DOI:10.1145/1953122.1953145

Can a programming language really help programmers write better programs?

BY MIKE BARNETT, MANUEL FÄHNDRICH, K. RUSTAN M. LEINO, PETER MÜLLER, WOLFRAM SCHULTE, AND HERMAN VENTER

# Specification and Verification: The Spec# Experience

A MAIN REASON for the difficulty of progamming is the inability of programmers to ensure their programs behave as intended. Needed is both a way to record that intent and the tools to enforce it. Spec# (pronounced "speck sharp")[a] is a project from Microsoft Research aimed at addressing the problem in the context of modern object-oriented languages, using the well-known approach of contracts, or specification constructs, to document behavior.[22] Contracts standardize the common practice of writing assertions within code through two main constructs:

*Method pre- and postconditions.* Method pre- and postconditions are part of the application- programming interface (API) for methods. Preconditions describe what is to be true at method entry, callers establish them,

a http://specsharp.codeplex.com

**key insights**

- **Programmers would provide more information about their code than existing programming languages allow, if only they received some benefit from doing so.**
- **Object-oriented languages should use a type system that distinguishes references that can never be null; programmers spend far too much of their time having to compensate for this limitation.**
- **Program verification systems receive a tremendous benefit by leveraging a shared infrastructure and a higher-level way of communicating with automatic theorem provers.**

and implementers can assume them. Postconditions describe what is to be true at method exit; implementers establish them, and callers can assume them upon method return.

*Object invariants.* Object invariants are a way to specify the steady-state properties that all "good" instances of a class should maintain. A crucial feature setting Spec# apart from previous programming systems is a sound technique for reasoning about when object invariants hold; why this is such a difficult problem and how Spec# solves it are covered in the section on invariants.

Spec# enforces both kinds of contracts with instrumentation for run-time checking and with an automatic program verifier for static, compile-time checking.

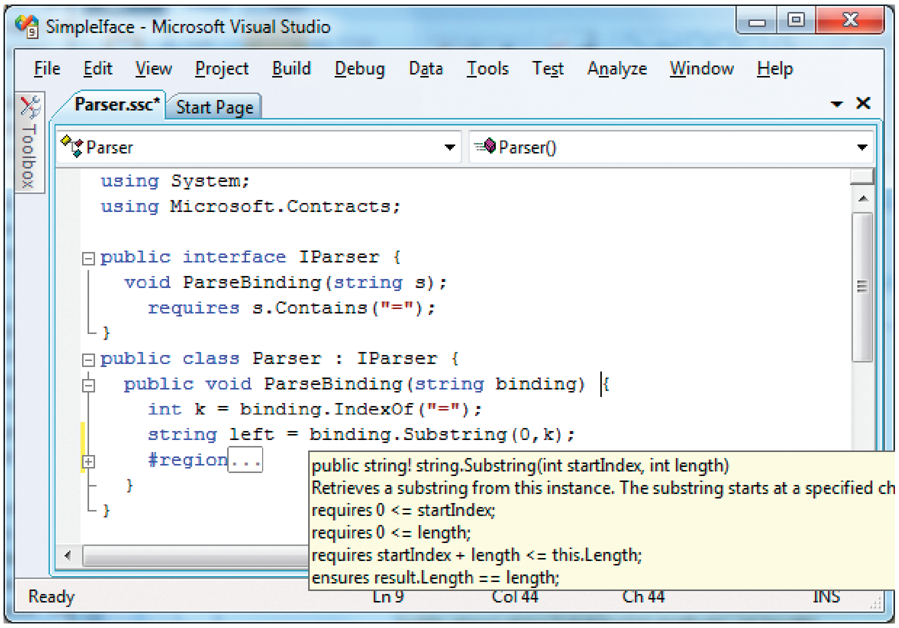
Programmers interact with Spec# just as they do with any other programming system: Type in the program, and respond to errors. The difference is that in Spec#, one writes specifications, as well as code. In return, the system analyzes the program as it is being written and detects many errors traditional approaches would reveal only during testing (or deployed execution).

Figure 1 is a glimpse at what Spec# has to offer, with a Spec# project edited in the Visual Studio integrated development environment (IDE); shown is the definition and implementation of an interface used in a parsing framework. The method `ParseBinding` is used to pull apart a string of the form "a = b". Like many programming languages, Spec# does not allow interface methods to contain code though does allow them to have contracts, in this case a precondition (keyword **`requires`**) saying the argument provided to the method must contain the character '='.

Contracts are a native part of the Spec# language in two ways: Method contracts are part of the signature of a method, and the expressions contained in the contracts are written in the programming language itself, not in a secondary logical metalanguage (see Figure 2). `ParseBinding`'s precondition is written using a call to a method in the standard .NET library, `Contains`. The Spec# programming system comes with a set of contracts for the .NET Framework, providing a (partial) semantics for such commonly used methods.

All implementations of `ParseBinding` written in Spec# *inherit* the interface method's contract and so do not have to perform error checking or defensive programming. This behavior is illustrated in the implementation that calls the library method `IndexOf` with the assurance that the return value is a valid index into the receiver string, as guaranteed by the postcondition (not shown) of the `Contains` method. Thus, programmers can use the return value as an argument to the method `Substring`, the precondition of which requires the argument to be a valid index.

Spec# enforces `ParseBinding`'s precondition on any client making a call to the interface method. A (particularly stupid) client is shown in Figure 2, where the Spec# system has noticed

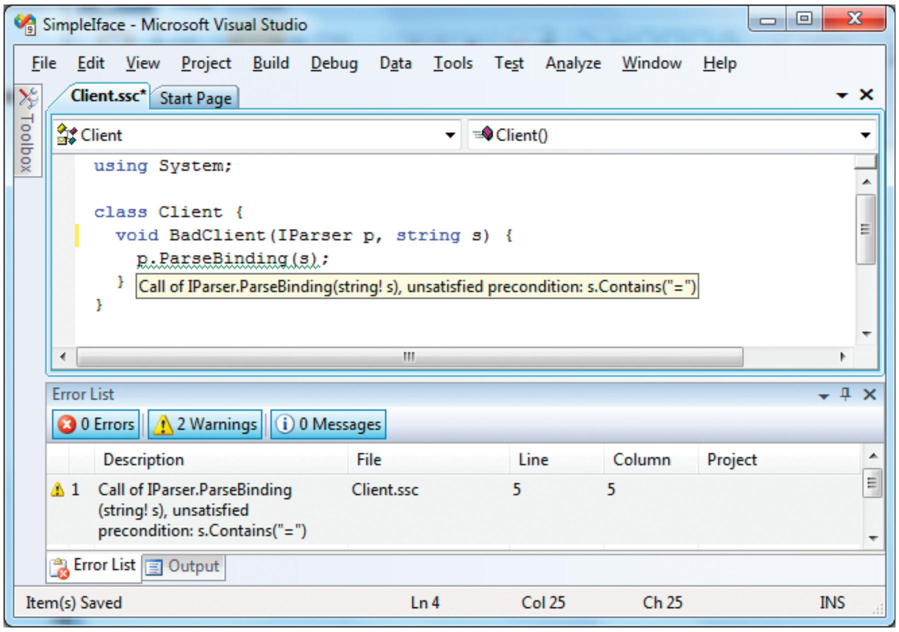

SimpleIface - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Parser.ssc* Start Page
Toolbox
Parser
Parser()
Ready Ln 9 Col 44 Ch 44 INS

```
using System;
using Microsoft.Contracts;

public interface IParser {
  void ParseBinding(string s);
    requires s.Contains("=");
}
public class Parser : IParser {
  public void ParseBinding(string binding) {
    int k = binding.IndexOf("=");
    string left = binding.Substring(0,k);
    #region...
  }
}
```

```
public string! string.Substring(int startIndex, int length)
Retrieves a substring from this instance. The substring starts at a specified ch
requires 0 <= startIndex;
requires 0 <= length;
requires startIndex + length <= this.Length;
ensures result.Length == length;
```

**Figure 1. A (partial) Spec# interface. The yellow box is a tooltip that appears when the mouse hovers over the call to Substring, showing the signature of the method and a short programmer-written summary and its contract. Just like the barking dog (as in *Silver Blaze*, Arthur Conan Doyle), the important thing to notice is the *absence* of warnings on the call to `Substring`.**


SimpleIface - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Client.ssc* Start Page
Toolbox
Client
Client()
Error List
0 Errors
2 Warnings
0 Messages
Error List Output
Item(s) Saved Ln 4 Col 25 Ch 25 INS

```
using System;

class Client {
  void BadClient(IParser p, string s) {
    p.ParseBinding(s);
  }
}
```

Call of IParser.ParseBinding(string! s), unsatisfied precondition: s.Contains("=")

| | Description | File | Line | Column | Project |
|---|---|---|---|---|---|
| ⚠ 1 | Call of IParser.ParseBinding (string! s), unsatisfied precondition: s.Contains("=") | Client.ssc | 5 | 5 | |

**Figure 2. A client using the `IParser` interface incorrectly; notice that verification errors are presented in the same format as compiler errors.**

the error as the programmer was typing the code. The resulting *squiggly*, the visual underlining in the editor, alerts the programmer that the code is violating a contract. The tooltip window shown pops up in response to the programmer hovering over the squiggled text with the mouse. The error list in the IDE is also populated with a warning about the contract error.

Note that Figures 1 and 2 do not include warnings about object references possibly being null because Spec# distinguishes between non-null types and possibly-null types. In the examples, `IParser` and **`string`** are both non-null types.

## The Spec# Language

Spec# is an object-oriented language, a superset of C# v2.0 (released in 2005), compiling to the Microsoft Intermediate Language bytecode (MSIL) and running on the .NET virtual machine and integrated into Visual Studio's IDE, which provides language services (such as syntax highlighting and the ability to run the program verifier in the background as the code is being written).

The extensions to C# consist chiefly of the standard design-by-contract features[22] (method contracts and object invariants), as well as a non-null type system. For a full introduction to the language, see the Spec# tutorial[20]; Figure 3 outlines the most commonly used features.

The first postcondition (keyword **`ensures`**) of `CrankItUp` uses the expression **`old`**`(Volume())` to refer to the value of `Volume()` on entry to the method, promising the value of `Volume()` is increased by `amount`. The second postcondition expresses that the method returns the final value of `Volume()`; the Spec# keyword **`result`** refers to the return value of the method.

Since contracts must not cause state changes, methods may be used in contracts only if they are side-effect free,[4] as indicated by the `[Pure]` custom attribute[b], as in the definition of `Volume()`.

Reasoning about a method call is in terms of the method's contract. Because method contracts are inherited in subclasses, this reasoning applies even in the presence of dynamic method dispatch where the particular method implementation invoked may not be known until runtime; that is, contract inheritance enforces the well-known concept of behavioral subtyping.[9,21]

The class `Stereo` declares three object invariants to specify what it means for an object of this class to be consistent. Whereas the first two invariants constrain the values of the fields of a `Stereo` object, the third invariant relates the states of two sub-objects. The first assignment statement in the body of the method `CrankItUp` might break that invariant before the subsequent assignment reestablishes it. To indicate that an object invariant might be temporarily violated, the two assignment statements must appear within an **`expose`** statement, described in more detail in the section on invariants. Note that no **`expose`** statement is needed in `ChangeCD`, because the single assignment maintains the invariants.

Null-dereference problems are the bane of object-oriented programming. We and others have found the single most common specification is the exclusion of the `null` value from the possible values of a field, method parameter, or result. Spec# refines C#'s type system by distinguishing non-null types (such as the type `Speaker` of the fields `left` and `right` in Figure 3) and possibly null types (written with a postfix question mark, as in `Speaker?`)[c]. References of non-null types can be dereferenced safely without requiring runtime checks or proof obligations to prevent errors.

Each object of type `Stereo` is an aggregate object containing references to other objects that make up its internal representation. In Spec#, the aggregate/sub-object relation is expressed using the `[Rep]` custom attribute in the declaration of the field pointing to the sub-object. In our example, the speakers are sub-objects of a `Stereo` object; we say the `Stereo` object

b A .NET feature that allows associating metadata with program elements.

c For traditionalists, Spec# also offers the complementary mode where `Speaker` represents the possibly null type and the non-null type is written as `Speaker!`.

**Figure 3. A (partial) Spec# program demonstrating the language's basic features, including method contracts that describe (part of) the method behavior, as well as object invariants that describe the consistent state of each instance of the class.**

```
public class Stereo {
  int currentCDSlot;
  [Rep] Speaker left = new Speaker();
  [Rep] Speaker right = new Speaker();

   invariant 0 <= currentCDSlot;
   invariant left != right;
   invariant left.Gain == right.Gain;

   public int CrankItUp(int amount)
     requires 0 <= amount;
     ensures Volume() == old(Volume()) + amount;
     ensures result == Volume();
   {
     expose (this) {
       left.Adjust(amount);
       right.Adjust(amount);
     }
     ...
   }

   [Pure] public int Volume()
   { return left.Gain; }

   public void ChangeCD(int newSlot)
     requires 0 <= newSlot;
   { currentCDSlot = newSlot; }
}
```

owns its speakers. Due to this ownership relationship, Spec# enforces that two `Stereo` objects do not share their speakers and that, in general, a speaker can be modified only through its owning `Stereo` object. This lets a `Stereo` object maintain object invariants over the state of its speakers (such as the third object invariant).

## Enforcing Spec# Contracts

A spectrum of possibilities is available for checking Spec# contracts. One extreme would be to verify them all statically; another would be to check them all dynamically. Either extreme is impractically expensive. The former involves a prohibitive specification and verification effort; the latter involves prohibitive runtime overhead. Instead, Spec# makes some checks mandatory; splitting them between dataflow analyses performed during compilation and runtime checks performed during execution; the rest are optionally enforced by a static program verifier.

The runtime checker is straightforward: each contract indicates some particular program points at which it must hold; the Spec# compiler generates a runtime assertion for each, and any failure causes an exception to be thrown.

The dataflow analysis part of the Spec# compiler primarily checks three properties: The first, and most important, is enforcing the non-null type system, which can be used independently without the other kinds of contracts in the Spec# system.

In general, a type system guarantees the static type of an expression accurately describes the possible values to which the expression can evaluate at runtime. In Spec#, an expression with a non-null type can never be observed to have a value of **`null`**. Guaranteeing this property requires controlling both assignments and initialization.[10] In particular, the type system must guarantee that a fresh object doesn't escape from its constructor before the constructor initializes all non-null fields (such as `left` and `right` in Figure 3) with non-null values. Spec# offers two solutions to this problem. One is based on a flexible placement of the **`base`** constructor call within a constructor body; the other caters to legacy code by a more sophisticated dataflow analysis.[11]

The second property is to enforce the purity of, or side-effect free, contracts; that is, they are side-effect free. Purity ensures dynamic contract checking does not interfere with the execution of the rest of the program and that contracts have a simple semantics that can be encoded in the static verifier.

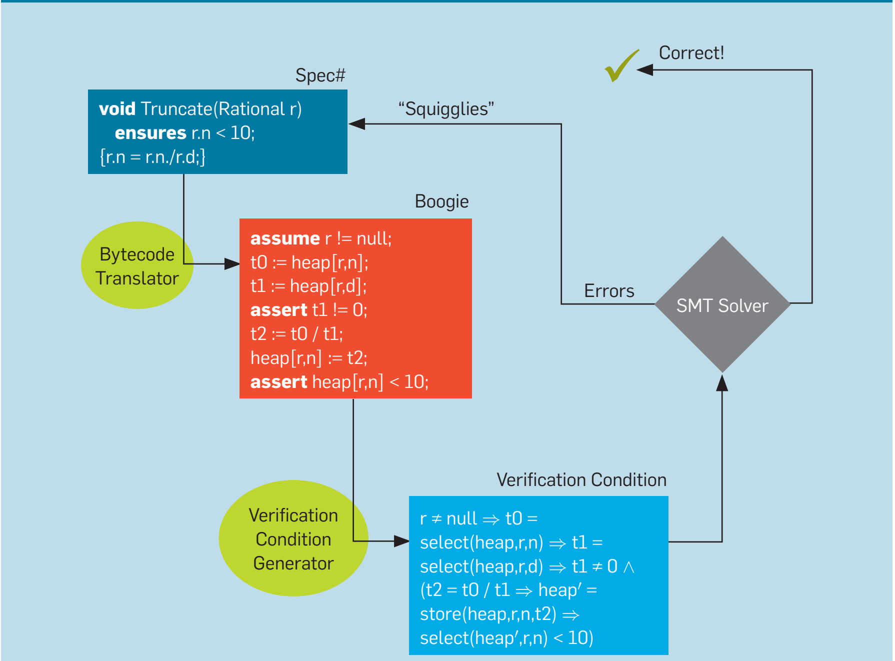
Purity could easily be enforced by forbidding all side-effecting operations (such as field updates), but doing so would be too restrictive; for instance, a method called in a specification might want to iterate over a collection. Creating and advancing an iterator are side effects; however, they are not observable when the method returns. Following JML,[18] Spec# thus enforces weak purity, forbidding pure methods from changing the state of existing objects but allowing updates to objects created within the (dynamic) scope of the method's lifetime.

The final property the compiler enforces is to limit what can be mentioned in an object invariant and what things a pure method is allowed to read. These admissibility checks are crucial for sound static verification.

The static program verifier flags violations of both the explicit contracts and the implicit contracts set forth by the language semantics (such as null dereference and array index out of bounds). It checks one method at a time. If the verification fails, Spec# displays an error message, the location of the error, the trace through the method containing the error, and possibly a counterexample; Figure 2 illustrates how the IDE reports verification errors to the programmer.

The Spec# static verifier is sound but not complete; that is, it finds all errors in a program but might also warn about methods that are actually correct. Such spurious warnings are often fixed by providing more comprehensive specifications. In some cases, it may be necessary for the programmer to add an assumption to the program using the program statement **`assume`** `e`. The condition `e` is blindly assumed by the static verifier but is checked at runtime. Assumptions require special attention during testing and code reviews.

**Figure 4. Verfication pipeline.**



The semantics of Spec# is encoded in Boogie. The heap is modeled as a 2D array indexed by object references and field names. Within a method body, preconditions and type information are encoded as assumptions and postconditions and implicit language contracts as assertions. The verification condition is expressed in a standard format supported by many automatic theorem provers. When the theorem prover reports errors, they are mapped back to the Boogie program and then to the Spec# source. Another outcome (not shown) is when the prover runs out of time or space.

Verification proceeds via a series of transformations starting with the Spec# program and ending with a mathematical formula that is then input to an automated first-order theorem prover. The formula, called a verification condition, is valid if and only if there are no violations of implicit or explicit contracts (see Figure 4). The gap between the program and the formula is bridged by translating the Spec# program into a much simpler program, much as a compiler bridges the gap between source program and machine code by translating into an intermediate representation. For this purpose, we defined the intermediate verification language Boogie[2]; we also created a way to derive verification conditions for Boogie programs by computing their weakest preconditions. In essence, Boogie has only assignment statements, assertions, assumptions, and branches. A method call is modeled by asserting all of the method's preconditions, assigning arbitrary values to anything the method might modify (things within its frame) and assuming the method's postconditions. Boogie was designed to support more front ends than Spec#.

Note that static verification does not fully replace testing. Tests are still necessary to ensure the requirements are captured correctly, to check the properties that are not expressed by contracts, and to check properties ignored by the Spec# verifier (such as stack overflows).

**If verification ever makes it into the daily rhythm of mainstream programming, it will be through a design-time interface providing online verification.**

## Invariants

Spec# performs modular reasoning, which is to a verifier what separate compilation is to a compiler; each module is verified separately and does not need to be re-verified when the modules are combined into a whole program (see Figure 3). To reason soundly in the presence of object invariants and mutable state, Spec# uses a methodology that restricts programs and guides their use of specifications. Here, we outline this important contribution of sound modular reasoning; a full explanation is in the Spec# tutorial.[20]

A first problem with object invariants is deciding on the program points at which they should hold. An invariant cannot always hold; it is generally necessary to temporarily violate an invariant with later state changes reestablishing it, as outlined in Figure 3 by method `CrankItUp`. It is also not possible to say an object invariant holds on method boundaries; method calls made within a method could make the object accessible outside the class while in an inconsistent state.

A second related problem is that an object invariant often depends on the state of other objects; for instance, the invariant of an aggregate object typically depends on the state of its sub-objects, as illustrated by `Stereo` in Figure 3. Consequently, modification of these sub-objects potentially violates the invariant of the aggregate. This situation is inescapable for any system with reusable components, so the methodology must allow it. But the verifier must ensure the aggregate object's invariants are reestablished before the aggregate relies on them again.

Spec# solves the first problem with the **`expose`** statement, which is similar to a lock statement in concurrent programming. The **`expose`** statement indicates a non-reentrant lexical region within which an object's state is vulnerable and within which the invariant may be temporarily violated. The object invariant must hold in order for the block to be entered or exited.

The second problem is solved by introducing an ownership system in which the objects of the heap are organized into a collection of tree structures. The edges of the trees indicate ownership, or an aggregate/sub-object relationship. An object invariant can depend roughly only on state contained in the subtree of which it is the root. Within an **`expose`** block, a method can call down in the ownership tree but not up, preventing method calls on inconsistent objects.

This is the basic approach to specifying aggregate objects in Spec#. However, many object-oriented programs not only involve hierarchical data structures but also consist of mutually referring objects (such as the subject-observer pattern or doubly linked lists). To deal with such peer relationships, the Spec# methodology uses the notion of peer consistency, or that an object and all its peers are consistent;

## The Spec# Ecosystem

**Spec# also relies on other projects developed within Microsoft Research:**

***CCI.*** **The Microsoft Research Common Compiler Infrastructure is a set of base classes that implement common functionality needed by compilers, taking care of intermediate code generation and helping with symbol table management, metadata importing, name resolution, overload resolution, and error reporting. It also includes functionality for language integration into the Visual Studio development environment. First developed within Microsoft as part of the implementation of Comega, it has since been used for other compilers, including the Spec# compiler. A redesigned version of the core parts of CCI was released in 2009 as an open source project (http://ccimetadata.codeplex.com).**

***Boogie.*** **Boogie[19] is a verification platform consisting of an intermediate verification language and a tool that generates logical verification conditions. The Boogie language offers a level of abstraction suitable for modeling the behavior and proof obligations of a source language; for example, it supports procedures with contracts, local and global variables, structured and unstructured control flow, a polymorphic type system, and first-order mathematical definitions. These features make it convenient for verification systems to encode imperative and object-oriented programs. Verifiers built on top of Boogie translate source programs into Boogie programs and invoke the Boogie tool. Boogie computes efficient verification conditions for its input program, sends them to a theorem prover (such as the SMT solver Z3), and makes the results available to programmers and upstream tools. We initially developed Boogie within the Spec# project, but, as noted, Boogie is now employed by many program verifiers. Boogie has been an open source project (http://boogie.codeplex.com) since 2009.**

***Z3.*** **Z3[7] is a state-of-the-art SMT solver combining decision procedures for functions, arithmetic, and logical quantifiers. Due to its high performance, it is the default SMT solver used by the Boogie verification engine. When a proof attempt fails, Z3 returns information from which Boogie extracts the failed assertion, the trace through the method to be verified leading to the failure, and a counterexample with possible values for local variables and heap locations. This information gives programmers precise and helpful error messages; see Z3's Web site http://research/microsoft.com/projects/z3/.**

see the Spec# tutorial[20] for details.

A third problem of modular reasoning is framing, which deals with what "frame" can be put around a method call to limit the effects the call might have; for instance, in method `CrankItUp`, what is the program state after the call to `left.Adjust`? A pessimistic approach is to treat the call as modifying everything in the heap, since potentially all objects in the heap are reachable from every method (such as through static fields). A better solution would be to know exactly which parts of the heap a method changes, but, in the presence of subclassing and information hiding, a method contract cannot name these parts directly. Instead, some form of abstraction is needed but one that is precise enough for the program verifier. Spec# solves this problem by again utilizing its ownership system; without an explicit specification stating otherwise (keyword **`modifies`**), a method may modify only the fields of the receiver and those of objects within the subtree of which the receiver is the root. Using ownership to abstract over the modifications of sub-objects is justified, because clients of an object should not be concerned with its sub-objects; for instance, clients of `Stereo` objects need to know only about the result of `Volume()`, not how the volume is stored in the sub-objects of `Stereo`.

### Songs of Innocence

We began the Spec# project in 2003 as an attempt to build a comprehensive program-verification system,[3] hoping to build a real system real programmers could use on real programs to perform real verification, a system the "programming masses" could use in their everyday work. Along the way, we wanted to explore and push the boundaries of specification and verification technology to get closer to realizing these aspirations.

At the time, program verification was already decades old, starting with formal underpinnings of program semantics and techniques for proving program correctness.[14] Supported by mechanical-proof assistants, early program verifiers included the GYPSY system and the Stanford Pascal Verifier. Later systems, still used today, include full-featured proof assistants like PVS and Isabelle/HOL.

Another approach to improving program quality via verification technology is extended static checking, which included checkers like ESC/Modula-3[8] and ESC/Java.[13] These tools have been more closely integrated into existing programming languages and value automation over expressivity or soundness. The automation is enabled by a breed of combined decision procedures that today is known as Satisfiability Modulo Theories, or SMT, solvers. To make their use easier and more cost-effective, extended static checkers were intentionally designed to be unsound; that is, they could miss certain errors.

Dynamic checking of specifications has always been done by the Eiffel programming language,[22] which pioneered inclusion of contracts in object-oriented languages. The tool suite for the Java Modeling Language (JML) also included a facility for dynamic contract checking.[4] The strong influence of both Eiffel and JML on Spec# is evident.

Our plan in 2003 targeting real programs was no doubt our single most important decision and has permeated every aspect of the Spec# design. Targeting real programs meant not designing a tool for a toy language with idealized features. In addition to learning how to handle difficult or otherwise uncomfortable language features, a benefit of this decision is the large body of programs and libraries that can be used as starting points for specification and verification. It also implied a connection with an existing language, so we built our language extensions around C# and the .NET platform. Other well-known real languages with specifications were Eiffel, Java+JML, and SPARK Ada[1]; as in GYPSY and Eiffel, our extensions made specifications part of the language itself.

Our plan to build a system for real programmers immediately ruled out the possibility of exposing programmers to an interactive proof assistant. We felt that while programmers must know how to specify a program, they should not need to understand proof theory, the logical encoding of a program's semantics, or how to issue tactics to guide the proof search. Instead, we turned to an automatic SMT solver. This is not to say that verification is fully automatic; Spec# programmers must still supply specifications, but all interaction between them and Spec#'s tools takes place in the context of the program and its specifications. The major contender here was ESC/Java and similar tools using JML specifications in Java programs.

Another important consequence of building a system for real programmers was the need for something to attract real programmers. It is a long journey indeed for programmers to arrive at the point of writing specifications that lead to effective verification. To give them immediate benefit for any specification they write, however partial, we included in Spec# dynamic checking of specifications. Throughout the project, we also worked on providing good defaults so programmers would not be unduly burdened in the most common cases.

Finally, our plan for real verification meant not compromising on soundness while also aligning with fully featured proof assistants. Sound verification of object-oriented programs does not come easily. Of the unsound features in ESC/Java, many were known to have sound solutions. But two open key areas were how to verify object invariants in the presence of subclassing and dynamically dispatched methods (giving rise to the possibility of callbacks, or situations where the caller of a method is reentered during the execution of the method it called), as well as method framing. To ensure our verifier would scale to large programs and could be applied to libraries, we also wanted to support modular verification. We began the project with an idea for a methodology that addresses these problems, providing a glimmer of hope for building a sound and modular verifier.

Though we aimed for a broad design, we initially left out several things so we could provide simpler specifications; for example, we provided no support for writing specifications for unsafe (non-type-safe) code, concurrency, higher-level aspects of closure objects, and some functional correctness concerns of algorithmic verification. The specifications focused instead on partial properties, of the kind every programmer could write down and for which might be willing to accept the runtime overhead of dynamic checking. We subsequently added other features we left out of our initial design (such as generics).

We set out to build a programming system where both the programming language and integrated tooling support specifications. The system was intended to blend into existing practices, provide a range of assurance levels, from dynamic checking to static verification, and deliver static verification that was sound and automatic. Its success depended on answering a number of scientific questions, as well as solving non-trivial engineering concerns.

## Influence

Here, we explore Spec#'s influence on researchers and language designers in academia and industry.

**Scientific results.** The Spec# project's main research focus has been on improving verification methodology by identifying common programming idioms and developing techniques and notations for their specification and verification. We built a state-of-the-art system and advanced the state of verification. First, the Spec# methodology supports sound modular verification of object invariants in the presence of multi-object invariants, subclassing, and reentrancy. We also worked out some of the difficulties with abstraction features (such as pure methods). Spec#'s dynamic ownership model allows programmers to express heap topologies and use them for verification. The Spec# project gained practical experience through a design of non-null types and incorporated flexible object initialization schemes. It advanced the foundations of program verification by, for instance, providing a verification-condition generator for unstructured programs. And finally, by providing IDE support and continuously running the program verifier in the background, Spec# broke new ground in how programmers work with a verifier. The scientific contributions of the Spec# project have been published in more than 30 articles.

**Impact on academic research and teaching.** A number of research projects build directly on the Spec# infrastructure; for example, SpecLeuven[16] is an extension of the Spec# methodology and tools to handle concurrency, using Spec#'s ownership system to enforce locking strategies. Several research groups use the Boogie verification engine developed as part of the Spec# project[2]; for instance, various Java/JML, bytecode/BML, and Eiffel projects use Boogie as a target for their verifiers. At the other end, Boogie's output is now also fed to interactive theorem provers. In addition, we've seen researchers encode and verify new logics, as well as verify challenging examples (such as garbage collectors).

Other projects do not use the Spec# infrastructure but seem to be influenced and inspired by the project. For example, Eiffel supports attached types, a variation of a non-null type system. JML does not include a non-null type system but offers non-null annotations, which are the default for all reference types. The idea to run a program verifier within an IDE and report verification errors just like compiler errors has been picked up by ESC/Java2, which comes with Eclipse integration. Likewise, the Rodin tool provides Eclipse integration for the Event-B tools.

Spec# has also been used to teach program verification at universities, mostly in graduate seminars. We and others have also taught Spec# in a number of summer schools, as well as at major conferences.

**Impact within industry.** In 2003, we hoped to convince one of the programming language teams at Microsoft to add Spec#-like features. However, influencing such a team is itself a difficult proposition, and even if we had succeeded, our single-language story did not address the fact that .NET is a multi-language platform. We also lacked support for unsafe code and for concurrency while battling a perception that verification is relevant only for safety-critical software. Even so, Spec# has influenced other projects in Microsoft Research and several Microsoft product groups. This influence can be grouped into two main categories: Boogie and Code Contracts for .NET:

*Boogie.* The verification engine originally developed for Spec#, called

Boogie, has become an independent project (see the sidebar "The Spec# Ecosystem") used in other projects inside and outside Microsoft. Here are some of them:

The HAVOC tool uses Boogie to verify low-level sequential systems code written in C[5] and has been applied to verify properties of device drivers and critical components in the Windows kernel. A version of HAVOC has also been targeted in Microsoft at finding specific errors in a very large code base of systems code.

The VCC[6] tool built at Microsoft Research adopts Spec#'s tool chain and methodology for C code, addressing Spec#'s limits in two dimensions: full functional verification and verifying concurrent operating system code. For the latter, VCC allows two-state invariants spanning multiple objects without sacrificing thread or data modularity. VCC is being used to verify the kernel of Microsoft Hyper-V (an industrial virtualization platform), the PikeOS embedded operating system, and the LEDA data-structure library.

The type safety of the Verve operating system built at Microsoft Research has been verified; the lowest level of that verification concerns assembly code written for verification and compilation in a stylized form of the Boogie intermediate verification language.

Also at Microsoft Research, SymDiff is a project built on Boogie providing an infrastructure for building tools and techniques for statically providing feedback about program changes. It is being used to ensure "app-compat," whereby evolving programs are checked for compatibility, or relative correctness as opposed to absolute correctness.

*Code Contracts for .NET.* In 2009, Spec# inspired a new project: Code Contracts for .NET. To avoid having to get programmers to adopt (and support) a new language, we introduced a library-based approach where specifications are written as method calls to the library within the actual code. Calls to the contract library, including methods like `Contract.Requires` for precondtions can be called from any .NET program. Both method contracts and object invariants are supported, though we intentionally do not (yet) offer a sound treatment for invariants. Non-null types and purity checking are not supported.

Standard compilers generate the normal MSIL code for calls to contract methods, wheras the Code Contracts tools use post-build steps to extract the contracts and use them for both dynamic and static checking. Starting with .NET 4.0 in 2010, the contract library is now a part of mscorlib, .NET's standard library. The associated tools are distributed through DevLabs, a Visual Studio Web site[d] where early technology is made available for collecting community feedback.

## Songs of Experience

Here, we reflect on our initial aspirations and design decisions:

**Not a toy language.** The fact that we built Spec# as a full-scale .NET language and developed a mode for it within the Visual Studio IDE has had far-reaching consequences; for example, it made the scope of the project large enough to include a wealth of scientific and engineering challenges. The project shows it is possible to build a practical verifier at this scale; given the availability of SMT solvers and verification engines like Boogie and Why,[12] the task of building a verifier is now more straightforward than it was a decade earlier.

Most important, being a full language that compiles to a common platform has increased the credibility of Spec# research, letting us approach programmers and managers, especially at Microsoft, who might not have been impressed by a one-off system. Integration into Visual Studio allowed us to perform background verification at design time, immediately indicating errors in the program text by design-time squigglies. It also allowed us to populate tool tips with contracts that boost programmer understanding of the code. A crucial consequence of the IDE integration is that it has allowed us, through live demos, to communicate the Spec# vision. Demos aside, if verification ever makes it into the daily rhythm of mainstream programming, it will be through a design-time interface providing online verification.

Having access to existing programs has two important advantages: let research teams try out ideas and brutally reveal problems that still need solutions. It thus both validates research and guides the way to more research problems to be tackled. Seeing the results of our Spec#-related experiments over the years often forced us to support previously ignored features and alter and expand our specification methodology.

Dealing with a full language also has disadvantages. Building the prototype system takes effort but is helped by the initial enthusiasm that goes with creating a new research project. However, with most of the system in place, adding or modifying features becomes a larger effort than one would wish. We often felt we could not move as quickly as we wished; for example, adding a new syntactic feature required changes not just to the parser but often also to the rest of the compiler, the admissibility checker, the part of the system that persists and recalls contracts in compiled libraries, and the verifier. We also put a lot of effort into producing contracts for the existing .NET libraries, using a heuristic tool that mined the binaries, manually adding contracts as needed, and adding system support for them.

Our IDE integration did just enough to communicate our vision. However, our implementation is far inferior to product-quality integration, and the Spec# mode in Visual Studio is downright clunky compared to the whizbang C# mode. Using our own integration, we gave programmers feedback as the program is keyed in but unable to take advantage of all IDE advances (such as refactoring support) without a prohibitively large engineering investment.

If we were to do the Spec# research project again, it is not clear that extending an existing language (here C#) would be the best strategy. Not only does extending C# mean having to deal with constructs that are difficult to reason about but also presents a maintenance problem as the base language evolves; for example, migrating Spec# to extend C# versions 3 and 4 would require more development resources than our small research team has, all for the purpose of supporting features of marginal research return.

d http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx

In contrast, SPARK Ada was built as a subset of Ada and more readily lets a language designer pick features that mesh well with verification and trivially solve the problem of what to do as the base language evolves. Subsetting makes it more difficult to apply the verifier to legacy code.

A final point about developing a verifier in a multi-language platform is the question of where in the compilation chain to apply verification. The Spec# verifier actually starts with the MSIL bytecode the compiler produces, letting the verifier ignore syntactic variations offered by the source language (such as `for` loops versus `while` loops) and allowing the building of cross-language verifiers. But for some features, like the syntactic support for iterators in C#, it would be much easier to start with the source constructs than either verify or first reverse engineer the auxiliary classes and chopped-up method bodies the compiler emits into the bytecode. While it may be tempting to start at the source, the right thing is to start with MSIL; otherwise, every language would have to write its own verifier. Compilers should annotate the bytecode to make it easy to recover higher-level information.

**Non-null types.** Non-null types have proven useful and easy to use. Like other successful type features, they provide an enforceable discipline that hits a sweet spot of ruling out most programs with certain kinds of errors while allowing most programs without such errors. In our experience, programmers almost universally like the non-null types, with the exception of converting legacy code.

We began the Spec# project with reference types being possibly null by default (as in C# and Java) and requiring the type modifier ! to express a non-null type, except for local variables where the type checker automatically inferred the non-null mode. We found that in our own code, the non-null-by-default option led to less clutter. Consequently, our suggestion to future language designers is to let possibly-null types be the option.

Our non-null type system has some holes resulting from the engineering compromises we had to make to integrate the types into an existing platform. A complete system needs support from the .NET virtual machine to, say, ensure that element assignments to arrays respect the covariant arrays of .NET. Another example is that handling non-null static fields requires more control during class initialization than the .NET machine provides. We designed (sometimes complicated) workarounds for the lack of virtual-machine support but did not implement them all. We hope future execution platforms will be designed with non-null types in mind.

**In our experience, programmers almost universally like the non-null types, with the exception of converting legacy code.**

Our attempt at technology-transfer of non-null types into Microsoft languages also yielded a surprise. We had felt that the fruits of our non-null research were ready for prime time, but, as just described, non-null types do not reap their full benefits in a single language, instead needing platform support. Also as we described, we had better luck with technology transfer of contracts.

**Dynamic and static checking.** Another major goal of our initial design was support for both dynamic and static checking of specifications. Such support has several advantages; most important, it immediately rewards programmers for writing specifications, because they turn into useful runtime assertions. Each specification added in that way helps reduce the additional cost of writing provable specifications at a time when the program might be verified. Another advantage of using specifications for both dynamic and static checking is that programmers only need to learn one specification language.

Whenever dynamic checking would be too expensive (such as in the enforcement of method frames) our principle was to drop the dynamic check. If dynamic checks are a subset of the static checks, then such a design has the nice property that any program that is statically verified will run without dynamic violations of specifications. An important exception to this subsetting is the `assume` statement, the sole purpose of which is to trade a dynamic check for an assumption by the static verifier.

A point that often comes up in discussions with colleagues is the prospect of omitting the dynamic checks of those specifications that have been statically verified. We never got around to trying such an optimization for Spec#. However, we offer a word of caution to others who might consider doing so: Preserving the soundness of such optimizations is difficult unless the whole program is forced to undergo verification. Such a requirement on the whole program is practically impossible in the .NET environment, where the interoperating languages have neither static nor dynamic checking.

**Verifying loops.** A familiar issue with static verification is the need for loop invariants, which are analogous to the inductive hypotheses used to prove theorems inductively in mathematics. Whereas a tool like ESC/Java avoids this issue by checking only a bounded number of iterations of each loop, Spec# verifies all iterations, but comes at a cost. We have found this cost to be moderately low, explaining it as follows: The effective loop invariant draws from three sources: One is that the Spec# program verifier automatically infers simple loop invariants. A second is loop-modification inference,[8] whereby Spec# enforces the method frame on every heap update, so the `modifies` clause of the enclosing method can be incorporated as an automatic part of the loop invariant. This is the "biggest" part of the effective loop invariant and also the part no user would want to supply explicitly. The third source is user-supplied loop invariants. Due to the first two sources, many loops require no user-supplied loop invariants at all, especially for methods with no postconditions. As programmers and tools take steps toward functional-correctness verification, the need for user-supplied loop invariants is likely to increase.

**Methodology.** The Spec# methodology led us to the first implementation of a sound modular approach to specifying and verifying object invariants and method frames. Compared to earlier solutions,[23] the Spec# methodology is better suited for automatic verification using SMT solvers. Since 2003, other researchers have designed alternative methodologies.[17,24] The Spec# methodology has been streamlined for some common object-oriented patterns (such as aggregate objects), lending itself to concise specifications of programs that fall within those common patterns. However, for programs using more complicated patterns, the methodology can be too restrictive; for example, it caused us to lose the interest of one programmer at Microsoft who was avidly trying to verify a large body of code.

**We set out to build a programming system where both the programming language and integrated tooling support specifications.**

The learning curve of the methodology has been steeper for programmers than we would have liked, and non-experts of the methodology sometimes have problems knowing what to do in response to certain error messages. The fact that we constantly changed the methodology to improve it, and along with it the terminology we used, complicated programmers' understanding. We hope the 2010 tutorial[20] has mitigated this problem.

**More.** We'd like to make a few more remarks about our Spec# experience: First, we set out to build a sound verification system. While we found sound solutions to fundamental problems of modular verification of object-oriented programs, our implementation is not perfect, including several unimplemented features and other unfound errors and semantic encoding.

Second, the best and most far-reaching single design decision we made in implementing the Spec# verifier was to introduce the intermediate language Boogie between the Spec# program and the formulas sent to the theorem prover. Boogie permits manual authoring, as well as automatic translation.[2] This extra layer of indirection allowed us to investigate many alternative design decisions in a lightweight fashion by hand-modifying the translated Spec# program. Another benefit of having this important separation of concerns is the use of Boogie as the backend for other programming languages and verification systems and as the frontend for different theorem provers.

Third, a conclusion we draw from watching programmers beginning to use Spec# is that they really do appreciate contracts, which are not just an esoteric feature prescribed by "Dijkstra clones." Unfortunately, we have also seen them develop unreasonable expectations for how easy it will be to statically verify their programs using contracts. Post-installation depression might then set in as they have difficulty trying to verify their own programs.

Fourth, from a research standpoint, having source syntax for specification constructs lets program text be concise and usefully descriptive, making clear the important concepts. However, from an adoption standpoint, this approach involves two problems: One is that the engineering overhead associated with being a superset language is a high price for the developers of the verification system to pay; the other is that we want specifications to be adopted in the platform, not just in a single language. Therefore, we gradually steered toward a language-independent solution by providing the specification constructs via the Code Contracts library, making it possible for individual languages to consider adding convenient source syntax.

## Conclusion

Following the Spec# project, the Verified Software Initiative[15] was formed in 2005 to encourage the verification community to work toward larger projects, addressing larger risk, and taking a long-term view of program verification. All our work on Spec# fits this initiative.

Spec# has evolved from a single-language vision into four ongoing technical themes:

One is the Spec# project itself, which entered a new phase in 2009, hoping, through a new open-source release[e], to see continued improvement in the Spec# programming system from a larger community and also more use of Spec# in teaching, especially in light of the 2010 comprehensive tutorial.[20]

A second theme is the Boogie language and verification engine, providing an infrastructure used as an abstraction layer in several verification projects.[f]

A third theme is a new strand of research attempting automatic functional-correctness verification. Using a variation of the Spec# methodology and the Boogie intermediate verification language, VCC is an example of such a project, and Hyper-V's experience with it has been positive.

A fourth theme aims at the mass adoption of specifications in programming. The language-independent Code Contracts project bypasses the problem of trying to get a single new language to be accepted as the standard programming language for all programmers by putting specification facilities into the underlying platform. We hope that since all .NET programmers can make use of the associated tools for runtime checking, static checking, automatic documentation generation, and intelligent programmer assistants, this will lead to an improved general software-engineering process.

These themes continue to push frontiers in the quest for verified software.

## Acknowledgments

Spec# owes a great deal to the research interns who helped bring it into existence and to the feedback from programmers who have used it. We also express our thanks to the early reviewers and anonymous referees of this article. C

e http://specsharp.codeplex.com

f http://boogie.codeplex.com

**References**

1. Barnes, J. *High-Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.
2. Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K.R.M. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of Formal Methods for Components and Objects, Volume 4111 LNCS* (Amsterdam, The Netherlands, Nov. 1–4, 2005). Springer, 2006, 364–387.
3. Barnett, M., Leino, K.R.M., and Schulte, W. The Spec# programming system: An overview. In *Proceedings of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Volume 3362 LNCS* (Nice, France, Mar. 8–11). Springer, 2005, 49–69.
4. Burdy, L., Cheon, Y., Cok, D. R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., and Poll, E. An overview of JML tools and applications. *Electronic Notes in Theoretical Computer Science 80* (2003), 212–232.
5. Chatterjee, S., Lahiri, S.K., Qadeer, S., and Rakamaric, Z. A reachability predicate for analyzing low-level software. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems, Volume 4424 LNCS* (Braga, Portugal, Mar. 24–Apr. 1). Springer, 2007, 19–33.
6. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *Proceedings of Theorem Proving in Higher Order Logics, Volume 5674 LNCS* (Munich, Aug. 17–20). Springer, 2009, 23–42.
7. de Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems, Volume 4963 LNCS* (Budapest, Mar. 29–Apr. 6). Springer, 337–340.
8. Detlefs, D.L., Leino, K.R.M., Nelson, G., and Saxe, J.B. *Extended Static Checking Research Report 159.* Compaq Systems Research Center, Palo Alto, CA, 1998.
9. Dhara, K.K. and Leavens, G.T. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the International Conference on Software Engineering* (Berlin, Mar. 25–30). IEEE Computer Society Press, 1996, 258–267.
10. Fähndrich, M. and Leino, K.R.M. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, Oct. 26–30). ACM Press, New York, 2003, 302–312.
11. Fähndrich, M. and Xia, S.. Establishing object invariants with delayed types. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Montréal, Oct. 21–25). ACM Press, New York, 2007, 337–350.
12. Filliâtre, J.-C. and Marché, C. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of Computer Aided Verification, Volume 4590 LNCS* (Berlin, July 3–7). Springer, 2007, 173–177.
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R. Extended static checking for Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, June 17–19). ACM Press, New York, 2002, 234–245.
14. Floyd, R.W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Volume 19 of Proceedings of Symposia in Applied Mathematics.* American Mathematical Society, 1967, 19–32.
15. Hoare, C., Misra, J., Leavens, G.T., and Shankar, N. The verified software initiative: A manifesto. *ACM Computing Surveys 41*, 4 (2009), 1–8.
16. Jacobs, B., Leino, K.R.M., Piessens, F., Smans, J., and Schulte, W. A programming model for concurrent object-oriented programs. *ACM Transactions on Programming Languages and Systems 31*, 1 (2008), 1–48.
17. Kassios, I.T. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of World Congress on Formal Methods, Volume 4085 LNCS* (Hamilton, Canada, Aug. 21–27). Springer, 2006, 268–283.
18. Leavens, G.T., Baker, A.L., and Ruby, C. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes 31*, 3 (2006), 1–38.
19. Leino, K.R.M. *This is Boogie 2. Microsoft Research Technical Report* (June 2008).
20. Leino, K.R.M. and Müller, P. Using the Spec# language, methodology, and tools to write bug-free programs. *Advanced Lectures on Software Engineering: LASER Summer School 2007/2008, Volume 6029 LNCS.* Springer, 2010.
21. Liskov, B.H. and Wing, J.M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 6 (1994), 1811–1841.
22. Meyer, B. *Object-oriented Software Construction.* Prentice-Hall, 1988.
23. Müller, P. *Modular Specification and Verification of Object-Oriented Programs, Volume 2262 LNCS.* Springer, 2002.
24. Parkinson, M.J. and Bierman, G.M. Separation logic and abstraction. In *Proceedings of the Symposium of Principles of Programming Languages* (Long Beach, CA, Jan. 12–14). ACM Press, New York, 2005, 247–258.

**Mike Barnett** (mbarnett@microsoft.com) is a principal research software design engineer in the RiSE group at Microsoft Research, Redmond, WA.

**Manuel Fähndrich** (maf@microsoft.com) is a senior researcher in the RiSE group at Microsoft Research, Redmond, WA.

**K. Rustan M. Leino** (leino@microsoft.com) is a principal researcher in the RiSE group at Microsoft Research, Redmond, WA.

**Peter Müller** (peter.mueller@inf.ethz.ch) is a professor in the Department of Computer Science at the Swiss Federal Institute of Technology (ETH), Zürich, Switzerland.

**Wolfram Schulte** (schulte@microsoft.com) is the research area manager leading the RiSE group at Microsoft Research, Redmond, WA.

**Herman Venter** (hermanv@microsoft.com) is a principal research software design engineer in the RiSE group at Microsoft Research, Redmond, WA.