



Hanselminutes

Hanselminutes is a weekly audio talk show with noted web developer and technologist Scott Hanselman and hosted by Carl Franklin. Scott discusses utilities and tools, gives practical how-to advice, and discusses ASP.NET or Windows issues and workarounds.

Text transcript of show #110

April 22, 2008

Microsoft Research: Spec#

Scott sits down with Mike Barnett and Rustan Leino of Microsoft Research and talks about the Spec# programming language. The compiler enables Design By Contract and extends C#. The team needs your help to get these features in the next version of C#!

(Transcription services provided by [PWOP Productions](#))



Our Sponsors

 **telerik**
deliver more than expected
<http://www.telerik.com>

 **nsoftware**
<http://www.nsoftware.com>

NET 
DEVELOPER'S JOURNAL
<http://dotnet.sys-con.com>





Lawrence Ryan: From hanselminutes.com, it's Hanselminutes, a weekly discussion with web developer and technologist, Scott Hanselman, hosted by Carl Franklin. This is Lawrence Ryan, announcing show #110, recorded live Tuesday, April 22, 2008. Support for Hanselminutes is provided by Telerik RadControls, the most comprehensive suite of components for Windows Forms and ASP.NET web applications, online at www.telerik.com. Support is also provided by .NET Developers Journal, the world's leading .NET developer magazine, online at www.sys-con.com. In this episode, Scott talks with Rustan and Mike from the Microsoft Research Group about the Spec# programming language.

Scott Hanselman: Hi, this is Scott Hanselman and this is another episode of Hanselminutes. I'm sitting here at the ALT.NET Conference in Seattle, Washington, with Rustan and Mike from the Microsoft Research Group working on a product called Spec# that they've just demoed to the ALT.NET crowd here. Thank you, gentlemen, for sitting down with me today.

Rustan Leino: Thanks for having us.

Scott Hanselman: So, Spec#, it says # so that must mean it has something to do with C#, possibly the .NET Framework. What does Spec# do?

Rustan Leino: Spec# is a superset of C#, of the C# language and it adds to the language specifications, which is where the language gets its name. The specifications are contracts like pre-imposed conditions and we also enhance the type system with non-null types, for example, and with those contracts you can record your design decisions in the program text and have them be checked by various tools.

Scott Hanselman: So, you are superset of C#. So, you're saying that "here are some keywords and some things that are not in C# that we've added." You enhance the language itself.

Rustan Leino: That's correct. So, we add a few keywords, a few new constructs here and there.

Scott Hanselman: Are you your own compiler or are you a post compiler?

Mike Barnett: We're a full pledged compiler, so it's not a source to source transformation. It takes the surface syntax and compiles it all the way down to ILL and creates valid .NET assemblies just as VB compiler or the C# compiler do.

Scott Hanselman: So, did you have to recreate work that was already done by the C# compiler?

Mike Barnett: Yes because it's completely independent. I mean it does a full parsing and code

generation and the codebase is absolutely separate from the C# compiler.

Scott Hanselman: Is that a good thing or a bad thing?

Mike Barnett: It was a good thing for quickly getting a compiler up and running. It was a bad thing in the sense that now we are a superset of C# 2.0 and not a superset of C# 3.0.

Scott Hanselman: Ah, I see, but of course the intent is to make this something that is useful for everyone because you're trying to take the notion of a specification and add it to all .NET languages.

Mike Barnett: That's right, so one of the things we're looking at now is a non-language-based solution and we use Spec# as a way to sort of paint a picture of the future of what it could look like if a language was to fully embrace the idea of design by contract to make specifications a first class citizen, but we designed Spec# in a way that has made it possible to peel off layers and potentially get those layers implemented in all .NET languages.

Scott Hanselman: Oh. So, you've added a number of keywords to C# in order to allow this to then design by contract concept. What are some of the keywords and explain how it would decorate one of my methods with these keywords?

Rustan Leino: The first one that one runs across is not the keyword at all, but just another symbol, which is the exclamation point, the bang.

Scott Hanselman: The bang.

Rustan Leino: If you write the reference type like T or string or object and you follow it by bang, making it string bang or object bang, T bang, what you're saying is that that type is the type that holds a reference that is not null and that's something that you tend to use all over. So, in the Spec# type system, we allow programs to use non-null types and then the type system, the compiler will make sure that just as you use types elsewhere in the program that you don't give a Boolean and a string is expected here to also does that for non-null entities.

Scott Hanselman: This is a kind of an assertion where I'm making a declaration that type T shall never be null.

Rustan Leino: That's correct. That is a kind of assertion like that or some people even call that a contract as well.

Scott Hanselman: Okay. So, I'm strengthening, I'm constraining the system such that the system cannot get into a state, which is not what I want.



Rustan Leino: That's correct. For the non-null types themselves, we also support a mode where you switch the defaults where the types stand for non-null types and if you want the nullable versions, you do like you do with value types in C#, which is add a question mark afterwards.

Scott Hanselman: Right, so people are familiar with being able to say something like `int?` or suddenly a value type, which for many years has not been nullable, it becomes nullable.

Rustan Leino: Right, exactly.

Scott Hanselman: And I can do the inverse now with class `Person`, which would initially start out null if I said `Person P = null` and now if I said `Person bang`, I can't assign null to it.

Rustan Leino: That's right.

Scott Hanselman: It will never be null within a scope of the project.

Rustan Leino: Exactly, right. So, that's the simplest keyword or thing that we're adding to the language, simplest at least syntactically. Then the other things that you see next are preconditions and post conditions and they are written with the `requires` and `ensures` keywords, which fall all the signature of the method, so you would write `void m` and give the parameters and the close parenthesis and then you would say `requires` and you would give a Boolean condition that says under which conditions the method is allowed to be called.

Scott Hanselman: Okay. So, trying to visualize this, remembering that we've got people commuting in an audio world and we don't have any video to show them, I've created a method that takes a string and an integer and before the opening curly brace but after the method signature, I'm going to say that it is the case that this string and this integer meet these conditions if one is going to be allowed to called this method.

Rustan Leino: Right. For example, you might say that the integer is less than the length of the string as an example.

Scott Hanselman: Interesting. So, when I express a method contract right now, the extent of that method contract is that it takes these types and I can expand that contract by adding overloads and a lot of people have asked for things like optional parameters and C#, but you're tightening the news as it were and saying that not only might I say that this string must be this length or that this integer must be below this value, but that they can interrelate to each other.

Rustan Leino: That's correct. So, what we do with types typically just express to something about each variable independently and with the contracts like pre-imposed conditions, you can very, very easily and naturally constrain several variables at one time.

Scott Hanselman: Now, what about the result that would be coming back because you made the comment to avoid `m`, but let's say that I take a string and an integer, but I'm going to be returning some value, another string perhaps.

Mike Barnett: `Spec#` has added `contracts` depending keyword result and that keyword can appear in a post condition and `ensures` clause, which guarantees the callers of the method some condition involving the return value. So, it stands for the value that's being returned from the method or the condition.

Scott Hanselman: Ah, so just like when I'm creating a property and I'm doing a property setter and I have this keyword called `value` that is contact specific returning to the value that's coming in to the setter, this is a keyword that refers to what's exiting.

Mike Barnett: Exactly.

Scott Hanselman: Now, I could make post conditions and pre-conditions now by putting in a debug that assert at the beginning or debug that assert as I exit, but why would this be more desirable to use `Spec#` for that, particularly on the exit case?

Mike Barnett: The crucial difference is that if you use `Debug.Assert` inside of your method body, it is something that is visible to you and to the program only internally inside of that method. Callers of that method are not able to take advantage of the fact that you happen to have put those conditions on the code. If you use `Spec#`, then the contracts are visible to all callers of the method and the `Spec#` tools can be applied to those callers to guarantee that the caller will not violate the preconditions of the method being called and can enjoy the benefits of what the post condition guarantees them. For instance, if you call `string.concat`, there's not necessarily a guarantee that what comes back is non-null, but if `concat` had a post condition or said the result is non-null, then you would know that you can freely de-reference the result that comes back from that method without having to do a runtime check to make sure that it was non-null.

Rustan Leino: To add to that, as an analogy, you might consider making all of your methods take object as parameters and that is that the type of all of them would be object and inside of your method, you would immediately cast them to an integer or a string or whatever it is that you want, but by instead making them part of the method signature, you're telling callers that that's what you expect. In a similar way, by instead of using `Debug.Assert` inside of your



method body, you put pre-imposed conditions on the method. You're telling something to your caller.

Scott Hanselman: That is a really interesting way to put it. Just by having the type signature there, you are constraining that contract. I could certainly have a method that took an array of object of indeterminate length and where each object is of indeterminate type and unfortunately I've seen method signatures like that in the wild that returned then an array of objects and I have really no way of knowing other than this is a method called m and I'm constraining that by adding something as simple as types. So, we continue to tighten things up, so we're saying that the domain of what this method is responsible for is smaller and smaller and smaller. Does that make it easier to test?

Rustan Leino: First of all, it makes the requirements clear what is expected of the caller and what is expected of the implementation, but it also means that when you analyze the implementation, which you could do either by testing, which is how it's strictly done, or by some other tools like for example our static verifier then we only need to consider those input states that satisfy those pre-conditioned constraints.

Scott Hanselman: So, I'm hearing you have runtime checking, which would be the equivalent of what I was saying of doing a Debug.Assert, so that's happening at runtime, but this is the importance of this being static, you have tightened up the message signature such that callers of that method signature are aware of that contract.

Rustan Leino: Right. We want to support dynamic checking because it's an easy way to get into the using contract. If one adds a little bit of code just like you would add a Debug.Assert somewhere and it's going to check some condition when you're running it and at the same time you're recording your design decisions in the code, but we want to go a step further. So, in our research, we have spent a good bit of effort on trying to statically verify these things. What we do there technologically is we take the program and convert it into a mathematical formula, a very large mathematical formula, but it's typically mathematically shallow, meaning we're not trying to prove for math's last conjecture or something like that. So, what we do, but there are lots and lots of details to check, so we pass that to an automatic theorem prover, which then analyzes the mathematical formula to see if it's a valid logical formula and if it is, that means that the program is correct. If it isn't, the theoremprover will return to us some mathematical counter example that shows that it's not the mathematically valid formula and then we have kept enough information so we can take that mathematical counter example and bring it back into an error message that the .NET programmer will understand, things like here you're calling a method and you don't

satisfy the precondition or here you're trying to index an array outside its bounds.

Scott Hanselman: Now, previously on this podcast, I had spoken 'Peli' de Halleux and Nikolai about their product Pex and they had spoken of the way that they had layered it such that they could peel off pieces and they spoke of the solver that they pass information into. Is this in fact the same component?

Rustan Leino: In fact, it is the same one. The particular theoremprover is called Z3 and it's also developed at Microsoft Research.

Scott Hanselman: Interesting. So, there's a relationship between the kinds of things that Pex is trying to solve, which is to prove that code will run as it is written and that it appropriately meets the Spec and what you guys are trying to do.

Mike Barnett: Absolutely. I would say all formal tools like Pex and Spec# functioned by treating a program as a mathematical object, which it is because it's a set of formal symbols, which have to be that way so that the computer can understand it and execute the program and then you make progress by using different mathematical theories to analyze properties of that program.

Scott Hanselman: Now, a lot of talk has been made lately of F# and the notion of a functional language being more easily mathematically provable by its very nature. One can say that this is proven to work thusly. Are we trying to take C# in that same kind of a direction or would it be more appropriate to simply use a language that was provable by its nature?

Rustan Leino: If one looks at the research that has been done for program verification, most of it has been done for imperative programs, not so much with a functional language, which is unfortunate and we, because functional languages start off with something that is, well, as you perhaps thought of it as more correct, it's more rigorous in some ways, you know more things about the things in the program, but there are still many difficult issues in trying to apply program verification even to functional programs, but what we're trying to do from the Spec# perspective is we find many places where one would like to specify that a method has no side effect and then we give the opportunity to say that it is a pure method or if you want to say that the class is something that once its instances are constructed, they don't change. You can mark that class to be immutable and those things do help in the static verification of the programs and in addition, the annotations that we give additional documentation to the program of course.

Scott Hanselman: Interesting. All right. Well, I'm just going to take a very brief moment and we're going to thank our sponsors and we'll come right back.



Hi, it's Scott here from another place and time. I hope you're enjoying the show so far. I apologize for interrupting it, but I wanted to let you know that assembling a podcast like this every week isn't free. Certainly, the bandwidth bill crushes us every month, so I want to let you know that this show is sponsored by Telerik. They make the show possible and make pretty cool products as well. For example, if you're trying to build a web 2.0 AJAXy application, trying to use the web 1.0 components is kind of difficult. You got to get the next gen stuff if you're trying to build the *NextGen* websites and that's exactly what the folks at Telerik have got in their new upcoming product, which is codenamed RadControls Prometheus. It's a big pack of web controls built entirely on top of the Microsoft ASP.NET AJAX stuff that you already understand. It's going to give you a lot of performance interactivity in your next project. They mirror the ASP.NET AJAX API so development is really straightforward. Client scripts are shared. Loading time is pretty fast. You set a couple of properties. You can even bind to Web Services for really efficient operation. The new RadEditor for ASP.NET AJAX loads up to four times faster than before and the RadGrid will do thousands of records in milliseconds, but of course it's better to try these things for yourself, so you can visit telerik.com/aspnetajax and download a trial. Thanks a lot for listening and we'll get right back to the show.

So, this notion of side effects and an object is changing what it maybe shouldn't change or entering a state in which it isn't appropriate, I saw that you had a keyword for invariant.

Rustan Leino: Correct. When people think of contracts, they typically first think of pre-imposed conditions, but the very important aspect of a contract is also the invariant, which states properties that you intend to hold of the steady state of a data structure, that is, the invariant says what the internal consistency of your data structures are. To verify a program statically, you need to have invariance and invariance are difficult to deal with because they are not entirely invariant. They do change, that is, there are some points in your program where the invariance do not hold and that has probably been one of the largest scientific contributions that we've done in the Spec# research project to try to figure and wrestle with when do variants hold and what do we do when they don't hold so that other parts of the program don't rely on them holding at that point. So, we do allow invariants. The invariants are checked at the end of the constructor and we also from the runtime perspective, we only check them at the end of exposed statements, so we have an exposed statement in Spec#, which is like a block of code. It looks a bit like a lock statement in concurrent code and it says that within the exposed statement, you're allowed to modify the variables, the invariant might not hold throughout that, but at the end of the

exposed block, we check that the invariant holds again.

Scott Hanselman: Oh, interesting.

Rustan Leino: So, statically, in the static verifier, we also check the invariants at those points, but in addition if you modify state outside of an exposed block, we check that every assignment that you do outside an exposed block will indeed maintain the invariant.

Scott Hanselman: Just to make sure I am hearing you correctly, the word you're using is expose, E-X-P-O-S-E.

Rustan Leino: That's correct.

Scott Hanselman: So, for the moment, for the time that I am in that block, I am exposed, I am in states that may very well be inappropriate, but I'm calling it out explicitly and saying that by the time I leave, I best be back in my appropriate state.

Rustan Leino: That's correct.

Scott Hanselman: Interesting. I really like the words that you've chosen. I find that when I'm, you know, in object-oriented programming, certainly naming is everything and if you can name it appropriately and that mean it feels correct, it feels natural that it will just allow you to use it in a comfortable way and I see that you've got assume, requires, ensures, expose, they're all words of similar length and they all feel that they're kind of in the same family and it seems like a natural extension to the language.

Rustan Leino: Thank you. Keywords are important than sometimes we'd wrestle with them in which we had better ones, but that's our current set.

Scott Hanselman: So, fundamental to what Spec# is trying to accomplish is this notion of really encouraging design by contract and from my point of view, the most obvious contract that we have available to us in C# is this notion of an interface. It's a very high level construct that I can say, "I want you to be constrained to this interface, implement this interface," but that's just a series of methods that take certain parameters of certain types. Can I apply these kinds of Spec# constructs to an interface to say, "Not only do I want you to look like this, but I wish that you behave like this."

Mike Barnett: Exactly. Spec# allows all of the same constructs we've been talking about at the method level and interface methods as well and it's precisely to give interfaces behavioral descriptions that then every implementation is obligated to live up to and it allows the promise of component programming where the only thing you know about a



component is its interface to have a semantic base as well as just knowing that you've conformed to the method signatures.

Scott Hanselman: Yes. In retrospect, looking at an interface and saying that syntactically, this is anywhere near appropriate of what I need to get my job done when the semantics are really transmitted out of band where out of band might be a Word document. This is a completely inappropriate way to pass semantics for interface, but you're saying I can actually impose this on things that have yet to be written. That seems kind of miraculous. That seemed kind of creepy almost. You're saying that someone can come in totally later and implement that interface and you're going to walk back up to the interface and see and confirm that I am conforming to the requirements, to the specification.

Mike Barnett: As long as that eventual implementation is written in Spec#, then yes, the Spec# compiler will make sure that that implementation conforms to all the requirements that the interface specifications spell out. In addition, because Spec# is an object-oriented language, the same holds true for subtypes and that's crucial in object-oriented programming because if the static type of a parameter is a type T, when you receive that type at runtime, it could be a subtype of T. It doesn't necessarily have to be a T itself and so contracts in virtual methods are also binding upon all overrides of these methods and that's the way you can guarantee the robustness of a system that you write the system today and it's robust against future evolution or is more robust against future evolution because you know that whatever the eventual subtype is, it will conform to the contract that you know about the static type.

Scott Hanselman: And this applies to even abstract base classes which is another kind of an interface.

Mike Barnett: Yes, exactly. The same exact details apply to abstract base classes.

Scott Hanselman: So, what about exceptions? Now, I'm starting to feel like there's something like checked exceptions in Java because people have always asked for when it comes to C#, they're saying, "Where are my checked exceptions?" Maybe tell us what checked exceptions are for our listeners who may not be familiar with that technology.

Rustan Leino: Checked exceptions are ones that the compiler will look at how they can flow in the program, so that you can make sure that there are some handler for the checked exceptions.

Scott Hanselman: So, it's actually a way of enforcing. I'm going to call method foo and internally somewhere it could throw an argument null exception,

but currently in C# there's no way for me to know that and there's also no way to force the caller of foo to have a try catch block.

Rustan Leino: That's exactly right. So, if you want to track that in the compiler and force the caller to either catch it or admit itself that it might let one of those exceptions through, then you would use a checked exception. Now, checked exceptions, there are certain kinds of conditions in your program where you really would like to use a checked exception. An example might be a socket closed exception where the programmer really needs to be aware of the possibility that the socket might close, but you don't want to check it on every call, that is, you don't want a return code from every call and have to check that into the call.

Scott Hanselman: Right. The argument against checked exception is that suddenly it litters my system with try catch blocks or explicit calling out that I don't care that that's going to throw an exception.

Rustan Leino: That's right and in the cases where checked exceptions are appropriate, you would have a block of code that operates on some object or objects and then you would have a catch block at the end and in those cases, the programmer really would like to know that the exception is not missed and that's what checked exceptions give you. Now, if you decide then to let such an exception rip through, then you have to admit to that and then it's the next caller in line that gets to deal with it, but this is different from the example you gave a minute ago was the null owner exception, which we consider to not be one of these checked exceptions because it's not something that a caller can reasonably do something with. That is, if you get one of those exceptions or an out of memory exception, sometimes you can recover from that or a CLR internal error exception or something like that. Those things could happen and if they occur, there's something disastrous that has happened in your program and you don't really expect to catch them except maybe at the backstop in your main program.

Scott Hanselman: So, there are different classes of exceptions, those ones that I can do something about and those ones that are truly exceptional, which would be not appropriate to call out explicitly.

Rustan Leino: Correct and that's why we give you the possibility to make checked exceptions for when you want those and leave the other ones, those uncheck ones and we can then do more tracking in the compiler of the check ones.

Scott Hanselman: So, if I had a giant library, maybe even a base class library like the BCL full of thousands and thousands and thousands of methods that I would want to be able to call, now that I've got the Spec# mindset, I would really say, "Oh, well,



gosh, I hope that the base class library, the .NET Framework system.* was instrumented with all of this good information." That's an example of a huge chunk of C# code, not Spec# code and certainly I don't think that they're going to convert that to Spec# anytime soon. How do I get the benefits of Spec# when I don't have a Spec# library?

Mike Barnett: We have created a set of what we call out of band contracts for the entire base class library. Spec# compiler can be run in a special mode where it will compile a file that contains no code, but only the contracts and marks the resulting assembly such that rest of the Spec# tools look glued together to the original runtime assembly, such as MS core loop and the out of band contract for MS core loop so that it looks inside the tool as if the contracts have been there from the beginning. Now, in one sense, that's a lie because the contracts weren't there and so we created these contracts. We think they're reasonable, we think they're accurate, but it would be so much better if the actual library itself came with its own contracts that would guarantee by having been written in the code in the first place and we have been exploring for several years now with different product groups in Microsoft to try to make progress on that front and be able to provide a mechanism where the library providers will provide their own form of the contracts.

Scott Hanselman: Interesting. We have out of band documentation, we have PDB files, and then we could potentially have these contract files, certainly would be ideal if these were built into the language. It seems to me like I would want to start pressuring Anders, the inventor of C# and maybe the listeners would want to know, what's the best way they can get Spec# built in. You guys are Microsoft Research, right? You're not a product group. You're doing this to push the concept forward, but I assume you've met with Anders and you've told him about this several times, but he's got a lot on his plate. How can the listeners get involved? How can we get people to force Anders to listen at this great idea and get this built into C#?

Rustan Leino: Well, every time we've talked to Anders, he's been quite supportive of both the goals of the Spec# project and the things we've been able to accomplish, the tools we've produced. He has indicated that the important thing is to know that there are users out there that want it and that can take advantage of it and so I would say for any listeners interested, they should download Spec#, which is publicly available for free from the Microsoft Research website and use it and see if it's useful for their code and then make noise about it. They may either send email or post on blogs or tell their friends.

Scott Hanselman: Present at user groups?

Rustan Leino: Exactly.

Scott Hanselman: Whatever it takes, absolutely. So, in conclusion here, I want to make sure I understand the scope of this because it's not just runtime type checking. There's this fundamental notion of being able to statically prove that this is correct. When I saw your demo earlier, I got to see little green squiggles and we've seen squiggles before in Visual Studio. I read squiggles that I usually compare to being like spelling errors, these are syntactical issues, but I have made the comment to you that green squiggles felt like a grammar issue, but you said that was wrong headed because you can extend this grammar.

Rustan Leino: Right. The wonderful squiggles that we're used to in Microsoft word, both the red one for spelling mistakes and the green one for grammatical errors are quite similar to what we provide in that the squiggles come up at design time, that is, as you're sitting there typing your programming and the squiggles appear and go away and as you go along. The difference is that with our squiggles, when they appear, there are ways that you can suppress them, that is, they warn you about some condition. In Microsoft Word, if you've made a grammar mistake, you can fix it, but in some cases the grammar checker is just not clever enough to do what you want it to do, then you can just turn off the check, but in Spec#, you can do something better which is that you can explain to it why this is correct. For example, if you use one of your parameter as an array index perhaps inside of your method, then you get a complaint about it, then maybe the way to do it to tell the checker that you're still doing the right thing is to add a precondition to your method that says, "Well, this is not my responsibility. It's the responsibility of the caller to give me a good value," and then the green squiggly will go away, your code remains as it is, but now you've made that contract explicit.

Scott Hanselman: Interesting. Fantastic. So, people can go and download Spec# now. They can play with this now. They can instrument their code. You had mentioned that there were two different modes for this? There was a C# mode and a Spec# mode?

Rustan Leino: Right. The standard mode that we use inside, for example, our program itself which is written in Spec#, that's maybe about 65,000 lines of Spec# right now, is to use the Spec# mode in Visual Studio. Unfortunately, that mode has some drawbacks because we have not put in all of the whizbang functionality that you get in the C# mode in Visual Studio.

Scott Hanselman: Okay.

Rustan Leino: That means that you get our additional design time support in that mode, but for



programmers who either would like to keep all of the refactoring and all of the code formatting and features that are really nice to use in C#, they can download Spec#, then in the properties pane of the project, we add a new contracts tab and one can turn on contract. What happens then is when you then compile your C# program, you first run the C# compiler and then immediately afterwards we run the Spec# compiler. The Spec# compiler then would produce the same result as the C# compiler, but the Spec# compiler is also going to peek into specially marked comments. So, what you can do with a precondition is if you put it into a comment that begins with a carat sign, then that's just a comment to the C# compiler, but the Spec# compiler will read it and understand it and generate code based on that.

Scott Hanselman: Now, I'm realizing that some listeners who are a little more familiar with this kind of concepts may think that this is a lot like some of the aspect-oriented compilers that have come out for C# like XC# where you could decorate C# with an attribute and say that I wish to insert tracing pre-imposed conditions. How is this different from those kinds of aspect-oriented things?

Mike Barnett: In many ways, it's quite similar. Pre-imposed conditions of course can be thought of as aspects because they slice into a particular well-defined joined point. The difference with Spec# is that the checking we do, the static checking we do is something that goes far beyond what an aspect-oriented tool would be capable of. An aspect-oriented tool is capable of injecting code into the runtime for dynamic behavior, but our static verification relies upon a particular discipline of how you write your programs and use your fields in your objects to guarantee the correctness and robustness against future code evolution so the surface syntax in that sense would be the same. It's like putting an attribute on a method. It's something that the C# compiler ignores, but the value then is in the tools that we can apply to that code afterwards.

Scott Hanselman: I see, okay. So, I could see myself using an aspect-oriented type of a system in conjunction with Spec#, but other than that, initial superficial similarity in that there are pre-imposed conditions and these are as you said the joined points, that's about where the similarity ends and it's really about static verification of the correctness of your application. So, presumably then I should be able to access that information and then generate documentation from it.

Rustan Leino: Yes. The information is there and one could extract it into a document that reads in English or with the formulas to describe what the documentation ought to say.

Scott Hanselman: So, that's almost like Spec# enforces my comments.

Rustan Leino: Yes. It's a way to formalize your comments, that is, put them into something that is going to be read by the machine and if you want to then also checked by the machine to make sure that they are up to date with the current program text.

Scott Hanselman: Very cool. I think that anything that allows the programmer to more explicitly and formally express their intent such that it is unambiguous and a machine readable format is definitely a really good thing.

Rustan Leino: That's right. Expressing an intent is what contracts are about.

Scott Hanselman: Cool. Well, thank you very much Rustan and Mike from the Microsoft Research Team and, again, the product is Spec# and you can download that. We'll have links up on the show site.

Rustan Leino: Thank you very much.

Mike Barnett: Thank you.

Scott Hanselman: And this has been another episode of Hanselminutes and we'll see you again next week.