifier can nevertheless often prove that the invariant is preserved.

of `GetX()`, which means that clients are separated from the implementation decision of storing the counter as the difference between the private fields `inc` and `dec`.

The way that the program verifier reasons about a pure method is via the specification of the method: if the (implicit and explicit) precondition holds, then the result will be a value that satisfies the postcondition.[20] In the common special case that the implementation of a pure method consists of a single statement **`return`** `E`, the compiler implicitly adds a postcondition **`ensures result`** `== E` to the pure method if that would make a legal postcondition, there is no explicit postcondition, and the method is not virtual.[21] In the example, it is this implicit postcondition of `GetX()` that lets the `Counter` constructor and the `Inc` and `Dec` methods be verified. Experience shows that this "postcondition inference" is usually desired. If one does not want the postcondition inference, the workaround is simply to provide an explicit postcondition, like **`ensures true`**, or to introduce a second statement in the body of the pure method, like:

```
[Pure] T MyPureMethod() { T t = E; return t; }
```

The implicit precondition for pure methods is different from the peer-consistency precondition of ordinary methods. To see why, consider the following possible method of the `Band` class (*cf.* Fig. 8):

```
[Pure]
public int MagicNumber() {
  return 3 + gt.StringCount();
}
```

where `gt` is a `[Rep]` field and `StringCount` is a pure method of the `Guitar` class. To compute its result, the implementation of this pure method makes use of values computed by components of the aggregate—here, the number of guitar strings. Such pure methods are common and of good form. However, if pure methods required peer consistency, then `MagicNumber` would have to expose **`this`** before calling `gt.StringCount()`. For side-effect-free methods, which can never break any invariants, exposing objects is unnecessary overhead. So, instead of requiring the receiver and its peers to be consistent (which implies that the owner is mutable), pure methods only require the receiver and its peers to be valid (saying nothing about the state of the owner). We call this condition *peer validity*. Note that if an object, like a `Band` object, is peer valid, then so are all its components, like the object `gt`.

[20] If a contract calls a pure method, then the contract itself must make sure that the pure method is called only when its precondition holds. In other words, a contract must always be well defined. However, the current version of the program verifier does not check all contracts to be defined. It does so for object invariants, but not for pre- and postconditions, for example. This current omission in the program verifier can sometimes lead to some confusion. In particular, if a pure method's precondition is violated in a contract, then the effect will be that the program verifier does not know anything about the result of the pure method.

[21] Since Spec# encourages the use of specifications, it would be natural if the compiler did the reverse: add an implicit implementation whenever the postcondition of a pure method has the simple form **`ensures result`** `== E`. However, this is currently not supported by the compiler.

The example in Fig. 18 uses the pure method GetX in the postcondition of the Counter constructor. Such usage is common but regrettably tricky. We explain it in the following advanced remark.

Since the constructor is by default delayed (see advanced remark in Section 2.3), the type checker enforces that any method it calls (on **this**) is also delayed, including any call that occurs in the postcondition. Therefore, the example declares GetX to be [Delayed]. A consequence of that declaration is that method GetX cannot rely on the non-null properties of the receiver or the object invariant, but that does not present any problem in the example.

If the pure method GetX had to rely on the object invariant, things would get more complicated. Consider a variation of class Counter that represents a non-negative integer and includes a precondition `0 < GetX()` for method Dec. Then, the class could be proved to maintain the invariant `dec <= inc`, and one may want to add the postcondition `0 <= result` to the pure method GetX(). To prove such a postcondition, the method would require the invariant to hold, which is at odds with GetX() being [Delayed]. Note that it is not an option to require the invariant through a precondition `requires dec <= inc`, because this precondition would reveal hidden implementation details.

To specify this variation of class Counter, it is necessary for the constructor and GetX both to be non-delayed, which is achieved by removing [Delayed] from GetX and adding [NotDelayed] to the constructor. Furthermore, the constructor must live up to the implicit precondition of GetX, which is peer validity. This is a problem, because at the end of the Counter constructor, **this** is valid only for Counter and its superclasses—any subclass constructors have not yet finished, so the object is not yet valid for those class frames. A simple, but sometimes untenable, way to address that problem is to declare Counter as a **sealed** class, which forbids it from having subclasses. The resulting variation of Fig. 18 is available on the tutorial web page.

An alternative to making Counter sealed is to use the [Additive] mechanism mentioned in an advanced remark in Section 2.2. This mechanism allows us to express that the receiver of GetX is valid for the class frames Counter and **object**, but mutable for all other class frames (if any). This is exactly the case after the body of Counter's constructor, at the time the call to GetX in the postcondition is evaluated. See the tutorial web page for an example.

7.1 Property Getters

In .NET, the usual way to write a GetX() method is to write a *property* X and to provide for it a *getter*. Similarly, a SetX(x) method is usually written as the *setter* for a property X. Properties are a facility that hides the underlying representation—X looks like a stored value, but its value may be computed in some way rather than being directly represented. Figure 19 shows the Counter class written in this more common way of using a property getter.

In common usage patterns, property getters tend to present abstractions to clients, and the implementations of property getters tend to have no side effects. Therefore, Spec# makes property getters [Pure] by default. What we said in Section 7.0, for example about inferred postconditions and about delayed type correctness and object invariants, also applies to property getters. Note in Fig. 19 that, syntactically, the attribute [Delayed] is placed on the getter itself, not on the enclosing property declaration.

```
public class Counter {
  int inc;
  int dec;

  public int X {
    [Delayed]
    get { return inc - dec; }
  }

  public Counter()
    ensures X == 0;
  {}

  public void Inc()
    ensures X == old(X) + 1;
  { inc++; }

  public void Dec()
    ensures X == old(X) - 1;
  { dec++; }
}
```

Fig. 19. The `Counter` class of Fig. 18 written with a property getter `X` instead of method `GetX()`.

7.2 Purity: the Fine Print

Pure methods (including pure property getters) need to be side-effect free. Consequently, a pure method's implicit **`modifies`** clause is empty and the method must not use any explicit **`modifies`** clause. But the handling of pure methods is more subtle than simple side-effect freedom. Pure methods must also satisfy the following three requirements:

- *Mathematical Consistency:* We reason about pure methods in terms of their specifications. For this reasoning to be sound, the specifications must be mathematically consistent. In particular, if pure methods are specified recursively, the recursion must be well founded. Spec# ensures well-foundedness by assigning a *recursion termination* level (a natural number) to each pure method. The specification of a pure method `M` may call a pure method `p.N` only if `p` is a `[Rep]` field (or a sequence of `[Rep]` and `[Peer]` fields, starting with a `[Rep]` field) or if `p` is **`this`** and `N`'s recursion termination level is strictly less than `M`'s. Consequently, for each such call, the height of the receiver in the ownership hierarchy decreases or the height remains constant, but the level decreases, which ensures well-foundedness. For most pure methods, Spec# infers a recursion termination level automatically; it can also be specified using the `[RecursionTermination(r)]` attribute.
- *Determinism:* Pure methods are usually used as if they had all the properties of mathematical functions. In particular, pure methods are generally expected to be deterministic. However, this is not the case when a pure method returns a newly

allocated object. The compiler performs some conservative, syntactic checks to ensure that this non-determinism is benevolent. These checks may instruct the user to apply the attribute `[ResultNotNewlyAllocated]` to the pure method (in which case the program verifier checks that the result value is not newly allocated) or apply the attribute `[NoReferenceComparison]` to a pure method that may potentially get two newly allocated references as parameters.

– *Dependencies:* It is important to know what state changes may interfere with the value returned by a pure method. This is achieved by specifying the *read effect* of each pure method using the `[Reads]` attribute. Typical values for this attribute are:
  - `[Reads(ReadsAttribute.Reads.Owned)]` (the default for pure instance methods), which allows the contract and body of the pure method to read the fields of **`this`** and any object that is transitively owned by **`this`**
  - `[Reads(ReadsAttribute.Reads.Nothing)]` (the default for static pure methods), which allows a pure method to depend only on **`readonly`** fields.
  - `[Reads(ReadsAttribute.Reads.Everything)]`, which allows a pure method to depend on all objects.

  The compiler uses various syntactic rules to enforce that the contracts of pure methods stay within their allowed read effects. However, neither the compiler nor the program verifier in the current version of Spec# checks the body of a pure method against its specified read effects. Hence, any violation of the read effects in the body goes undetected.

Further details and examples of all three requirements are available online.

7.3 Model Fields

Besides pure methods and property getters, Spec# provides yet another abstraction mechanism. In contrast to regular fields, a *model field* cannot be assigned to; the model fields of an object `o` get updated automatically at the end of each **`expose`** `(o)` block [10]. The automatic update assigns a value that satisfies a constraint specified for the model field.

Figure 20 shows another version of the `Counter` class. The model field `X` is declared with a **`satisfies`** clause whose constraint holds whenever the object is valid.

A model field is simpler to reason about than a pure method. First, its value changes only at the end of **`expose`** blocks, whereas the value of a pure method may change whenever a location in the read effect of the pure method is modified. Second, a model field can be read even for mutable objects, whereas a pure method typically requires its receiver to be valid. This makes it much easier to use model fields in constructors (note that the code in Fig. 20 does not require a `[Delayed]` attribute) and object invariants.

However, model fields are more restrictive than pure methods. First, they have no parameters. Second, a **`satisfies`** clause may depend only on the fields of **`this`** and objects transitively owned by **`this`** (like a pure method marked with the attribute `[Reads(ReadsAttribute.Reads.Owned)]`). So, a general guideline is to use pure methods or property getters when model fields are too restrictive; otherwise, model fields are the better choice.

```
public class Counter {
  protected int inc;
  protected int dec;

  model int X {
    satisfies X == inc - dec;
  }

  public Counter()
    ensures X == 0;
  {}

  public void Inc()
    ensures X == old(X) + 1;
  {
    expose (this) { inc++; }
  }

  public void Dec()
    ensures X == old(X) - 1;
  {
    expose (this) { dec++; }
  }
}
```

Fig. 20. Class `Counter` of Fig. 18 with a model field `X` instead of method `GetX()`. The updates of `inc` and `dec` must be done inside an **`expose`** block to ensure that `X` is being updated accordingly.

A **`satisfies`** clause does not have to specify a single value for the model field. Especially in abstract classes, it is often useful to give a weak **`satisfies`** clause and then declare additional **`satisfies`** clauses in subclasses. The value of a model field `o.f` satisfies the **`satisfies`** clauses of those classes for which `o` is valid. The program verifier checks that the **`satisfies`** clauses are feasible, that is, that there is a value that satisfies them. This check fails for the following model field, because there is no odd number that can be divided by 6:

```
model int Y {
  satisfies Y % 2 == 1 && Y % 6 == 0;
}
```

Like for pure methods, the verifier applies heuristics to find an appropriate value. When the heuristics are too weak, it is also possible to convince the verifier of the feasibility by providing a suitable value using a **`witness`** clause in the declaration of a model field:

```
model int Z {
  satisfies Z % 2 == 1 && Z % 5 == 0;
  witness 5;
}
```

8 Conclusions

Learning to use a program verifier requires a different kind of thinking than is applied in some other programming contexts today. It is harder than making sure that the program sometimes works—it forces the programmer to think about all possible inputs, data-structure configurations, and paths. The programmer does not need to imagine all cases up front, because the program verifier will do this exhaustively. But the programmer constantly needs to face the question "How am I going to convince the program verifier that this part of my program design is correct?". This process, by itself, has side benefits. For example, it can encourage simpler designs and better use of information hiding. Also, the specifications that are written while interacting with the verifier record design decisions that programmers otherwise have to reconstruct manually from the code. Knowing what is involved in program verification can also improve one's programming practices in other languages, since it raises one's awareness of correctness issues and teaches the use of contracts at module interfaces.

Spec# is a state-of-the-art programming system for programming with specifications and applying program verification. Nevertheless, the system is not nearly as simple to use as we wish it were. For example, it is impossible to go very far without understanding what it means for an object to be "consistent". Therefore, learning how to use the system takes patience, and experience will show how best to handle certain situations.

We hope this Spec# tutorial provides a basis for understanding the many features of the system and understanding the error messages that the verifier produces. We also hope that the tutorial and the Spec# system itself will inspire others to improve the state of the art, in the open-source distribution of Spec# as well as in other programming systems yet to be designed.

Acknowledgments

We are grateful to Rosemary Monahan for the extensive feedback on drafts of this tutorial.

References

0. 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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
1. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

3. Patrick Cousot and Rhadia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.
4. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.
5. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.
6. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.
7. Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. *SIGPLAN Not.*, 42(10):337–350, 2007.
8. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from http://www.jmlspecs.org, May 2008.
9. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, June 2004.
10. K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *LNCS*, pages 115–130. Springer, March 2006.
11. K. Rustan M. Leino and Peter Müller. Spec# tutorial web page, 2009. http://specsharp.codeplex.com/Wiki/View.aspx?title=Tutorial.
12. K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 5295 of *LNCS*, pages 192–208. Springer, 2008.
13. K. Rustan M. Leino and Angela Wallenburg. Class-local object invariants. In *First India Software Engineering Conference (ISEC 2008)*. ACM, February 2008.
14. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
15. Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3(3):243–263, 1974.
16. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.