The Spec# Programming System: Challenges and Directions

Mike Barnett, Robert DeLine, Bart Jacobs, Manuel Fähndrich,
K. Rustan M. Leino, Wolfram Schulte, and Herman Venter

Microsoft Research, Redmond, WA, USA
{mbarnett,rdeline,maf,leino,schulte,hermanv}@microsoft.com
bart.jacobs@cs.kuleuven.be

Manuscript KRML 156, 30 September 2005.

0 Introduction

The Spec# programming system [2] is a new attempt to increase the quality of general purpose, industrial software. Using old wisdom, we propose the use of specifications to make programmer assumptions explicit. Using modern technology, we propose the use of tools to enforce the specifications. To increase its chances of having impact, we want to design the system so that it can be widely adopted.

For a programming system to be adopted widely, we think that it must:

- build on a widely used object-oriented programming language; in our case C#;
- build on existing infrastructure and allow interoperability with existing code, here the .NET runtime.
- fully integrate into an development environment; in our case the Microsoft Visual Studio environment.
- build on a teachable and sound methodology; in our case a revised design-by-contract methodology;
- include tools that enforce the methodology; in our case this includes type checking, easily usable dynamic checking, as well as high-assurance automatic static verification;
- support a smooth adoption path whereby programmers can gradually start taking advantage of the benefits of specification;
- be moderate; we added only a few constructs to C#, and soundness is guaranteed only as long as the source comes from constructs that are under our control.

In this extended abstract, we give an overview of the Spec# programming system, the rationale of its design, and a sketch of some open problems. Spec# is currently under development at Microsoft Research, Redmond.

1 The Language

The Spec# language is a superset of C#, an object-oriented language targeted for the .NET Platform. C# features single inheritance whose classes can implement multiple

interfaces, object references, dynamically dispatched methods, and exceptions, to mention the features most relevant to this paper.

Spec# adds to C# type support for distinguishing non-null object references from possibly-null object references [12], method specifications like pre- and postconditions, a discipline for managing exceptions [18], and support for constraining the data fields of objects [0]. While conceptually simple, all have some complicated consequences. Next, we give an overview of each feature.

1.0 Non-Null Types

Many errors in modern programs manifest themselves as null-dereference errors. We have opted to add type support for nullity discrimination to Spec#, because we think types offer the easiest way for programmers to take advantage of nullity distinctions. Here is a list of challenges with some solutions.

Initialization of fields and arrays For type safety, any time a variable of a non-null type is read, the value obtained must be non-null. Without further restrictions, this is not guaranteed for instance fields, static fields, and array elements. Spec# offers a simple syntactic solution for instance fields, which ensures that they are initialized before the object being constructed can be accessed. Handling non-null static fields requires ordering restrictions on the initialization of classes. Spec# supports non-null static fields, but the initialization restrictions are checked only at run time. Supporting non-null element types of arrays is tricky, because there is no language construct that marks the end of the initialization of an array's elements.

Language interoperability The .NET platform supports multiple languages, not all of which may be required to support non-null types. This raises many problems, like calling from such a language methods whose in-parameters have non-null types. One could disallow such calls, but we want to encourage libraries to be updated with explicit non-null information without forcing changes in the clients. A better solution is to add non-null types to the virtual machine and to adapt the just-in-time compiler, but this would greatly impact all virtual-machine implementations.

Arrays with non-null element types also require changes in the virtual machine, because of array covariance. For this reason, we do not support them in Spec#.

Stability of non-nullity One must support some form of down-cast of an expression from a maybe-null type to a non-null type. The Spec# compiler infers the non-nullity of local variables by performing a data-flow analysis that takes into account type casts and tests for null. However, this is problematic for fields, whose values may be changed between a test and a use by calls or by other threads about which the compiler has no information.

Non-null types with various degress of type soundness have also been used in program checking systems like LCLint [11] and are being incorporated into the object-oriented language Eiffel [20]; see Fähndrich and Leino's paper [12] for more information about previous work.

1.1 Method Contracts

To allow programmers to capture more complicated properties, which may involve more than one variable, we advocate using general specifications instead of further enrichments to the type system. Like for example Eiffel [19], Spec# supports pre- and postconditions of methods; these use ordinary side-effect free boolean expressions.

Inheritance Pre- and postconditions are inherited in method overrides. Spec# allows overrides to declare additional postconditions. However, Spec# does not allow an override to weaken the precondition, despite the fact that this would be sound. Here's our justification: There is a mindset in .NET (and in Java) that the misuse of language primitives (like indexing an array outside its bounds) and methods always be detected and reported immediately when the violation occurs at run time. It would seem to go against this mindset to allow an override to eliminate this detection. Actually, though a fine syntactic convenience, there is never a need for weakening preconditions, because a subclass can always define a new method with a weaker precondition and let the old method call the new one.

Because of interfaces, there is a limited form of multiple inheritance. In these cases, a method implementation can inherit its specification from several sources. However, to avoid weakening preconditions, Spec# allows this multiple inheritance only in certain situations where all the inherited preconditions are the same. Actually, this is not a semantic restriction in Spec#, because a class can provide different implementations for multiply inherited methods.

Frame conditions To reason statically about a call, one needs to know what variables the callee may modify. Specifying this *frame condition* is difficult, because of information hiding: the frame condition must say that the caller's variables are unchanged and must allow the implementation's variables to be changed. But these variables are in general not visible to both the caller and the implementation. Spec# uses syntactic and semantic abstractions to address this problem, but we do not yet have enough evidence to evaluate whether or not this particular solution is the right one.

Evaluation of contracts We take the view that contracts should contribute to the specification of programs, but not to their effects. Therefore, Spec# insists that expressions used in contracts are side-effect free. We do allow procedural abstraction in contracts, as long as any procedure (method, rather) called also is side-effect free, that is, *pure*. A syntactic notion of purity is overly restrictive, so we are exploring notions of *behavioral purity*, but don't have all the answers yet [4, 22].

The fact that contracts are given by expressions of the language raises the problem that the evaluation of those expressions can fail. Spec# chooses to hold the specification writer accountable for such errors. That is, contracts are enforced to be totally defined.

1.2 Class Contracts

Invariants are key to describing the correctness of programs, and Spec# allows classes to declare invariants that describe the internal consistency of the state of their objects.

However, the problem of maintaining invariants that span several objects has been under-studied. Two central problems are delineating when an object invariant holds (which is made difficult because of re-entrancy) and controlling changes to sub-objects (which is made difficult because references to these sub-objects may be leaked).

Spec# uses a sound methodology that addresses both of these central problems (and the problem of writing frame conditions) [0,15, 3, 16]. Spec# provides a block statement that delineates where object invariants may temporarily be violated, and it uses ownership domains to confine references.

1.3 Concurrency

Multi-threading introduces the possibility of race conditions and deadlocks. This makes data consistency even more difficult to achieve in a multi-threaded program. We feel that a programming methodology should permit extensions that cover concurrent programs, too. We have formulated an extension of our object-invariant methodology for multi-threading [13]. It ensures mutual exclusion on ownership domains and maintains data consistency. But our extension still has several shortcomings, including: it does not check for deadlocks and its locking is sometimes too coarse-grained.

1.4 Data Abstraction

More elaborate specifications of programs require a way to present a view of objects that abstracts away from implementation details. We find that this form of data abstraction already exists in many .NET programs in the form of *properties* (which are essentially parameter-less methods). If a property is a pure function of the state of an object and its immediate ownership domain, then we call it a *model field* [7, 14, 17, 21]. We are hopeful that one can formulate the definition and use of model fields in a way that is amenable to static program verification.

2 System Architecture

Architecturally, the Spec# programming system consists of the compiler, a runtime library, and the static program verifier. Spec# has been integrated into the Microsoft Visual Studio environment. For example, violations of the non-null type system are indicated by “red squigglies”, specifications of methods (including any specifications on library methods) are available in tool tips. The static program verifier runs continuously while editing the program and interactively produces red squigglies for semantic errors, but it can also be run as a standalone tool.

2.0 Levels of Checking

Spec# provides three levels of checking.

The first level of checking is provided by the type checker, which runs as part of the compiler and which must accept the program before any code is emitted. The type

checker is stronger than some other traditional type checkers in that it is sometimes sensitive to data flow.

The second level of checking is provided by compiler-emitted run-time checks. These checks are always emitted, but we do provide some compiler options that disable them, because development organizations sometimes feel they have reached a point in the development cycle where they are willing to risk not having the checks and would rather gain performance. The run-time checks enforce many contracts, but do not enforce the entire Spec# programming methodology. For example, frame conditions (modifies clauses) are not checked at run time and, by default, ownership domains are not enforced at run time.

The third level of checking, which is optional, is provided by the Spec# static program verifier. It enforces all contracts and the entire Spec# programming methodology, except assume statements, which are provided for the specific purpose of introducing a run-time check for programmer assumptions that would take great effort to prove statically.

2.1 Contract Persistence

The Spec# compiler preserves the specifications as metadata in the same binary assembly as the compiled code. This enables reuse of specifications across tools.

To enable clients of a legacy library to be verified, such a library needs to be retrofitted with specifications. If the library cannot be converted into Spec#, we support the compilation and use of *out-of-band contracts* in shadow assemblies. These give the illusion that the specifications were declared in the library itself.

2.2 Static Verification

From MSIL (the .NET bytecode), Spec#'s static program verifier (whose codename is Boogie) constructs a program in its own intermediate language, BoogiePL [10]. BoogiePL is a simple imperative language with procedures. BoogiePL also supports the introduction of uninterpreted function symbols and axioms, which makes it suitable as a stepping stone in program verification. In fact, all of the Spec# to be verified is translated into BoogiePL, including the axiomatization of the Spec# type system, *etc.* From the BoogiePL program, Boogie infers loop invariants using abstract interpretation [8, 9, 5, 6] and generates verification conditions (logical formulas that are valid iff the program is correct) that it passes to an automatic theorem prover [1]. Counterexamples reported by the theorem prover are translated back into error messages about the source code, which are reported to the user.

Technology for abstract interpretation and theorem proving exist. We feel that the work in this area has reached the kind of maturity where what is needed for Spec# consists mainly in adapting and tuning existing technologies for the verification task at hand. One difference from previous work is perhaps the emphasis on the heap in Spec# programs. There is also room for improving the combination of various abstract domains as well as exploring the combination of abstract domains and decision procedures.

In the verification community, various standard or canonical formats have been useful for interoperability, substitutability, and evaluation of tools (for example, the primitive DIMAC format for SAT formulas). Our position is that BoogiePL is a good candidate for playing that same role in the space of verifying programs. Boogie can be invoked not just on MSIL assemblies, but also on BoogiePL programs directly. This means that other researchers can reuse the abstract interpretation and verification-condition generation in Boogie.

3 Conclusion

We are excited about the Verification Grand Challenge working conference. We see the largest remaining challenge to be the formulation of programming methodology that allows modern programs to be specified and verified, as well as the serious engineering effort to build such a programming system.

The Spec# system, including the Boogie program verifier, can be downloaded from `http://research.microsoft.com/SpecSharp`.

References

0. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004. www.jot.fm.
1. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2005*. ACM, September 2005.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
3. Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Seventh International Conference on Mathematics of Program Construction (MPC 2004)*, Lecture Notes in Computer Science. Springer-Verlag, July 2004.
4. Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. Proceedings, 6th workshop on Formal Techniques for Java-like Programs, June 2004.
5. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In Radhia Cousot, editor, *VMCAI 2005*, volume 3385 of *Lecture Notes in Computer Science*, pages 147–163. Springer, January 2005.
6. Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants. In *Proceedings of First International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005.
7. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, 2005.
8. Patrick Cousot and Rhadia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.

9. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.
10. Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.
11. David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In David S. Wile, editor, *SIGSOFT '94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM SIGSOFT Software Engineering Notes 19(5), pages 87–96, December 1994.
12. Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, volume 38, number 11 in *SIGPLAN Notices*, pages 302–312. ACM, October 2003.
13. Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In Bernhard K. Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods*, pages 137–146. IEEE, September 2005.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science, 2003. See www.jmlspecs.org.
15. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.
16. K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Formal Methods 2005*, July 2005.
17. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
18. K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In Jorge R. Cuellar and Zhiming Liu, editors, *SEFM 2004—Second International Conference on Software Engineering and Formal Methods*, pages 218–227. IEEE, September 2004.
19. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
20. Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In *ECOOP 2005*, pages 1–32. Springer, July 2005.
21. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.
22. David A. Naumann. Observational purity and encapsulation. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005*, volume 3442 of *Lecture Notes in Computer Science*, pages 190–204. Springer, April 2005.