The Spec# Programming System: An Overview

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte

Microsoft Research, Redmond, WA, USA
{mbarnett,leino,schulte}@microsoft.com

Abstract. The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. This paper describes the goals and architecture of the Spec# programming system, consisting of the object-oriented Spec# programming language, the Spec# compiler, and the Boogie static program verifier. The language includes constructs for writing specifications that capture programmer intentions about how methods and data are to be used, the compiler emits run-time checks to enforce these specifications, and the verifier can check the consistency between a program and its specifications.

0 Introduction

Software engineering involves the construction of correct and maintainable software. Techniques for reasoning about program correctness have strong roots in the late 1960's (most prominently, Floyd [25] and Hoare [33]). In the subsequent dozen years, several systems were developed to offer mechanical assistance in proving programs correct (see, *e.g.*, [37, 27, 51]). To best influence the process by which a software engineer works, one can aim to enhance the engineer's primary thinking and working tool: the programming language. Indeed, a number of programming languages have been designed especially with correctness in mind, via specification and verification, as in, for example, the pioneering languages Gypsy [1] and Euclid [38]. Other languages, perhaps most well-known among them Eiffel [54], turn embedded specifications into run-time checks, thereby dynamically checking the correctness of each program run.

Despite these visionary underpinnings and numerous victories over technical challenges, current software development practices remain costly and error prone (*cf.* [56, 52]). The most common forms of specification are informal, natural-language documentation, and standardized library interface descriptions (of relevance to this paper, the .NET Framework, see, *e.g.*, [61]). However, numerous programmer assumptions are left unspecified, which complicates program maintenance because the implicit assumptions are easily broken. Furthermore, there's generally no support for making sure that the program works under the assumptions the programmer has in mind and that the programmer has not accidentally overlooked some assumptions. We think program development would be improved if more assumptions were recorded and enforced. Realistically, this will not happen unless writing down such specifications is easy and provides not just long-term benefits but also immediate benefits.

The Spec# programming system is a new attempt at a more cost effective way to produce high-quality software. For a programming system to be adopted widely, it must provide a complete infrastructure, including libraries, tools, design support,

integrated editing capabilities, and most importantly be easily usable by many programmers. Therefore, our approach is to integrate into an existing industrial-strength platform, the .NET Framework. The Spec# programming system rests on the Spec# programming language, which is an extension of the existing object-oriented .NET programming language C#. The extensions over C# consist of specification constructs like pre- and postconditions, non-null types, and some facilities for higher-level data abstractions. In addition, we enrich C# programming constructs whenever doing so supports the Spec# programming methodology. We allow interoperability with existing .NET code and libraries, but we guarantee soundness only as long as the source comes from Spec#. The specifications also become part of program execution, where they are checked dynamically. The Spec# programming system consists not only of a language and compiler, but also an automatic program verifier, called Boogie, which checks specifications statically. The Spec# system is fully integrated into the Microsoft Visual Studio environment.

The main contributions of the Spec# programming system are

- a small extension to an already popular language,
- a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks,
- tools that enforce the methodology, ranging from easily usable dynamic checking to high-assurance automatic static verification, and
- a smooth adoption path whereby programmers can gradually start taking advantage of the benefits of specification.

In this paper, we give an overview of the Spec# programming system, its design, and the rationale behind its design. The system is currently under development.

1 The Language

The Spec# language is a superset of C#, an object-oriented language targeted for the .NET Platform. C# features single inheritance whose classes can implement multiple interfaces, object references, dynamically dispatched methods, and exceptions, to mention the features most relevant to this paper. Spec# adds to C# type support for distinguishing non-null object references from possibly-null object references, method specifications like pre- and postconditions, a discipline for managing exceptions, and support for constraining the data fields of objects. In this section, we explain these features and rationalize their design.

1.0 Non-Null Types

Many errors in modern programs manifest themselves as null-dereference errors, suggesting the importance of a programming language providing the ability to discriminate between expressions that may evaluate to null and those that are sure not to (for some experimental evidence, see [24, 22]). In fact, we would like to eradicate all null-dereference errors.

We have opted to add type support for nullity discrimination to Spec#, because we think types offer the easiest way for programmers to take advantage of nullity distinctions. Backward compatibility with C# dictates that a C# reference type T denote a possibly-null type in Spec# and that the corresponding non-null type get a new syntax, which in Spec# we have chosen to be $T!$.

The main complication in a non-null type system arises in addressing non-null fields of partially constructed objects, as illustrated in the following example:

```
class Student : Person {
  Transcript! t;
  public Student(string name, EnrollmentInfo! ei)
    : base(name) {
    t = new Transcript(ei);
  }
```

Since the field t is declared of a non-null type, the constructor needs to assign a non-null value to t. However, note that in this example, the assignment to t occurs after the call to the base-class constructor (as it must in C#). For the duration of that call, t is still null, yet the field is already accessible (for instance, if the base-class constructor makes a dynamically dispatched method call). This violates the type safety guarantees of the non-null type system.

In Spec#, this problem is solved syntactically by allowing constructors to give initializers to fields before the object being constructed becomes reachable by the program. To correct the example above, one writes:

```
class Student : Person {
  Transcript! t;
  public Student(string name, EnrollmentInfo! ei)
    : t(new Transcript(ei)),
      base(name) {
  }
```

Spec# borrows this field-initialization syntax from C++, but a crucial point is that Spec#, unlike C++, evaluates field initializers before calling the base-class constructor. Note that such an initializing expression can use the constructor's parameters, a useful feature that we deem vital to any non-null type design. Spec# requires initializers for every non-null field.

Spec# allows non-null types to be used only to specify that fields, local variables, formal parameters, and return types are non-null. Array element types cannot be non-null types, avoiding both problems with array element initialization and problems with C#'s covariant arrays.

To make the use of non-null types even more palatable for migrating C# programmers, Spec# stipulates the inference of non-nullity for local variables. This inference is performed as a dataflow analysis by the Spec# compiler.

We have settled on this simple non-null type system for three reasons. First, problems with null references are endemic in object-oriented programming; providing a solution should be very attractive to a large number of programmers. Second, our simple

solution covers a majority of useful non-null programming patterns. Third, for conditions that go beyond the expressiveness of the non-null type system, programmers can use method and class contracts, as described below.

1.1 Method Contracts

Every method (including constructors and the properties and indexers of C#) can have a specification that describes its use, outlining a contract between callers and implementations. As part of that specification, *preconditions* describe the states in which the method is allowed to be called, and hence are the caller's responsibility. *Postconditions* describe the states in which the method is allowed to return. The *throws set* and its associated *exceptional postconditions* limit which exceptions can be thrown by the method and for each such exception, describe the resulting state. Finally, *frame conditions* limit the parts of the program state that the method is allowed to modify. The postconditions, throws set, exceptional postconditions, and frame conditions are the implementation's responsibility. Method contracts establish responsibilities, from which one can assign blame in case of a contract-violation error.

Uniform error handling in modern programming languages is often provided by an *exception* mechanism. Because the exception mechanisms in C# and the .NET Framework are rather unconstrained, Spec# adds support for a more disciplined use of exceptions to improve the understandability and maintenance of programs. As a prelude to explaining method contracts, we describe the Spec# view of exceptions.

Exceptions Spec# categorizes exceptions according to the conditions they signal. Looking at exceptions as pertaining to particular methods, Goodenough [28] categorizes exceptions into two kinds of failures, which we call *client failures* and *provider failures*. A client failure occurs when a method is invoked under an illegal condition, that is, when the method's precondition is not satisfied. We further refine provider failures into *admissible failures* and *observed program errors*. An admissible failure occurs when a method is not able to complete its intended operation, either at all (*e.g.*, the parity of a received network packet is wrong) or after some amount of effort (*e.g.*, after waiting on input from a network socket for some amount of time). The set of admissible failures is part of the contract between callers and implementations. An observed program error is either an intrinsic error in the program (*e.g.*, an array bounds error) or a global failure that's not particularly tied with the method (*e.g.*, an out-of-memory error).

An important consideration among these kinds of exceptions is whether or not one expects a program ever to catch the exception. Admissible failures are part of a program's intended possible behaviors, so we expect correct programs to catch and handle admissible failures. In contrast, correct programs never exhibit client failures or observed program errors, and it's not even clear how a program is to react to such errors. If the program handles such failures at all, it would be at the outermost tier of the application or thread.

Because of these considerations, Spec# follows Java [29] by letting programmers declare classes of exceptions as either *checked* or *unchecked*. Admissible failures are signaled with checked exceptions, whereas client failures and observed program errors are signaled using unchecked exceptions.

ArrayList.Insert Method (*Int32*, *Object*)

Inserts an element into the *ArrayList* at the specified index.

public virtual void *Insert*(**int** *index*, **object** *value*);

Parameters

- *index* The zero-based index at which *value* should be inserted.
- *value* The *Object* to insert. The *value* can be a null reference.

Exceptions

Exception Type	Condition
ArgumentOutOfRangeException	*index* is less than zero. *–or–* *index* is greater than *Count*.
NotSupportedException	The *ArrayList* is read-only. *–or–* The *ArrayList* has a fixed size.

Fig. 0. The .NET Framework documentation for the method *ArrayList.Insert*.

In Spec#, any exception class that implements the interface *ICheckedException* is considered a checked exception. For more information about the exception design in Spec#, see our companion paper on exception safety [48].

Preconditions Perhaps the most important programmer assumption is the precondition. Here is a simple example of a method with a precondition:

```
class ArrayList {
  public virtual void Insert(int index, object value)
    requires 0 <= index && index <= Count;
    requires !IsReadOnly && !IsFixedSize;
  { ... }
```

The precondition specifies that the index into which the object is to be inserted in the array list must be within bounds, and that the list can grow. To enforce these preconditions, the Spec# compiler emits run-time checks that throw a *RequiresViolationException*, indicating a client failure, if a precondition is not met. If the user invokes Boogie on a call site, then Boogie attempts to verify statically that these preconditions hold at the call site, reporting an error if it cannot.

The .NET Framework documentation for this method is shown in Figure 0. There is a subtle difference between the .NET documentation for *Insert* and our specification of it above. Both specifications state what's expected of the caller; the difference lies in the action taken in the event that preconditions are violated. To support this typical robust-programming style of .NET Framework specifications, Spec#'s preconditions can have **otherwise** clauses. These can be used to tell the compiler to use a specified exception,

rather than the default $RequiresViolationException$, in the event that a precondition violation is detected at run time:

```
class ArrayList {
  void Insert(int index, object value)
    requires 0 <= index && index <= Count
      otherwise ArgumentOutOfRangeException;
    requires !IsReadOnly && !IsFixedSize
      otherwise NotSupportedException;
  { ... }
```

Since it represents a client failure, the exception used in an **otherwise** clause must be an unchecked exception.

Postconditions Method specifications can also include postconditions. For example, one can specify the postconditions of $Insert$ as follows:

```
ensures Count == old(Count) + 1;
ensures value == this[index];
ensures Forall{int i in 0 : index;  old(this[i]) == this[i]};
ensures Forall{int i in index : old(Count);  old(this[i]) == this[i + 1]};
```

These postconditions say that the effect of $Insert$ is to increase $Count$ by 1, to insert the given value at the given index, and to keep all other elements in their same relative positions. This example also shows some other Spec# specification features: In the first line, $\mathbf{old}(Count)$ denotes the value of $Count$ on entry to the method. In the third line, the special function $Forall$ is applied to the comprehension of the boolean expression $\mathbf{old}(\mathbf{this}[i]) == \mathbf{this}[i]$, where i ranges over the integer values in the half-open interval from 0 to less than $index$. Comprehensions and quantifiers are syntactically restricted in such a way that the compiler can always generate code that computes them.

Boogie attempts to verify each implementation of $Insert$ against these postconditions. When Boogie's verification is successful, then the run-time checks (which would throw an $EnsuresViolationException$ in this case) are not needed since they would never fail.

For run-time checking, we have adopted Eiffel's mechanism for evaluating $\mathbf{old}(E)$. On entry to a method, the expression E of any $\mathbf{old}(E)$ occurring in a postcondition is evaluated and the resulting value is saved away. Then, whenever (and if) this value of $\mathbf{old}(E)$ is needed during the evaluation of the postcondition, the saved value of E is used. Note that the value of $\mathbf{old}(E)$ may in fact not be needed during the evaluation of the postcondition due to short-circuit boolean expressions or because the method does not terminate normally.

The example above also illustrates a more general point about the differences between checking contracts statically and dynamically. Boogie has knowledge about the program and its built-in data structures. It also has support for quantifiers and can therefore check the postconditions of $Insert$ statically. Contracts that use procedural abstraction, however, can be a problem for static modular checking, since such checking has access only to a limited part of the program. Likewise, contracts that use higher-level data structures can be a problem for static checking, because of limitations of

decisions procedures and axiomatizations of some theories. Here, dynamic checking is straightforward. On the other hand, the dynamic checking of postconditions can be quite involved when **old** expressions mention quantified variables, as exemplified above.

Though we expect the bulk of specifications to be simple, the more general point is that Spec# supports expressive specifications even when those specifications push the limits of today's checking technology.

Exceptional postconditions As in Java, each method whose invocation may result in a checked exception must account for that exception in the method's throws set. For example, the declaration

```
char Read()
  throws SocketClosedException;
{ ... }
```

where *SocketClosedException* is a checked exception class, allows the method to throw any checked exception whose allocated type is a subclass of *SocketClosedException*, but is not allowed to throw any other checked exception. The Spec# compiler holds every implementation to its throws set by a conservative control-flow analysis. A **throws** clause in Spec# can only mention checked exceptions.

Spec# allows a **throws** declaration to be combined with a postcondition that takes effect in the event that the exception is thrown. For example, the exceptional postcondition in

```
void ReadToken(ArrayList a)
  throws EndOfFileException ensures a.Count == old(a.Count);
{ ... }
```

says that the length of *a* is unchanged in the event that the method results in an *EndOfFileException*.

Without further restrictions, it would be possible for a program to foil the compiler's throws-set analysis, which would then undermine Spec#'s guarantee that every checked exception is accounted for. Consider the following example:

```
void ExceptionScam() {
  Exception e = new MyCheckedException();
  throw e;
}
```

The root of the exception class hierarchy, *Exception*, is an unchecked exception (because it comes from C#, where all exceptions are unchecked). Since checked exceptions are subtypes of *Exception*, the **throw** statement in *ExceptionScam* would have the effect of throwing a checked exception even though the method does not advertise it. Spec# prevents this: whenever the static type of a thrown expression is an unchecked exception and the static analysis cannot guarantee that the dynamic type is likewise unchecked, then the compiler inserts a run-time check that detects any violation of Spec#'s distinction between checked and unchecked exceptions.

For more information about exceptions in Spec#, see our companion paper on exception safety [48].

Frame conditions Spec# method contracts also include **modifies** clauses (also known as *frame conditions*), which restrict which pieces of the program state a method implementation is allowed to modify. For example, in the class

```
class C {
  int x, y;
  void M() modifies x; { ... }
```

method M is permitted to have a net effect on the value of x, whereas the value of y on exit from the method must have the same value as on entry.

Any realistic design of **modifies** clauses includes some facility for abstracting over program state that for reasons of information hiding cannot be mentioned in the method contract. For example, the implementation of $ArrayList.Insert$ is going to modify the private representation of the $ArrayList$, but private variables are not allowed to be mentioned explicitly in the contract of a public method. Instead, a wildcard can be used. For example, the specification

```
modifies this^ArrayList;
```

allows any field of **this** declared in class $ArrayList$ to be modified. Spec# also supports other flavors of wildcards (see [3]), which additionally address the problem of specifying the modification of state in subclasses (*cf.* [41]).

But wildcards are still just a partial solution to the frame problem, because they don't extend to aggregate objects. For example, the $ArrayList$ implementation consists of an array and a count. The **modifies** clause above allows the count and the reference to the array to be changed, but does not give explicit permission to modify the array elements. To deal with aggregate objects, Spec# uses a concept of *ownership*. We say that the $ArrayList$ owns its underlying array, that the array is *committed* to the $ArrayList$. Modifications to the state of committed objects do not need to be mentioned explicitly in the **modifies** clause. For more details, see [3], which also describes the connection between ownership and object invariants.

Frame conditions serve as documentation and are used and enforced by Boogie, but they are currently not enforced at run time. There are two reasons for not checking **modifies** clauses at run time. First, they can be prohibitively expensive, since the checking must compare arbitrarily large portions of the heap in a method's pre-state and post-state. Second, we are aiming for a smooth transition to Spec# from C#; we do not want to incur run-time errors in C# programs that otherwise are correct.

Inheritance of specifications In Spec#, a method's contract is inherited by the method's overrides. The run-time checks evoked by the method contract are thus also inherited. Not only does this make the specifications more definitive and reliable than today's documentation, but the Spec# specifications also make the code of an implementation easier to read, since today's manually written code for checking preconditions can be rather lengthy.

A method override can add more postconditions by declaring additional **ensures** clauses. The override can add exceptional postconditions only for those exceptions that are already covered by the throws set. The override is not allowed to give any **modifies**

clause: enlarging the frame would be unsound, and shrinking the frame can be done with an added postcondition. Spec# does not allow any changes in the precondition, because callers expect the specification at the static resolution of the method to agree with the dynamic checking.

Methods declared in an interface can have specifications, just like the methods declared in a class. Interfaces give rise to a form of multiple inheritance, because a class can inherit a method signature from the superclass and its implemented interfaces. Traditionally, these inherited specifications are combined [62], which is what Spec# does for postconditions. Spec# also combines exceptional postconditions, but the inherited specifications must have identical throws sets. If a class implements an interface method, then the interface declaration of the method must have a frame condition that is a superset of the class implementation of the method. Spec# does not combine preconditions, unless they are the same, for the reason explained above. Since the obvious definitions of "the same" are either syntactic and brittle, or semantic and require theorem proving, Spec# uses the radical solution of allowing multiple inherited specifications only when these have no **requires** clauses.

We give an example that shows Spec#'s radical precondition solution not to be too draconian. Consider the following interfaces:

$$\textbf{interface}\ I\ \{\ \textbf{void}\ M(\textbf{int}\ x)\ \textbf{requires}\ x <= 10;\ \}$$
$$\textbf{interface}\ J\ \{\ \textbf{void}\ M(\textbf{int}\ x)\ \textbf{requires}\ x >= 10;\ \}$$

Suppose a class C wants to implement both interfaces I and J. In this case, Spec# does not allow C to provide one shared implementation for $I.M$ and $J.M$. Instead, class C needs to give explicit interface method implementations for M:

$$\textbf{class}\ C : I, J\ \{$$
$$\quad \textbf{void}\ I.M(\textbf{int}\ x)\ \{\ \ldots\ \}$$
$$\quad \textbf{void}\ J.M(\textbf{int}\ x)\ \{\ \ldots\ \}$$

(Explicit interface method implementations are a feature of C#.) Because an explicit interface method implementation cannot be accessed other than through the interface, it gets its contract straight from the interface.

Taken together, the Spec# rules for contract inheritance guarantee that a derived specification always properly obeys the behavioral subtyping rules [21, 23].

1.2 Class Contracts

Specifying the rules for using a library or abstraction is done primarily through method contracts, which spell out what's expected of the caller and what the caller can expect in return from the implementation. To specify the design of an implementation, one primarily uses specifications that constrain the value space of the implementation's data. These specifications are called *object invariants* and spell out what is expected to hold of each object's data fields in the steady state of the object. For example, the class fragment

$$\textbf{class}\ \mathit{AttendanceRecord}\ \{$$
$$\quad \mathit{Student}[\,]!\ \mathit{students};$$
$$\quad \textbf{bool}[\,]!\ \mathit{absent};$$
$$\quad \textbf{invariant}\ \mathit{students.Length} == \mathit{absent.Length};$$

declares that the lengths of the arrays $students$ and $absent$ are to be the same.

As we can see from the simple example above, it is not possible for an object invariant always to hold, because it is not possible in the language to change the lengths of two arrays simultaneously. This is why we say the object invariant holds in *steady states*, which essentially means that the object is not currently being operated on. Following our methodology for object invariants [3, 45, 6], Spec# makes explicit when an object is in its steady state versus when it is *exposed*, which means the object is vulnerable to modifications. Spec# introduces a block statement **expose** that explicitly indicates when an object's invariant may temporarily be broken: the statement

$$\begin{array}{l}\mathbf{expose}\ (o)\ \{\\ \quad S;\\ \}\end{array}$$

exposes the object o for the duration of the sub-statement S, which may then operate on the fields of o. Because field modifications in an object-oriented program tend to be encapsulated in the class that declares the field, the expression o is usually **this**. The object invariant is supposed to hold again at the end of the **expose** statement and Spec# enforces this with a run-time check. Object invariants are also checked at the end of constructors (though there's some flexibility that allows the initial check of an object invariant to be performed elsewhere; we omit the details here).

By default, whenever a class or any of its superclasses has a declared invariant, every public method of the class has an implicit

$$\mathbf{expose}\ (\mathbf{this})\ \{\ \ldots\ \}$$

around the method body. Our preliminary experience suggests that this default removes most of the need for explicit **expose** statements. In situations where reentrancy is desired, the default can be disabled by a custom attribute on the method.

Exposing an object is not idempotent. That is, it is a checked run-time error if $\mathbf{expose}\ (o)\ \ldots$ is reached when o is already exposed. In this way, the expose mechanism is similar to thread-non-reentrant mutexes in concurrent programming, where monitor invariants [34] are the analog of our object invariants. If exposing were idempotent, then one would not be able to rely on the object invariant to hold immediately inside an expose block, in the same way that the idempotence of thread-reentrant mutexes means that one cannot rely on the monitor invariant to hold at the time the mutex is acquired.

For Spec#'s object-invariant methodology to be sound, all modifications of a field $o.f$ must take place while the object o is exposed. Furthermore, the methodology uses an ownership relation to structure objects into a tree-shaped hierarchy. The relation is state dependent, which allows ownership transfer. Such modifications and ownerships are enforced by Boogie, but are not enforced at run time.

Object invariants can be declared in any class. To support modular checking of invariants, so that a class does not need to know the invariants of its superclasses and future subclasses, object invariants are partitioned into *class frames* according to the class that declares each invariant [3, 17]. The **expose** mechanism deals with class frames.

To reduce the programmer's initial cost of adding **expose** statements and to handle non-virtual methods in a more backward compatible way (see [3]), Spec# allows one **expose** statement to expose more than one class frame. To explain this feature, we first need to show the more general form of the **expose** statement in Spec#, which is

expose (o **upto** T) { ... }

where T is a superclass of the static type of the expression o. If "**upto** T" is omitted, T defaults to the static type of expression o. More precisely than we described it above, the statement exposes all of o's class frames from above its currently exposed class frame through T (also exposing the class frame T itself). Non-idempotence requires that at least one class frame is exposed as part of the operation. At the end of the **expose** block, the class frames that were exposed on entry are un-exposed, and the object invariant for each of those class frames is checked. This is done at run time using compiler-emitted dynamically dispatched methods that check the invariants.

Exposing an unknown number of class frames, and in particular checking the invariants for class frames whose declarations may not be in scope, poses a problem for modular, static verification. Therefore, we use a stricter model for **expose** in Boogie. In particular, whereas the precondition for

expose (o **upto** T) { ... }

as enforced by run-time checks is that o's T class frame is un-exposed—that is, that the o's most-derived un-exposed class frame is a subclass of T—Boogie strengthens this precondition by requiring o's most-derived un-exposed class frame to be exactly T. This way, Boogie is able to find all the object invariants that it needs to check at the end of the **expose** block. In effect, this difference in policy between the run-time behavior and what's enforced by Boogie means that programmers can start writing and running Spec# programs more easily, but then may need to exert additional effort in order to obtain the higher confidence in the program's correctness assured by Boogie (just as additional effort is required to make sure Boogie's modification and ownership rules are satisfied).

Object invariants are allowed to mention only constants, fields, array elements, state independent methods, and confined methods. A method is state independent if it does not depend on mutable state. A confined method may depend on the state of owned objects. The Spec# compiler includes a conservative effect analysis to check that these properties are obeyed.

Spec# also supports class invariants, which are useful to document assumptions about static fields. Methodology and constraints for class invariants are similar to those for object invariants, except that there is no inheritance [44]. The **expose** statement simply takes a class instead of an object as a parameter.

1.3 Other Details

Exceptions within contracts If an exception is thrown during the evaluation of a contract in Spec#, then the exception is wrapped in a contract evaluation exception and propagated. This is in contrast to the run-time evaluation of contracts in JML, where such exceptions are caught and the surrounding formula is treated as if it returned a boolean value according to certain rules, see [14].

Custom attributes on specifications C# provides *custom attributes* as a way to attach arbitrary data to program structures, such as classes, methods, and fields. A custom attribute is compiled into metadata whose standard format allows various applications to read the custom attributes attached to a particular declaration. Spec# also allows each specification clause to be annotated with custom attributes.

Custom attributes allow users of third-party tools to mark up specifications in tool-specific ways. For instance, the Spec# compiler uses the $Conditional$ custom attribute to control which specifications are emitted as run-time checks in the current build. For example, for the following method

$$\begin{array}{l}\mathbf{int}\ BinarySearch(\mathbf{object}[]!\ a, \mathbf{object}\ o, \mathbf{int}\ lo, \mathbf{int}\ hi)\\ \quad \mathbf{requires}\ 0 <= lo\ \&\&\ lo <= hi\ \&\&\ hi <= a.Length;\\ \quad [Conditional(\text{“}DEBUG\text{”})]\ \mathbf{requires}\ IsSorted(a, lo, hi);\\ \{\ \ldots\ \}\end{array}$$

the compiler emits run-time checks for both preconditions in the debug build, but emits a check only for the first precondition in the non-debug build. This supports the common programming style of debugging assertions (see, ***e.g.***, [53]).

Purity We want to have the property that a program that runs correctly with all contract checking enabled also runs correctly if some of the contract checking is disabled. Therefore, we require all expressions appearing in contracts to be *pure*, meaning that they have no side effects and do not throw any checked exceptions. The compiler enforces this condition using a conservative effect system. We are considering more liberal definitions of purity, such as observational purity [7] and that afforded by the heap analysis of Sălcianu and Rinard [58].

2 System Architecture

Architecturally, the Spec# programming system consists of the compiler, a runtime library, and the Boogie verifier. The compiler has been fully integrated into the Microsoft Visual Studio environment in terms of the project system, build process, design tools, syntax highlighting, and the IntelliSense context-sensitive editing and documentation assistance.

The Spec# compiler differs from an ordinary compiler in that it does not only produce executable code from a program written in the Spec# language, but also preserves all specifications into a language-independent format. Having the specifications available as a separate, compiled unit means program analysis and verification tools can consume the specifications without the need to either modify the Spec# compiler or to write a new source-language compiler.

The Spec# compiler can preserve the specifications in the same binary with the compiled code because it targets the Microsoft .NET Common Language Runtime (CLR) [10]. The CLR provides rich metadata facilities for associating many types of information with most elements of the type system (types, methods, fields, *etc.*). The Spec# compiler attaches a specification to each program component for which a specification

exists. (Technically, the specifications are preserved as strings in custom attributes. All names are fully resolved; while this renders the format quite verbose, it makes it much easier for any tools consuming it.)

As a result, we made the design decision to have Boogie consume compiled code, rather than source code. An additional benefit is that Boogie can be used to verify code written in other languages than Spec#, as long as there is an *out-of-band process* for attaching contracts to such code. We use such a process to attach specifications to the .NET Framework Base Class Library (BCL), see Section 2.2.

2.0 Run-time Checking

Spec# preconditions and postconditions are turned into inlined code. We do this not only for performance reasons, but also to avoid creating extra methods and fields in the compiled code. All such inlined code is tagged so that code corresponding to the Spec# contracts can be differentiated from the code that comes from the rest of the Spec# program. Such separation is required by any analysis tool that consumes Spec# contracts from the metadata. For instance, Boogie must be able to determine if the non-contract code in a method meets its postcondition, rather than the combination of the non-contract code followed by the code that checks the postcondition. The inlined code evaluates the conditions and, if violated, throws an appropriate contract exception.

To check object invariants, the compiler adds a new method to each class that declares an invariant. Special object fields, such as the invariant level [3] and owner of an object [45], are added to the super-most class that uses Spec# features within each subtree of the class hierarchy. As we mentioned in Section 1, the runtime does not enforce the whole methodology; for instance, run-time checking does not check that an object is exposed before updating a field. This means that an error may go undetected at run time that would be caught by Boogie.

2.1 Static Verification

From the intermediate language (including the metadata), Spec#'s static program verifier, Boogie, constructs a program in its own intermediate language, BoogiePL. BoogiePL is a simple language with procedures whose implementations are basic blocks consisting mostly of four kinds of statements: assignments, asserts, assumes, and procedure calls (*cf.* [47]).

An inference system processes the BoogiePL program using interprocedural abstract interpretation [15, 57] to obtain properties such as loop invariants. Any derived properties are added to the program as assert statements or assume statements. The BoogiePL program then goes through several transformations, ending as a verification condition that is fed to an automatic theorem prover. The transformations, such as cutting all loops to derive an acyclic control flow graph by introducing *havoc* statements, are done in a way that preserves the soundness of the analysis. A havoc statement assigns an arbitrary value to a variable; introducing havoc statements for all variables assigned to in a loop causes the theorem prover to consider an arbitrary loop iteration. All feedback from the theorem prover is mapped back onto the source program before it is delivered to the user [43]. The result is that programmers interact with Boogie's

prover only by making changes at the program source level, for instance by adding contracts.

Currently, Boogie uses the Simplify theorem prover [18], but we intend to switch to a new experimental theorem prover being developed at Microsoft Research.

2.2 Out-of-band Specifications and Other Goodies

All .NET applications use the Base Class Library (BCL) in one form or the other. Thus we want to provide specifications for the entire BCL. This gives any client an immediate benefit even before writing a single contract.

But this raises a problem: how to provide a mechanism for attaching Spec# contracts to code that was written without them? (Note that we cannot modify the BCL even if we would use its implementation, since doing so would break versioning.) *Out-of-band* specifications, that is, specifications for code external to Spec#, are compiled into a Spec# repository. The repository is consulted in case the Spec# compiler or Boogie encounters a method or class for which it requires a specification (*i.e.*, when the compiler emits run-time checks or when Boogie generates verification conditions), but the method or class in the original code does not have an attached specification.

Writing contracts for self-contained examples is easy, but realistic programming is highly dependent on libraries, such as the BCL. A large obstacle then is obtaining contracts for the existing libraries. A companion project is working on semi-automatically generating contracts for existing code. It has automatically extracted almost three thousand preconditions for the current version of the BCL.

We have plans to build an explainer that translates Spec# method contracts into natural-language documentation entries. For example, it seems that one could translate preconditions and throws sets into the stylized exception tables used in the .NET documentation, see Figure 0. This could better keep the documentation accurate and up-to-date.

Lastly, we are planning a tool for translating Spec# into plain C#. (There are still some problems, like figuring out what to do with field initializers, that we need to address.) This tool will allow the use of Spec# within the normal development process. For instance, most Microsoft development groups insist on building their products using only official Microsoft compilers. In this context, Spec# would function as a precompiler; however, it is this invisibility that is important to gaining acceptance in a rigorous build environment.

3 Related Work

A number of programming languages have been designed especially with correctness or verification in mind. These include the pioneering languages Gypsy [1], Alphard [63], Euclid [38], and CLU [49], which offered different degrees of formality. In Gypsy, which was the first language to include specifications as an integral part of the programming language, the specifications integrated in the source program were aimed directly at program verification via an interactive theorem prover. Alphard was designed around a programming methodology for designing and proving object-like data structures, but

the proofs were done by hand. In Euclid, specifications written in the programming language's boolean expressions were checked at run time, with the idea that more complicated specifications, which were supplied in comments, would be used by some external program-verification tool. The CLU programming methodology prominently included specifications, but these were recorded only as stylized comments.

Three modern systems with contracts that have had a direct effect on practical programs are Eiffel [54], SPARK [2], and B [0].

Eiffel [54] is an object-oriented language with almost 20 years of use. The standard library is well documented through contracts, so contracts fall prominently within the purview of programmers. The contracts are enforced dynamically. However, without a full methodology for **modifies** clauses and for object invariants in the presence of callbacks, it would not be possible to obtain modular static verification.

SPARK [2] is a limited subset of Ada, without many dynamic language features like references, memory allocation, and subclassing, yet large enough to be useful for many embedded applications. Praxis Critical Systems has used SPARK in the development of several industrial programs, and their measurements indicate that the rigor provided by SPARK can be cost effective [13]. SPARK offers a selection of static tools, from light-weight sanity checking to full verification with an interactive theorem prover. Compatibility with an existing language has been a high priority in the design of SPARK, just like for Spec#, but their approach is quite different from ours. By ruling out difficult features of Ada, SPARK achieves the property that any SPARK program can be compiled by any standard Ada compiler while retaining its SPARK meaning (all SPARK specifications are placed in stylized Ada comments, and thus they are not used by the compiler). To meet our goal of migrating normally skilled programmers to a higher-integrity language, we have been unable to follow SPARK's approach of designing a subset of an existing language. Instead, we have designed Spec# to be a superset of an existing language, aiming to support easy and gradual adoption of its new features.

The B approach [0] uses a different methodology for writing programs: starting from full specifications and supporting a machine-aided process for stepwise refining the specifications into compilable programs. The resulting programs are similar in expressiveness to SPARK programs. This methodology, which has been used with success for example in constructing the Paris Metro braking system software, produces only correct programs. However, the skills needed to go through the refinement process make for a steep learning curve for the system and become a barrier for many programmers. It is also not obvious how to extend the methodology to more expressive abstractions, like those in object-oriented programs today.

The Java Modeling Language (JML) [39, 40] is a notation for writing specifications for Java programs. JML specifications, which include rich flavors of method contracts, are recorded in Java source code as stylized comments. An impressive array of tools have been build around JML, including tools for documentation, run-time checking, unit testing, light-weight contract checking, and program verification [12]. Spec# provides a more focused methodology than JML, which for example has yet to adopt a full story for object invariants in the presence of callbacks. The design space of Spec# is somewhat less constrained than JML, since JML does not seek to alter the underlying

programming language (which, for example, has let Spec# introduce field initializers and **expose** blocks).

The language AsmL [32] has many of the same aspirations as Spec#: to be an accessible, widely-used specification language tailored for object-oriented .NET systems. However, AsmL is oriented toward supporting model-based development with its facilities for model programs, test-case generation, and meta-level state exploration [5]. Our experiences in using AsmL for interface specification [8], run-time verification [9], and an on-going project with a product group [4] contributed to the design of Spec#. The companion testing tool SpecExplorer [30], currently in use within Microsoft, uses the Spec# language to provide model-based testing with features for test-case generation, explicit-state model checking, and run-time conformance checking.

The Anna [50] specification language for Ada lets programmers write down important design decisions. The specifications are compiled into run-time checks.

The first mechanical systems for proving programs correct were conceived and built several decades ago. These include the early, but not entirely automatic, systems of King [37, 36] and Deutsch [20], Gypsy [27], and the Stanford Pascal Verifier [51]. More recent program verifiers include Penelope (for Ada) [31] and LOOP (for Java and JML) [60], both of which require interactive theorem proving.

Setting early efforts by Sites [59] and German [26] into full motion, the Extended Static Checker for Modula-3 (ESC/Modula-3) [19] changed the rules of the game by leveraging the power of an automatic theorem prover not for proving the full functional correctness of programs, but for the limited aim of finding common errors in programs. Continuing in that tradition, ESC/Java [24] wrapped that technology with a simpler contract language (a subset of JML), aiming to deliver a practical high-precision tool for normally skilled programmers. A key ingredient that enables these ESC tools to do useful checking is the willingness to miss certain errors, since that can lead to a simpler specification language and to better odds for the automatic theorem prover to succeed (see also [42]). Boogie attempts to completely verify a program without missing errors; its ability to do so is bound to depend on the simplicity of the specifications.

Spec# provides a limited type system for non-null types. A more comprehensive type-system solution has been proposed by Fähndrich and Leino [22]. Their design deals with the complication of non-null fields by introducing additional *raw* types for partially-constructed objects.

Various abstraction facilities that help define **modifies** clauses in modern object-oriented languages have been proposed (*e.g.*, [46, 55, 41]).

Our methodology for object invariants and **modifies** clauses relies on object ownership to impose a structure on the heap [3, 45, 6]. Similar effects have been achieved by ownership types and other alias-confinement strategies (*e.g.*, [16, 11]). The earliest such use we've seen dates back to Alphard [63], where the modifier **unique** specifies that a field points to an owned object.

4 Concluding Remarks

The foundation of the Spec# programming system is the Spec# programming methodology, the Spec# language, the Spec# compiler, and the Boogie static program verifier.

The methodology prescribes for the first time how to deal soundly with object invariants and subclasses in a modular setting. The Spec# language embodies the methodology: Spec# enriches C# with non-null types, contracts, checked exceptions, comprehensions, and quantifications. The Spec# compiler uses a combination of static-analysis techniques and run-time checks to guarantee soundness of the language. The verifier tries to check the consistency between a program and its specifications.

We are trying to make the Spec# system a practically useful software tool that enables normally skilled programmers to write down and verify their assumptions. Therefore, we start from a familiar programming language and use the metaphor of type checking for exposing the new capabilities of our static checking technology. We do not offer a way to axiomatize new mathematical theories. Rather our design focus is on limited, partial functional specifications, those that can be written using boolean expressions of the language and quantifiers.

We have designed Spec# to provide incremental benefit as programmers use more of its features. Even without writing their own specifications, programmers get immediate benefit as their Spec# code is checked against the partially specified Base Class Library. Programmers gradually receive more benefit as they add, for example, non-null types and preconditions to their code.

Our design of Spec# has focused on sequential programs, but we are already extending our methodology to styles of concurrent programs [35]. It seems plausible that Spec# could also be of direct help in building secure applications. It would be interesting to explore the combination of our methodology with the stack walking mechanism of code access security in the context of existing libraries for permissions, authentication, and cryptography.

Acknowledgments

Many colleagues have helped make Spec# what it is: Colin Campbell, Rob DeLine, Manuel Fähndrich, Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Peter Müller and David Naumann have contributed to the more advanced versions of our methodology for object invariants. Rob and Manuel are also members of the Boogie project. Jim Larus and Sriram Rajamani have provided support and helpful discussions. Craig Schertz provided the tool for extracting contracts from existing code. A big thanks goes to Herman Venter, who has been invaluable in the implementation of the Spec# programming language and development environment. Finally, we thank Gary Leavens, Peter Müller, and others for their useful comments on a draft of this paper.

References

0. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
1. Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, March 1977.

2. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003. With Praxis Critical Systems Limited.
3. 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.
4. Mike Barnett, Wolfgang Grieskamp, Clemens Kerer, Wolfram Schulte, Clemens Szyperski, Nikolai Tillmann, and Arthur Watson. Serious specification for composing components. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, May 2003.
5. Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, October 2003.
6. Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 54–84. Springer, July 2004.
7. Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In Erik Poll, editor, *Proceedings of the ECOOP Workshop FTfJP 2004, Formal Techniques for Java-like Programs*, pages 11–19, June 2004. University of Nijmegen, NIII report NIII-R0426.
8. Mike Barnett and Wolfram Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, November 2001.
9. Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, 2003.
10. Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2002.
11. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.
12. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.
13. Roderick Chapman. Industrial experience with SPARK. Presented at SIGAda'00, November 2000. Available from http://www.praxis-cs.co.uk.
14. Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003. Iowa State University, Department of Computer Science, Technical Report TR #03-09.
15. 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.
16. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.
17. Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, June 2004.

18. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
19. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
20. L. Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California, Berkeley, 1973.
21. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258–267. IEEE, 1996.
22. Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003*, volume 38, number 11 in *SIGPLAN Notices*, pages 302–312. ACM, November 2003.
23. Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, volume 36, number 11 in *SIGPLAN Notices*, pages 1–15. ACM, November 2001.
24. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
25. Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
26. Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1978.
27. Donald I. Good, Ralph L. London, and W. W. Bledsoe. An interactive program verification system. In *Proceedings of the international conference on Reliable software*, pages 482–492. ACM, 1975.
28. John B. Goodenough. Structured exception handling. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 204–224. ACM, January 1975.
29. James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.
30. Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Instrumenting scenarios in a model-driven development environment. Submitted manuscript, 2004.
31. David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.
32. Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. To appear.
33. C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12:576–580,583, 1969.
34. C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
35. Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In *Proceedings of the workshop on Specification and Verification of Component-Based Systems*, 2004. To appear.
36. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

37. James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, September 1969.
38. Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, October 1981. An earlier version of this report appeared as volume 12, number 2 in *SIGPLAN Notices*. ACM, February 1977.
39. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
40. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003.
41. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.
42. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175. Springer, January 2001.
43. K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 2004. To appear.
44. K. Rustan M. Leino and Peter Müller. Modular verification of global module invariants in object-oriented programs. Technical Report 459, ETH Zürich, September 2004.
45. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, June 2004.
46. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
47. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999.
48. K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM 2004—Second International Conference on Software Engineering and Formal Methods*, pages 218–227. IEEE, September 2004.
49. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
50. D. C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
51. D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.
52. Charles C. Mann. Why software is so bad. *MIT Technology Review*, July/August 2002.
53. Steve McConnell. *Code complete: A practical handbook of software construction*. Microsoft Press, 1993.
54. Bertrand Meyer. *Object-oriented software construction*. Series in Computer Science. Prentice-Hall International, 1988.

55. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.
56. RTI Health, Social, and Economic Research. The economic impact of inadequate infrastructure for software testing. RTI Project 7007.011, National Institute for Standards and Technology, May 2002.
57. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
58. Alexandru Sălcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, MIT, May 2004.
59. Richard L. Sites. *Proving that Computer Programs Terminate Cleanly*. PhD thesis, Stanford University, May 1974. Technical Report STAN-CS-74-418.
60. Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, April 2001.
61. Mickey Williams. *Microsoft Visual C# .NET*. Microsoft Press, 2002.
62. Jeannette Marie Wing. *A two-tiered approach to specifying programs*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, May 1983. MIT Laboratory for Computer Science TR-299.
63. William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-2(4):253–265, December 1976.