Dafny

Dafny is

- An OO programming language similar to Java and C#
  - ∗ With features to express code contracts
- A tool chain for compilation and verification
  - ∗ Verified methods do not crash.
    - · No null pointer dereferences
    - · No array index out of bounds
    - · No divide by zero
    - · No infinite loops or recursions
  - ∗ Verified methods implement their contracts

Dafny is one of several systems that can be used to verify code.

- VCC the verifier for concurrent C.
- Microsoft Code Contracts is a system for C# verification.
- OpenJML is a system for Java
- System Verilog Assertions – Verilog with property checking

The Toolset

Compiler

Dafny Program

$\Downarrow$ *Dafny Compiler*

C# Program

$\Downarrow$ *C# Compiler*

CIL code (.net)

Verifier

$$\boxed{\text{Dafny Program}}$$
$\Downarrow$ *Translator*
$$\boxed{\text{Boogie IVL program}}$$
$\Downarrow$ *Boogie*
$$\boxed{\text{1st order formulae}}$$
$\Downarrow$ *Z3 (SMT prover)*
$$\boxed{\text{Yes/No/Time out}}$$

Boogie IVL is a ‘programming’ language intended only for verification.

- Lacks heap, classes, modules, and other ‘high level’ concepts
- Unconstrained by the need to be executed.
- Boogie IVL and Boogie are shared by many projects.
- Boogie verifies Boogie IVL by generating “verification conditions”
  - ∗ I.e. 1st order formulae that need to be shown universally true.
  - ∗ Boogie uses Z3 to check these verification conditions

Verifier

Dafny Program

$\Downarrow$ *Translator*

Boogie IVL program

$\Downarrow$ *Boogie*

1st order formulae

$\Downarrow$ *Z3 (SMT prover)*

Yes/No/Time out

Z3 is a satisfaction modulo theories (SMT) automated theorem prover.

- SMT provers can show a wide range of formulas to be universally true.
- SMT prover can often generate counter examples when the VCs are not true.
- Counter examples can provide useful insight to the programmer.

Methods and contracts

Methods compute one or more values and may change state.

Methods declared outside classes are allowed (similar to C++)

Example

```
method between( p : int, r : int ) returns (q : int )
    requires r-p > 1
    ensures p < q < r
{
    q := (p+r) / 2 ;
}
```

Note:

- Input parameters, like p and r, are immutable.
- Output parameters, like q, are named.
- The requires clauses of the contract declare preconditions.
- The ensures clauses of the contract declare postconditions.
- The contract for between guarantees no state is changed by calling between.
- The **int** type is infinite (equivalent to $\mathbb{Z}$).
- Comparison operators are 'chaining': $p < q < r$ means $p < q$ && $q < r$

The verifier will attempt to prove that the method body implements the contract.

It does this by calculating the weakest precondition P such that

$$\{P\}\ q := (p + r)/2;\ \{p\ <\ q\ <\ r\}$$

is correct and then proving that the given precondition is as strong:

$$\forall p, r \in \mathbb{Z} \cdot r - p > 1\ \Rightarrow\ P$$

In this case the weakest precondition P is $p < (p+r)/2 < r$. So the prover needs to prove

$$\forall p, r \in \mathbb{Z} \cdot r - p > 1\ \Rightarrow\ p < (p + r)/2 < r$$

In this example, we could replace the method body with

q := p + 1 ;   or   q := r-1 ;

Calls

Example

```
method between( p : int, r : int ) returns (q : int )
    requires r-p > 1
    ensures p < q < r
{
    q := (p+r) / 2 ;
}
```

When a method is called, it is checked that the precondition is respected. E.g.

var a : int ; // a is initialized to an arbitrary int
var c : int := <u>between(a, a+1)</u>; ⟵ *Fails*

An error is reported because the attempt to prove

$$\forall a \in \mathbb{Z} \cdot a - (a+1) > 1$$

fails. In fact the prover can prove that

$$\neg\left(\forall a \in \mathbb{Z} \cdot a - (a+1) > 1\right)$$

After the call, the postcondition, with parameters replaced by arguments, is assumed to be true.

var a : int ; // a is initialized to an arbitrary int
var c : int := between(a, a+4) ; ⟵ *Succeeds*
assert a <= c-1 && c <= a+3 ; ⟵ *Succeeds*

In this case $(p < q < r)[p, r, q : a, a{+}4, c]$ is $a < \ c < \ a{+}4$.

The assert command verifies because the prover can prove

$$\forall a, c \in \mathbb{Z} \cdot a < \ c < \ a+4 \ \Rightarrow \ a \leq c-1 \wedge c \leq a+3$$

But

var a : int ; // a is initialized to an arbitrary int
var c : int := between(a, a+4) ; ⟵ *Succeeds*
assert c = a + 2 ; ⟵ *Fails*

Does not verify because the prover can not prove

$$\forall a, C \in \mathbb{Z} \cdot a < \; c < \; a + 4 \; \Rightarrow \; c = a + 2$$

I.e., methods are *abstraction boundaries*

The only information the caller can use about a method is in its contract.

Asserts

Use assert commands to document your code.

Asserts (like postconditions) are verified documentation.

Asserts (like pre- and postconditions) are ignored by the compiler.

So we can put in formulas that would be inefficient or impractical to run.

Asserts for debugging

Asserts can also be useful in debugging.

Suppose we have a big long method:

method divideWithRemainder(x : int, y : int)
returns (p : int, m : int)
    requires y > 0 &&x >= 0
    ensures p*y + m == x && 0 <= m < y
{
    var q : **int** ;
    Some code intended to make $q * y$ *bigger than* x
    Some more code
}

But the postcondition doesn't verify.

Bisect the code with an assert:

method divideWithRemainder(x : int, y : int)
returns (p : int, m : int)
    requires y > 0 &&x >= 0
    ensures p*y + m == x && 0 <= m < y
{
    var q : **int** ;
    Some code intended to make $q * y$ *bigger than* x
    assert q*y > x ;
    Some more code
}

If the assert verifies, all bugs are in the second half.

Dafny vs. proof outline logic

In POL, although correct, the following is not provably correct —i. e., it is not verifiable using only the rules presented earlier—

$\{a = 1 \wedge b = 1\}$
$b := a + b$
$\{\mathrm{Even}(b)\}$
$a := a + b$
$\{a = 3\}$

because $\{\mathrm{Even}(b)\}\, a := a + b\, \{a = 3\}$ is not correct.

Dafny verifies

var a := 1 ; **var** b := 1 ;
b:=a+b ;
assert Even(b) ; ⟵ *Succeeds*
a:=a+b ;
assert a == 3 ; ⟵ *Succeeds*

because it tries to verify

$\{a = 1 \wedge b = 1\}$
$b := a + b$
$\{\mathrm{Even}(b) \wedge P\}$
$a := a + b$
$\{a = 3\}$

where P is the weakest condition such that $\{P\}\, a := a + b\, \{a = 3\}$ is correct.

In other words: In Dafny, adding an assert can't hurt.[1]

[1] In practice extra asserts may increase verification time and lead to a timeout.

Loops

While loops should include an invariant and a variant.

Consider

```
method root( x : int ) returns ( p : int )
        requires x >= 0
        ensures p*p <= x < (p+1)*(p+1)
{
        p := 0 ;
        var r := x + 1 ;
        while p+1 != r
                invariant p+1 <= r
                invariant p*p <= x < r*r
                decreases r-p
        {
                var q := between(p, r ) ;
                if q*q <= x { p := q ; } else { r := q ; } } }
```

Inferring variants

In fact the verifier can often guess the variant on

For the example above, we can get away with

```
while p+1 != r
      invariant p+1 <= r
      invariant p*p <= x < r*r
{...}
```

The verifier fills in the variant with its best guess.

Inferring invariants

The verifier will also infer some invariants and effectively rewrite your code to add them in before generating verification conditions.

For example, this loop verifies:

```
while p+1 != r
    invariant p*p <= x < r*r
{
    var q := between(p, r ) ;
    if q*q <= x { p := q ; } else { r := q ; } } }
```

even though, the precondition of between does not follow from

$$p + 1 \neq r \wedge p \times p \leq x < r \times r$$

(Consider $p = 0$, $r = -2$.) So the verifier must have inferred (or guessed) some additional invariant, such as $p + 1 \leq r$.

Omitting necessary invariants makes your loops harder to read.

Advice:

- State all invariants needed to prove the loop, whether or not the verifier can infer them.
- Do not state invariants that are irrelevant to the correctness of the loop.

Assertions as Hints

Sometimes an assertion can guide the prover to consider facts it otherwise wouldn't.

(WARNING: This example is now out of date as Dafny can now verify the orginal version.)

Consider the Russian Peasant Multiplication algorithm

```
method mult( a0 : int, b0 : nat ) returns (c : int )
        ensures c == a0 * b0
{
        c := 0 ;
        var a := a0 ;
        var b : nat := b0 ;
        while b != 0
                invariant a0*b0 == c + a*b ⟵ Times out
                decreases b ;
        {
                if b%2==1 {
                        c := c + a ;
                        b := b - 1 ; }
                b := b/2 ;
                a := 2*a ; }
}
```

The prover used to time out trying to show that the indicated invariant is maintained by the loop.

Where is the problem?

We bisect the loop body with an assertion.

```
if b%2==1 {
    c := c + a ;
    b := b - 1 ; }
assert a0*b0 == c + a*b ; ⟵ Succeeds
b := b/2 ;
a := 2*a ;
```

The assertion verifies, but not the invariant. So the problem must be that the verifier can not deduce that

```
b := b/2 ;
a := 2*a ;
```

maintains the invariant.

But, this is only the case if b is even prior to b := b/2 ;

Perhaps the verifier has failed to use the fact that b is even prior to b := b/2 ;

We use an assertion to force the verifier to prove that b is even after the if.

```
if b%2==1 {
    c := c + a ;
    b := b - 1 ; }
assert b % 2 == 0 ;
b := b/2 ;
a := 2*a ;
```

Having proved that b is even, the verifier will try to make use of that information.

The loop now verifies.

Soundness and spurious errors

Dafny is intended to be **sound**:

If the verifier reports no errors for a method:

- The method is correct.
- I.e., it meets its specification and terminates.

Dafny is not **complete**:

If the verifier reports at least one "verification failure"

- Either the method is not correct,
- **or** the method is correct, but the verifier can not prove it (spurious failure).

Sources of spurious failures

- Loop invariants are too weak.
- The verifier needs more guidance.

Example traces

- When verification fails, the verifier produces an example trace.
- The verifier can not prove the trace is not a counter-example.
- Even spurious failures produce traces.
- In the VS environment, the Boogie Verification Debugger displays the example.
- Often it is useful to consider these examples to find the reason for failure.