Spec#

Andreas Vida

Motivation

- Correct and maintainable software
- Cost effective software production
- Implicit assumptions easily broken
  → Need more formal specification
- Integration into a *popular* language

What has been done before

- Pioneering languages
  - Gipsy
  - Euclid
- More recent developments
  - Eiffel
  - SPARK
  - JML

Now: Spec#

- Extension of C# (Microsoft Research) that provides:
  - A sound programming methodology
  - Tools to enforce this methodology
  - Smooth adoption path for new-comers

Components of Spec#

- Boogie static-verifier
- Spec# compiler
  - Emits run-time checks
- Integration into Visual Studio
  - IntelliSense code completion
  - Syntax Highlighting

Spec# language features

- non-null types
- checked exceptions
- class contracts (object invariants)
- method contracts
  - pre- and (exceptional) postconditions
- frame conditions
- inheritance of specification

Non-null types

- Notation: T!
- Constructors need initialiser-fields for each non-null field
  → evaluated before base-class-constructor call!

```
class Student: Person {
    Transcript! t;
    public Student(string name, EnrollmentInfo! e):
    t(new Transcript(e)), base(name)
    { /*...*/}
    //...
}
```

Checked vs. unchecked exceptions

- C# only has unchecked exceptions
- Spec# in this way similar to Java
- Considers 2 types of exceptions:
  - Admissible failures
    → interface: ICheckedException
  - Client failures, observed program errors
    → derived from: Exception

Method contracts

- Preconditions example:

```
class ArrayList {
   public virtual void Insert(int index, object
   value)
   requires 0 <= index && index <= Count;
   requires !IsReadOnly && !IsFixedSize;
   { /* ... */ }
   //...
}
```

Preconditions

- Enforced by run-time checks that throw a RequiresViolationException
- An alternative exception type can be specified using an otherwise clause:

```
class A {
    public void Foo(int a)
    requires a > 0
    otherwise ArgumentOutOfRangeException;
    { /* ... */ }
}
```

Postconditions

- ArrayList.Insert's postconditions:

```
ensures Count == old(Count) + 1;
ensures value == this[index];
ensures Forall{ int i in 0: index; old(this[i]) == this[i]};
ensures Forall{ int i in index: old(Count); old(this[i]) == this[i+1]}
```

- Complex quantified expressions supported
- Boogie attempts to verify postconditions
- Eiffel's mechanism: old() are saved away at the method's entrance

Exceptional postconditions

- Methods have a throws-set (as in Java)
- throws clause (only for checked exceptions) can be combined with postconditions:

```
void ReadToken(ArrayList a)
  throws EndOfFileException
  ensures a.Count == old(a.Count);
{ /*... */ }
```

- “Foolproof”: if static checks can’t ensure that the exception is checked then run-time checks are emitted

Class contracts

- Object invariants:

```
class AttendanceRecord {
    Student[]! students;
    bool[]! absent;
    invariant students.Length ==
absent.Length;
    /*...*/
}
```

- Often need to be temporarily broken
  → do this explicitely:
  `expose (variable) { ... };`

Frame conditions

- Restrict which part of the program state can be modified by a method

```
class C {
    int x, y;
    void M() modifies x;
    { x++; }
}
```

- How to change private parts of an outside class? → wildcards:
  `modifies this ^ ArrayList;`
- Still a problem: aggregate objects

Run-time checking

- Pre- and postconditions are turned into (tagged) inlined code
- Conditions violated at run-time → appropriate contract exception
- 1 method is added to each class using invariants
- Object fields added:
  - invariant level
  - owner of an object

Boogie: Static verification

- Intermediate language → BoogiePL
- Inference system
  - Obtains properties (loop invariants) then adds assert/assume statements
- Creates acyclic control flow graph by introducing havoc statements
- Calls the “Simplify” theorem prover
- Maps results back onto source code

Future plans

- Out-of-band specification
  - Add specification for the .NET base class library → semi-automatically
- Provide Transformations:
  - Contracts to natural language
  - Spec# to C# compiler

Time for questions