

Status Report for Week 2

Phillip Casey & Greg Glazer
12 April 2009

Accomplishments for the Week

- Created Presentation Outline
- Created Major Milestones
- Continued Research to start and understand how Spec# works
- Began to create some initial sample programs and *play* with Spec#

Goals for Next Week

- Create an outline of the Spec# components we want to research and understand
- Non-Null Types Sample Programs
- Exception Paradigm Sample Programs

Additions to Annotated Bibliography

Barnett, Mike, and Leino, Rustan, and Schulte, Wolfram. The Spec# Programming System: An Overview. Available from <http://research.microsoft.com/en-us/projects/specsharp/krml136.pdf>; accessed 7 April 2009.

Describes the goals and the architecture of the Spec# Programming System. One of the first formal and fully published introduction to the entire Spec# Programming System; while the material is now a little dated, this paper provided some key insights into the intention behind Spec# and into the architecture of Spec#. Specifies the overall Spec# methodology: Spec# enriches C#, uses a combination of static-analysis techniques and runtime checks to guarantee soundness.

Barnett, Mike. Spec#: Adding Contracts to C#. Available from <http://msevents.microsoft.com/CUI/WebCastEventDetails.aspx?culture=en-US&EventID=1032273351&CountryCode=US>; accessed 9 April 2009.

A one hour web meeting to provide an overview of Spec#; in particular in relationship to one of the core intentions of Spec#: Explicit Program Specification. Answers the following main questions: What is Spec#? What are the main components of Spec#? How does the Spec# program verifier use theorem-proving to check the consistency of the program and the contract?

Notes from the Week

Milestones

We identified the following major milestones for this project:

- Spec#, how it works?
 - C# Code with Spec# add-on

- Boogie Program Verifier (static verification)
- Runtime Checks
- Program demonstrations and samples:
 - Non-Null Types
 - Method Specification
 - Design by Contract
 - Pre and Post Conditions
 - Exception Paradigm
 - Class Contracts (Invariants)
- Application to the bigger picture
 - Responsibility
 - Caller
 - Provider
 - Focus on Practical Aspects
 - Helping aid the Software Engineering goal

New Learning's

Intentions of Spec#

A manner to capture the programmer's intentions; programmer's assumptions are left unspecified or, if specified, in a static, natural language documentation. Spec#'s intention is to allow those specifications and assumptions to be embedded with the program and then, further, taking those embedded specifications and turning them into run-time checks; in this manner allowing dynamic correctness checking.

Intended to be a superset of C#, providing the following features:

- Non-Null Types
- Method Specifications (Pre and Post Conditions)
- Exception Management Discipline
- Constraints over an object's data elements.

Non-Null Types

“we would like to eradicate all null-dereference errors”. The ability to determine what expressions may evaluate to null and those that are sure not to. When applied to objects, the internal data elements must be initialized before calling the base class—this is because the base class can result in a dynamic method dispatch (ie. A virtual method called from a constructor).

Method Contracts

A specification which outlines a contract between the caller and the implementation.

Pre-Condition: the state in which the method is allowed to be called. The caller's **responsibility**. Enforced through run time checks that throw `RequiresViolationException` if not satisfied.

Post-Condition: the state in which the method can return. The implementer's **responsibility**. Enforced through Boogie and, if Boogie is successful, then run-time checks are injected and throw an `EnsuresViolationException`

Exception Handling

Goodenough, John. Structured Exception Handling. ACM, January 1975. Classifies exceptions into Client Failures and Provider Failures. Client Failures occur when a method is invoked under an illegal condition. Provider Failure is a implementation failure.

Spec# extends this classification. Two types of Provider Failures:

- Admissible Failures: occurs when a method is not able to complete its intended operation (socket connection timeout, network error, etc..). Part of the contract between the caller and the implementer. Expectation is that the program would anticipate and handle these types of errors. Checked Exceptions (`ICheckedException`)
- Observed Program Errors: an intrinsic error in the program (`ArrayOutOfBounds`) or a global failure not tied to the method (`OutOfMemory`). `Unchecked Exceptions`.

Frame Conditions

Expressed in *modifies* statements. Abstract over program state. Must state what is modified, but information hiding states that this should not occur.

Class Contracts

Method Contracts: specifies what is expected of the caller and what the caller can expect in return.

Object Invariants: specify what is expected to hold of an object's data fields when the object is in a steady state.

Interestingly enough, we cannot expect an object invariant to *always* hold; changes sometimes require multiple statements, so the state may be violated for many statements of execution.

Steady State: When an object is not being operated upon.

Exposed State: The object is vulnerable to modifications.

Expose Block:

```
    expose( o )  
    {  
        S;
```

}

Run-Time checks are injected to ensure that an object's invariants hold.

Uses a state specific ownership relation that structures objects into tree-like hierarchies that can be transferred.

Modular checking of invariants (class does not need to know it's parent's or children's invariants) is handled through class frames which define the invariant. Class Frames are handled through the expose block.

Spec# Architecture

Compiler

Fully Integrated into Visual Studio which permits:

- Build Process
- Design Tools
- Syntax Highlighting
- IntelliSense

Produces two main 'outputs':

- Executable byte code
- Preserves the specifications in a language independent mechanism
 - Powerful: not only language and compiler independent, but allows independent reasoning over the specification.

Boogie

Consumes compiled code. Allows Boogie to verify programs written in other languages. Integrated via an Out of Band Process

Boogie's Architecture

Constructs a program into it's own intermediate language, BoogiePL.

BoogiePL is processed by an inference system (designed to automatically extend a knowledge base; maps sets into sets, handles the way rules are combined). This inference system produces derived properties and then adds appropriate statements (ie. assert, assume).

BoogiePL Program then goes through many transformations.

One example transformation: cutting all loops to derive an acyclic control flow graph
Feedback from the theorem prover is mapped back into the source program.

Practicality

Spec# really seems to focus on practically useful software with a focus on providing incremental benefits as programmers adopt the features while programmers start to get an immediate benefit.

Notes from Web Meeting

Contracts

Designed to capture a programmer's intentions about how methods and data should be used. Some examples include pre-conditions and post-conditions.

How does spec# emit run-time checks that enforce these contracts?

How does the spec# program verifier use theorem-proving to check the consistency of the program and the contract?

Compile Time checking (Static verification)

Runtime theorem prover (Boogie)

Features of Spec#

- enhanced version of type checking
- runtime checks to enforce the contracts
- static program verification system: at compile time, tells you about the problems between your written code and your contracts

What does the presentation define as the software engineering problem? To build and maintain large systems that are correct

- build
- maintain
- correct

Trying to get some "feeling" that it fulfills a specific scenario, or it contains a specific layering for architecture

- unit testing
- code reviewing

Specifications record design decisions: bridge the intention and the code. How about putting the specification in the code?

Tools that amplify human effort

- Assist programmers by providing tools to assist with what is already being done

What is Spec#?

A conservative Superset of C#. Any valid C# program is a valid Spec# program.

First example:

Constructor with null argument. Checking of arguments if they are null.

Add annotation on type (!) - cannot be null.

Correlates to requires object != null.

Preconditions:

- set of requires statements before method body

Mscorlib.Contracts

- Looked at provided base classes and defined contracts on these base classes. (ie. array.CopyTo)
- Attach contracts to base classes.

Violations of contracts detected in code come from a Theorem Checker running in the background that takes current code and tries to find counterexamples in the code which correlate to violations of the contract.

Not Type Safe Example

```
abstract class B
```

```
{  
public B {this.M();}  
public abstract int M();  
}
```

```
class C : B
```

```
{  
T! x;  
public C(T! y) : base  
{  
this.x = y;  
}
```

// x.f null dereference, because base() call happens before constructor executes

```
public override int M() { return x.f; }  
}
```

Solution: Spec# compiler requires all non-null fields be initialized before leaving flow of control.

Exception Based Paradigm:

- Caller Failures
- Callee Failures
 - Implementation bug (IndexOutOfBounds)
 - Anticipated Conditions (Socket Closed)
 - UnAnticipated Conditions (OutOfMemory)

What to do with exceptions?

- Caller Handles
- Caught by system or never handled

Caller should only handle Anticipated Conditions. - Checked Exceptions

All other conditions. - Unchecked Exceptions

ICheckedException

Methods must declare which checked exceptions they may throw. Avoids long (useless) list of all possible exceptions. Can also add post conditions that specify what conditions are guaranteed to be true if that exception occurs.

Invariants

Simple example: allocate a new array of size $y-x$, could end up with a negative number. Could use a precondition on method that $x < y$. But that's not what Preconditions are for; they are for contracts between class and client. Clients don't know internal representation.

Object Invariant: Private View of the Internal State of an object. Defines the relationship between fields inside of an object that means it's in a valid state.

Control the scope of where an invariant might be violated. Example of incrementing two integers where invariant $x \leq y$. Then you need `expose(this) {x++; y++}`

Consistent Vs Mutable

Every public method gets a default precondition: `requires this.IsConsistent`. Used for re-entrancy; can enter `IsMutable` methods inside an exposed block

Composite Objects.

You know if your self is consistent, but how do you know that objects you contain are consistent.

Leak method - exposes private member.

Spec# introduces Ownership. When your class has a field that is a reference type, the default is you own it (`[Owned]`)

When you expose self, ownership is relinquished and then, at the end of expose block, it is re-obtained

Three Possible States:

- Mutable (inside expose block)
- Consistent (invariant holds)
- Committed (locked, nobody else can modify it)

With composite types, reference type field transitions to `IsMutable`, then when it's done, back to `IsConsistent`, then back to owning object which does the same.

Ownership partitions the heap. Gives you tree structure relationships among objects. Locking Protocol guarantees that you always enter an exit at root. Means you can hand

out a reference to your internal type--can distinguish between reference and owner of object. Inviolable

Concurrency in C#: no guarantees.

Concurrency in Spec#.

Objects are born unshared, must be explicitly shared. Even when shared, still has ownership with Locking Protocol.

VERY POWERFUL: Sequential Reasoning still works.

How does a Program verifier work?

Programs with specifications goes to a Verification Condition Generator.

Verifications go into a Theorem Prover.

Spits out "Correct" or a list of errors.

You need a LOT of specification to make that work. What Spec# does is (diagram)

Inference Engine which infers as many specifications as it can. Tries to reduce the annotation burden.

Eifel does not have a sound system for Object Invariants.

Outline

- Non-Null Types

T x; can be null, or a reference to an object of type T

T! x; only can be a reference to an object of type T. CANNOT be null

- Pre and Post Conditions

- ICheckedException

- Static Program Verification (pre-condition violations)

- Parameter Validation (ArgumentNullException)