Specification and Verification: The Spec# Experience

Mike Barnett
Microsoft Research, Redmond
mbarnett@microsoft.com

Manuel Fähndrich
Microsoft Research, Redmond
maf@microsoft.com

K. Rustan M. Leino
Microsoft Research, Redmond
leino@microsoft.com

Peter Müller
ETH Zurich
peter.mueller@inf.ethz.ch

Wolfram Schulte
Microsoft Research, Redmond
schulte@microsoft.com

Herman Venter
Microsoft Research, Redmond
hermanv@microsoft.com

ABSTRACT

Spec# is a programming system that facilitates the development of correct software. The *Spec# language* extends C# with contracts that allow programmers to express their design intent in the code. The *Spec# tool suite* consists of a compiler that emits run-time checks for contracts, a static program verifier that attempts to mathematically prove the correctness of programs, and an integration into the Visual Studio development environment. Spec# shows how contracts and verifiers can be integrated seamlessly into the software development process. This paper reflects on the six-year history of the Spec# project, scientific contributions it has made, remaining challenges for tools that seek to establish program correctness, and prospects of incorporating verification into everyday software engineering.

0. INTRODUCING SPEC#

Programming is, as everyone knows, hard. There are many reasons for this, but a primary reason is the inability today to ensure that a program behaves as intended. What is needed is both a way to record that intent and tools that enforce it.

Spec# (pronounced "speck sharp"), available at `specsharp.codeplex.com`, is a research project aimed at addressing this problem in the context of modern object-oriented languages. It uses the well-known approach of providing *contracts*, specification constructs that document behavior [22]. Contracts standardize the common practice of writing *assertions* within code via two main constructs:

Method pre- and postconditions are part of the application-programming interface (API) for methods. Preconditions state what is to be true at method entry. Callers must establish them and implementers can assume them. Postconditions state what is to be true at method exit. Implementers must establish them and callers can assume them upon method return.

Object invariants provide a way to specify the steady-state properties that all "good" instances of a class should maintain. A crucial feature that sets Spec# apart is a *sound* system for reasoning about when object invariants hold. Sec. 3 explains why this is such a difficult problem and how Spec# solves it.

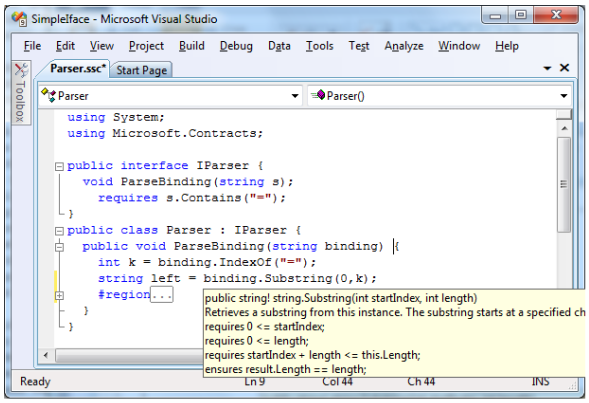

Figure 0: A (partial) Spec# interface. The yellow box is a *tooltip*: it appears when the mouse "hovers" over the call to `Substring`. It shows the signature of the method, a short programmer-written summary, and its contract. Just like the barking dog (cf. *Silver Blaze*, A. Conan Doyle), the important thing to notice is the *absence* of warnings on the call to `Substring`.

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

A programmer interacts with Spec# just like 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 while it is being written and detects many errors that traditional approaches would reveal only during testing (or deployed execution).

Fig. 0 provides a first glimpse of what Spec# has to offer: a Spec# project is being edited in the Visual Studio Integrated Development Environment (IDE). It shows 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". Spec#, like many programming languages, does not allow interface methods to contain code. However, it does allow them to have contracts, in this case a precondition (keyword **`requires`**) saying that the argument provided to the method must contain the character '='.

Contracts are a native part of the Spec# language in two ways. First, method contracts are part of the signature of a method. Second, the expressions contained in the contracts

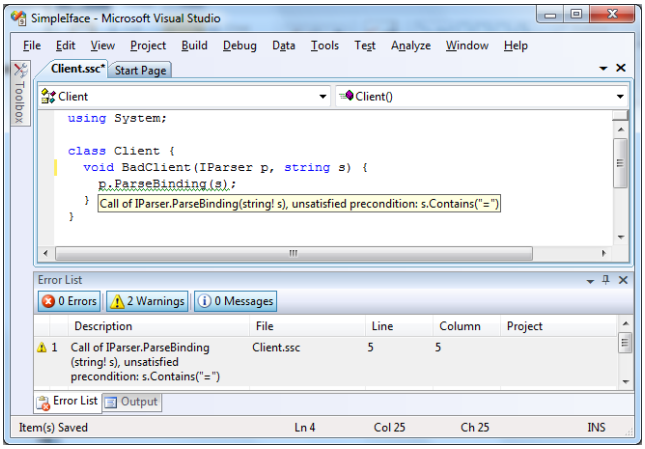

Figure 1: A client using the `IParser` interface incorrectly. Notice that verification errors are presented in the same format as compiler errors.

are written in the programming language itself, not in a secondary logical meta-language. `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 do any error checking or defensive programming. This is illustrated in the implementation, which 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, the programmer can use the return value as an argument to the method `Substring`, which has that as *its* precondition.

Spec# enforces `ParseBinding`'s precondition on any client making a call to the interface method. A (particularly stupid) client is shown in Fig. 1 where the Spec# system has noticed 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 that is 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.

Notice that Figs. 0 and 1 do not include any warnings about object references possibly being null. This is because Spec# distinguishes between non-null types and possibly-null types. In the examples, `IParser` and **`string`** are both non-null types.

1. THE SPEC# LANGUAGE

Spec# is an object-oriented language; it is a superset of C# v2.0 (released in 2005). It compiles to MSIL bytecode, runs on the .NET virtual machine, and is integrated into Visual Studio's integrated development environment (IDE), which provides language services such as syntax highlighting and also 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. A full introduction to the language is found the in Spec# tutorial [20]; Fig. 2 demonstrates the most commonly used features.

```
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; }
}
```

Figure 2: A (partial) Spec# program demonstrating the basic features of the language. It contains method contracts that describe (part of) the method behavior and object invariants that describe the consistent state of each instance of the class.

The first postcondition (keyword **`ensures`**) of `CrankItUp` uses the expression **`old`**`(Volume())` to refer to the value of `Volume()` on entry to the method. It promises that 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 any state changes, methods may be used in contracts only if they are side-effect free [3], which is indicated by the `[Pure]` custom attribute[0] 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 run time. In other words, contract inheritance enforces the well-known concept of *behavioral subtyping* [21, 9].

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 re-establishes it. To indicate that an object invariant might be temporarily violated, the two assignment statements must appear within an **`expose`**

[0] A .NET feature that allows associating *meta-data* with program elements.

statement, which is described in more detail in Sec. 3. Note that no **expose** statement is needed in `ChangeCD`, because the single assignment maintains the invariants.

Null-dereference problems have become the bane of object-oriented programming. We and others have found that 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 (like the type `Speaker` of the fields `left` and `right` in Fig. 2) and possibly-null types (written with a postfix question mark, as in `Speaker?`).[1] References of non-null types can be dereferenced safely without requiring run-time checks or proof obligations to prevent errors.

Each object of type `Stereo` is an *aggregate object*. It contains references to other objects which 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 that the `Stereo` object *owns* its speakers. Due to this ownership relation, 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. We provide more details in Sec. 3.

2. ENFORCING SPEC# CONTRACTS

There is a spectrum of possibilities for checking Spec# contracts. One extreme would be to verify all of them statically, another extreme would be to check them all dynamically. Either is impractically expensive. For the former, there are still many interesting programs that we are unable to specify and verify efficiently. For the latter, the run-time overhead is prohibitive. Instead, Spec# makes some checks mandatory—those checks are split between dataflow analyses performed during compilation and run-time checks performed during execution—and the rest are optionally enforced by a static program verifier.

The run-time checker is straightforward: each contract indicates some particular program points at which it must hold. A run-time assertion is generated for each, and any failure causes an exception to be thrown.

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

A type system guarantees that the static type of an expression accurately describes the possible values to which the expression can evaluate at run time. In Spec#, an expression that has a non-null type can never be observed to have a value of **null**. In addition to controlling assignments, this requires controlling initialization [10]. In particular, the type system needs to guarantee that a fresh object doesn't escape from its constructor before the constructor initializes the non-null fields (such as `left` and `right` in Fig. 2) 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 that contracts are *pure*, that is, side-effect free. This ensures that 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.

It would be easy to forbid the use of all side-effecting operations (such as field updates) in the body of a pure method, 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, these effects are not observable when the method returns. Following JML [18], Spec# thus enforces *weak purity*, which forbids pure methods from changing the state of existing objects, but allows 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 both of the explicit contracts and of the implicit contracts set forth by the language semantics (e.g., null dereference and array index out of bounds). It checks one method at a time. If the verification fails, it displays an error message, the location of the error, the trace through the method that contains the error, and possibly a counterexample. Fig. 1 illustrates how the IDE reports verification errors to the programmer.

Our static verifier is sound, but not complete. That is, it finds all errors in a program, but it might also warn about methods that are actually correct. Such spurious warnings can often be fixed by providing more comprehensive specifications. In some cases, it may be necessary 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 run time. Assumptions require special attention during testing and code reviews.

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. The different stages are shown in Fig. 3. The gap between the two is bridged by translating the Spec# program into a much simpler program: we defined an intermediate language named Boogie [1]. 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 that the method might modify (things that are within its *frame*, see Sec. 3), and then assuming the method's postconditions.

Note that static verification does not fully replace testing. Tests are still necessary to ensure the requirements have been captured correctly, to check those properties that are not expressed by contracts, and to check properties ignored by our verifier (for instance, stack overflows).

3. USING INVARIANTS

Spec# performs *modular reasoning*, which is to a verifier

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

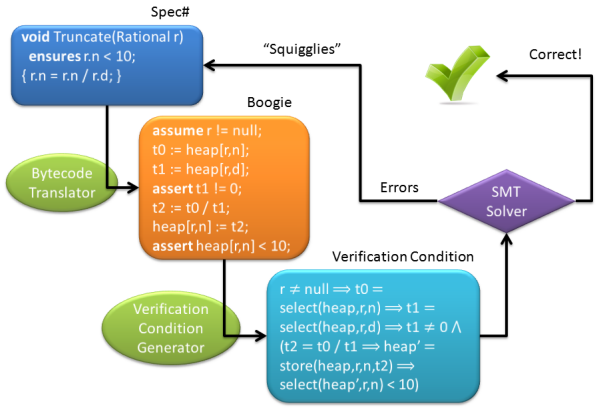

Figure 3: The Verfication Pipeline. The semantics of the Spec# program is encoded in Boogie. The heap is modeled as a two-dimensional array indexed by object references and field names. Within its body, a method's preconditions and type information are encoded as assumptions, and its postconditions and 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, these are mapped back to the Boogie program and then to the Spec# source. Of course, the prover may run out of time or space (not shown) or — in rare circumstances — prove that the program is actually correct!

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. To reason soundly in the presence of object invariants and mutable state, Spec# uses a *methodology* that restricts programs and guides the use of specifications. We give a short overview of this important contribution of sound modular reasoning here; a full explanation is in the Spec# tutorial [20].

A first problem is that there is no *a priori* set of program points where an object invariant can be said to hold. An invariant cannot *always* hold: it is generally necessary to temporarily violate an invariant with later state changes re-establishing it, as we illustrated in Fig. 2 by method `CrankItUp`. It is also not possible to say that an object invariant holds on method boundaries: method calls made within a method may make the object accessible outside the class while it is 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 Fig. 2. Consequently, modifications of these sub-objects potentially violate 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 that the aggregate object's invariants are re-established 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. It 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 using an *ownership* system: the objects of the heap are organized into a collection of tree structures. The edges of the trees indicate ownership, that is, an aggregate/sub-object relation. Roughly, an object invariant can depend only on state that is 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; this prevents method calls on inconsistent objects.

That is the basic approach to specifying aggregate objects in Spec#. However, many object-oriented programs are not only about 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* relations, the methodology uses a notion of *peer consistency*, which says that an object and all its peers are consistent. See the Spec# tutorial 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 may potentially 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 (for instance, 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, it is not possible for a method contract to name these parts directly. Instead, some form of abstraction is needed, yet 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 of those 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()`, but not how the volume is stored in the sub-objects of `Stereo`.

4. SONGS OF INNOCENCE

We started the Spec# project in 2003 as an attempt to build a comprehensive program verification system [2]. Our dream was to build a real system that real programmers can use on real programs to do real verification—a system that "the programming masses" could use in their everyday work. We wanted to explore and push the boundaries of specification and verification technology to get closer to realizing these aspirations. Let us consider in more detail the lay of the land at the time we started the project.

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

Another approach that uses verification technology were extended static 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 (SMT) solvers. To make them easier and more cost-effective to use, extended static checkers were intentionally designed to be *unsound*, that is, they may miss certain errors.

Dynamic checking of specifications has always been done by Eiffel [22], which also pioneered the inclusion of contracts in an object-oriented language. The tool suite for the Java Modeling Language (JML) also included a facility for dynamic contract checking [3]. The strong influence of both Eiffel and JML on Spec# is all too evident.

Our plan to target *real programs* was no doubt our single most influential decision: it has permeated every corner 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. The decision implied a connection with an existing language, so we decided to build our language extensions around C# and the .NET platform. Other well-known real languages with specifications were Eiffel, Java+JML, and SPARK Ada [0]. Like 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 users to an interactive proof assistant. We felt that while programmers need to 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, because Spec# users still need to supply specifications. But all interaction between users and Spec#'s tools takes place in the context of the program and its specifications. The major contender in this space 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 users. It is a long journey for a user to get to the point of writing specifications that lead to effective verification. To give users immediate benefit for any specification they write, however partial, we decided to provide dynamic checking of specifications. Throughout the project, we also worked hard on providing good defaults so that programmers would not be unduly burdened for the most common cases.

Finally, our plan to do *real verification* meant not compromising on soundness. This goal aligned our direction 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 (which give rise to the possibility of call-backs, that is, situations where the caller of a method is re-entered during the execution of the method it called) as well as method framing. To ensure that our verifier would scale to large programs and could be applied to libraries, we also wanted to support modular verification. We started the project with an idea for a methodology that addresses these problems, which gave us a glimmer of hope of building a sound and modular verifier.

Though we aimed for a broad design, we left out several things initially, so that 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 instead focused on partial properties, of the kind that every programmer could write down and for which one might be willing to accept the run-time overhead of dynamic checking. Some other features that we left out of our initial design, like generics, were subsequently added.

In summary, we set out to build a programming system with specifications as an integral component. The system was to blend into existing practices, it was to provide a range of assurance levels from dynamic checking to static verification, and the static verification was to be sound and automatic. Its success depended upon answering a number of scientific questions as well as solving non-trivial engineering concerns. In the next two sections, we look back at how the project fared with regard to these issues.

5. IMPACT

In this section, we summarize Spec#'s influence on researchers and language designers in academia and industry.

5.0 Scientific Results

The main research focus of the Spec# project has been on improving verification methodology by identifying common programming idioms and developing techniques and notations for their specification and verification. We have built a state-of-the-art system and we have advanced the state of knowledge. First, the Spec# methodology supports sound modular verification of object invariants in the presence of multi-object invariants, subclassing, and reentrancy. Second, we worked out some of the difficulties with abstraction features like pure methods. Third, Spec#'s dynamic ownership model allows one to express heap topologies and to use them for verification. Fourth, the Spec# project gained practical experience with a design of non-null types and incorporated flexible object initialization schemes. Fifth, it advanced the foundations of program verification, for instance, by providing a verification-condition generator for unstructured programs. Finally, by providing IDE support and continuously running the program verifier in the background, Spec# has broken new ground in how users work with a verifier. The scientific contributions of the Spec# project have been published in over 30 articles.

5.1 Impact on Academia

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. It uses Spec#'s ownership system to enforce locking strategies. Several research groups use the Boogie verification engine, which was developed as part of the Spec# project [1]. For instance, various Java/JML, bytecode/BML, and Eiffel projects use Boogie as target for their verifiers. At the other end, Boogie's output is now also fed to interactive theorem provers. In addition, we see people encode and verify new logics, for instance region logics, and verify challenging examples, like garbage collectors.

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

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

5.2 Impact on Industry

Initially, we had hoped to convince one of the programming language teams at Microsoft to add Spec#-like features. Not only is this a difficult proposition by itself, but our focus on the Spec# language did not address the fact that .NET is a multi-language platform. We also did not have any support for unsafe code or for concurrency and were battling a perception that verification is relevant only for safety-critical software. Even so, Spec# has influenced other projects in Microsoft Research and in product groups.

HAVOC. The HAVOC tool repurposes parts of the Spec# system to verify low-level sequential systems code written in C [4]. HAVOC has been applied to verify properties of device drivers and critical components in the Windows kernel. A version of HAVOC has also been targeted to find specific errors in a very large code base at Microsoft. HAVOC's recent focus has been to infer specifications for legacy code.

VCC. The Verifying C Compiler (VCC) project [5] is specifying and verifying a shipping code base: Microsoft Hyper-V—a thin layer of software that sits just above the hardware and beneath one or more operating systems. VCC adopts Spec#'s tool chain and methodology. It addresses Spec#'s limits in two dimensions: it includes full functional verification and it verifies concurrent operating system code. For the latter, VCC allows two-state invariants that span multiple objects without sacrificing thread or data modularity.

Code Contracts for .NET. Spec# inspired a new project, Code Contracts for .NET. To avoid the need to get users to adopt a new language (and the need to support it), we introduced a library-based approach. Calls to the *contract library*, which includes methods such as `Contract.Requires`, can be called from any .NET program. Both method contracts and object invariants are supported, although 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 the calls to contract methods, our post-build tools then extract the contracts and use them for both dynamic and static checking. Starting with .NET 4.0, the contract library is now a part of mscorlib, .NET's standard library. The associated tools are distributed through DevLabs, a web site where Visual Studio makes early technology available. We have also investigated how such contracts can be exploited by other tools.

6. SONGS OF EXPERIENCE

In this section, we reflect on our initial aspirations and design decisions.

6.0 Not Being a Toy Language

The fact that we built Spec# as a full-scale .NET language and made a mode for it inside the Visual Studio IDE has had far-reaching consequences. For one, it made the scope of the project large enough to include a wealth of both scientific and engineering challenges. The project shows that it is possible to build a practical verifier of this scale; in fact, given the present availability of SMT solvers and verification engines like Boogie or Why [12], the task of building a verifier is now smaller than it was. Let us take a closer look at some decisions we made and how they fared.

Most importantly, being a full language that compiles to a common platform has increased the credibility of the research. We feel that it has heightened the impact of the research, letting us approach people who, especially at Microsoft, might not have been impressed by a one-off system. The integration into Visual Studio allowed us to do 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 this is how it has allowed us to, through live demos, communicate the vision of our project. Demos aside, if verification will ever make it into the daily rhythm of mainstream programming, it will be through such a design-time interface that provides on-line verification.

Having access to existing programs has two important advantages: it lets you try out your ideas and it brutally reveals problems that still need solutions. It thus both validates research and guides the way to more research problems to be tackled. On numerous occasions, seeing the results of our experiments forced us to support previously ignored features and to 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 into the creation of a new research project. Once most of the system is in place, however, adding or modifying features becomes a larger effort than one would wish. Frequently, we felt that we were not able to move as quickly as we wished. For example, adding a new feature with syntax required changes not just to the parser, but also the rest of compiler, the admissibility checker, the part of the system that persists and recalls contracts in compiled libraries, and the verifier. We also had to put a lot of effort into producing contracts for the existing .NET libraries, by writing 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 thereof is far inferior to product-quality integration, and the Spec# mode in Visual Studio is downright clunky compared to the whiz-bang C# mode. Using our own integration, we were able to provide the programmer with feedback as the program is keyed in, but it also meant that we were unable to take advantage of all of the IDE advances, such as refactoring support, without a prohibitively large engineering investment.

If one were to do the research project again, it is not clear that *extending* an existing language (here, C#) is the best strategy. Not only does it mean having to deal with con-

structs that are difficult to reason about, but it also presents a problem when the base language evolves. For example, for us to migrate Spec# to extend versions 3 and 4 of C# would require more development resources than we have, all for the purpose of supporting features of marginal research return. In contrast, SPARK Ada is built as a *subset* of Ada, which more easily lets a language designer pick features that mesh well with verification and trivially solves the problem of what to do when the base language evolves. But a subset is also problematical, because it makes it more difficult to apply the verifier to legacy code.

A final point is the question of where in the compilation chain to apply verification. The Spec# verifier actually starts with the MSIL bytecode that the compiler produces. This lets the verifier ignore syntactic variations offered by the source language (for example, **`for`** loops versus **`while`** loops) and allows cross-language verifiers to be built. But for some features, like the syntactic support for iterators in C#, it would be much easier to start with the source constructs than having to verify or first reverse engineer the auxiliary classes and chopped-up method bodies that the compiler emits into the bytecode. While one might be tempted to start at the source, we feel that it is the right thing to start with MSIL—otherwise every language would have to write its own verifier. But compilers should annotate the bytecode to make it easy to recover higher-level information.

6.1 Non-null Types

Non-null types have proven to be 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, the non-null types are almost universally liked by users, with the exception entailed by the difficulty of converting legacy code.

We started 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, however, 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.

As a downside, our non-null type system has some holes resulting from the engineering compromises required to integrate the types into an existing platform. A complete system needs support from the .NET virtual machine, for example to ensure that element assignments to arrays respect the covariant arrays of .NET. As another example, 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 we did not implement all of these workarounds. Our hope is that future execution platforms will be designed with non-null types in mind.

Our technology-transfer attempt of non-null types into Microsoft languages left us with a surprise. We had felt that the fruits of our non-null research were ready for prime time. But, as we just described, non-null types do not reap its full benefits in a single language—they need platform support. Interestingly enough, as we described in Sec. 5.2, we had better luck with the technology transfer of contracts.

6.2 Dynamic and Static Checking

Another major goal of our initial design was to support both dynamic and static checking of specifications. Such a design has several advantages. Most importantly, it immediately rewards users for writing specifications, because these will turn into useful run-time assertions. And 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 the user has to learn just one specification language.

Whenever dynamic checking would be too expensive—for example, 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 one still obtains 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, whose sole purpose is to trade a dynamic check for an assumption by the static verifier.

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

6.3 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, which comes at a cost. Our experience shows this cost to be moderately low, which we explain as follows. The effective loop invariant draws from three sources. One source is that the Spec# program verifier automatically infers simple loop invariants. A second source of loop invariants is *loop-modification inference* [8]: Spec# enforces the method frame on every heap update, so the **`modifies`** clause of the enclosing method can therefore be incorporated as an automatic part of the loop invariant. This is the "biggest" part of the effective loop invariant, and it is also the part that no user would want to supply explicitly. Finally, the third source are the user-supplied loop invariants. Because of the first two sources, there are many loops that require no user-supplied loop invariants at all, especially for methods with no postconditions. As one takes steps toward functional-correctness verification, the need for user-supplied loop invariants is likely to increase.

6.4 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 we started the project, others have designed some alternative methodologies [17, 24]. The Spec# methodology has been streamlined for some common object-oriented patterns (e.g., ag-

gregate objects), and it lends itself to concise specifications of programs that fall within those common patterns. However, for programs that use more complicated patterns, the methodology can be too restrictive. For example, this caused us to lose the interest of one otherwise avid user at Microsoft who was trying to verify a large body of code.

The learning curve of the methodology has been steeper than we would have liked, and non-expert users can 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 the users' view. We hope to have mitigated this problem somewhat with our new tutorial [20].

6.5 Miscellaneous

We make four more remarks about our experience.

First, we had set out to build a sound verification system. While we have found sound solutions to fundamental problems of modular verification of object-oriented programs, our implementation is not perfect. There are several unimplemented features and other unfound errors in our implementation and semantic encoding.

Second, the best and most far-reaching single design decision we made in the implementation of the Spec# verifier was to introduce the intermediate language Boogie in between the Spec# program and the formulas sent to the theorem prover. Boogie permits manual authoring as well as automatic translation [1]. This extra layer of indirection allowed us to investigate many alternative design decisions in a lightweight fashion by just 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 front-end for different theorem provers.

Third, a conclusion we have drawn from our interaction with developers is that real developers do appreciate contracts—contracts are not just an esoteric feature prescribed by "Dijkstra clones". Unfortunately, we have also seen an unreasonable seduction with static checking. When programmers see our demos, they often develop a romantic enthusiasm that does not correspond to verification reality. Post-installation depression can then set in as they encounter difficulties while trying to verify their own programs.

Fourth, from the research standpoint, having source syntax for specification constructs lets the program text be concise and usefully descriptive, making clear the important concepts. However, from the adoption standpoint, there are two problems with this approach. One is that the engineering overhead associated with being a superset language is a high price to pay. The other is that we want adoption 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; individual languages can now consider adding convenient source syntax.

7. CONCLUSIONS

Since the Spec# project started, the Verified Software Initiative [15] has organized the verification community to work towards larger projects, larger risks, and a long-term view of program verification. All of our work fits into these aims.

Going forward, we find that Spec# has evolved from a single-language vision into four ongoing prongs.

One prong is the Spec# project itself, which is now entering a new phase: through a new open-source release[2], we hope to see continued improvements of the Spec# programming system from a larger community. We also hope to see more use of Spec# in teaching, especially in light of our new and comprehensive tutorial [20].

A second prong is the Boogie language and verification engine, which provides an infrastructure that is used as an abstraction layer in several verification projects.[3]

A third prong is a new strand of research that attempts 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 the experience with it so far has been very positive.

A fourth prong aims at the mass adoption of specifications in programming. Our language-independent Code Contracts library goes beyond what any single language can do—it puts a specification facility into the platform. Through the associated tool support, we hope this will lead to improvements in the software engineering process.

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

8. ACKNOWLEDGMENTS

Spec# would not exist were it not for all of the research interns and users who helped bring it into existence. We also express our thanks to the early reviewers and anonymous referees of this paper.

9. REFERENCES

[0] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[4] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, LNCS, pages 19–33. Springer, 2007.

[5] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Verifying system-level C code with the Microsoft verifying C compiler (VCC). In *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009.

[6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, Mar.–Apr. 2008.

[7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[2] `http://specsharp.codeplex.com`

[3] `http://boogie.codeplex.com`

[8] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

[9] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE*, pages 258–267. IEEE Computer Society Press, 1996.

[10] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM, 2003.

[11] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM, 2007.

[12] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[14] R. 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.

[15] C. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4):1–8, 2009.

[16] B. Jacobs, K. R. M. Leino, F. Piessens, J. Smans, and W. Schulte. A programming model for concurrent object-oriented programs. *ACM TOPLAS*, 31(1):1–48, 2008.

[17] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[19] K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at `http://research.microsoft.com/~leino/papers.html`.

[20] K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *LASER summer school lecture notes*, LNCS. Springer, 2009. To appear.

[21] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.

[22] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[23] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[24] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.

APPENDIX

A. PROPOSED SIDEBAR: THE SPEC# ECOSYSTEM

Spec# relies on other projects developed within Microsoft Research.

CCI The Microsoft Research Common Compiler Infrastructure (CCI) is a set of base classes that implement common functionality needed by compilers. It takes care of intermediate code generation and provides help with symbol table management, meta-data importing, name resolution, overload resolution, error reporting and so on. It also includes functionality that helps with integration into the Visual Studio development environment. It was first developed as part of the implementation of Comega, but has since been used for a number of other compilers, including the Spec# compiler. Recently, a newly redesigned version of the core parts of CCI has been released as an open source project (see `http://ccimetadata.codeplex.com`).

Boogie Boogie [19] is a verification platform that consists of an intermediate verification language and a tool that generates logical verification conditions. The Boogie language offers a level of abstraction that is suitable for modeling the behavior and proof obligations of a source language. For example, it supports procedures with contracts, local and global variables, structured as well as unstructured control flow, a polymorphic type system, and first-order mathematical definitions. These features make it convenient 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 (like the SMT solver Z3), and makes the results available to users and upstream tools. Boogie was first developed within the Spec# project, but is now employed by several program verifiers, including HAVOC and VCC as mentioned in the article. Boogie has been released as an open source project (see `http://boogie.codeplex.com`).

Z3 Z3 [6] is a state-of-the-art Satisfiability Modulo Theories (SMT) solver. It replaced our use of Simplify [7]. It combines decision procedures for functions, arithmetic, and logical quantifiers. Because of 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 that leads to the failure, and a counterexample with possible values for local variables and heap locations. This information is very useful to provide the programmer with precise and helpful error messages. Z3's website is `http://research.microsoft.com/projects/z3/`.

B. PROPOSED SIDEBAR: GLOSSARY

There were many things that we didn't have the room to include citations for. This is a list of things for which we could at least provide a one-sentence definition and citation for, if that seems desirable. (This list may not contain everything from the paper that it could. We can continue to add to it.)

- Behavioral subtyping
- bytecode/BML
- C#
- CLR
- Eiffel
- Event-B
- GYPSY
- Hypervisor
- IDE
- JML
- MSIL
- .NET
- Singularity/Sing#
- SMT
- SPARK Ada
- Weakest Precondition