Formal Methods

There are two key issues when building software systems:

1. Validation – are we building the right product?
2. Verification – are we building the product right?

Validation is done during requirements analysis with the feedback of the customer. Verification is done during testing of the product. Rigorous testing needs working code in order to run test cases. But we have also seen that even with testing we will never be able to prove the absence of bugs.

Therefore, for sensitive systems, particularly where human lives are at stake we need something else => Formal Methods.

A formal methods approach to software design implies writing a product specification in some formal notation such as first order logic. This in turn implies that the formal specification has a mathematical semantics or interpretation which allows us to rigorously inspect the specification for clarity (unique interpretation of each statement) and consistency (no contradictory statements). In addition, due to the mathematical nature of formal specification we can actually prove system properties to hold. For example we can formally show that a system will behave in a certain way given a particular class of inputs. This goes well beyond what test cases can do.

If the formal notation we chose is executable, then the formal specification of our system can be viewed as a prototype system. And we can actually run the formal specification to see if the system behaves the way the system design intended. This adds another aspect to formal methods, we can also use them for validation.

Thus, we can use formal specification methods for validation by using them to show customers the behavior of the specified system. Also, we can use formal methods for verification because we can prove mathematical properties of the system, in particular, we can prove that it behaves as desired over a given set of inputs.

Unfortunately, applying formal methods is time consuming and expensive. However, given the different aspects of formal methods we can apply them in varying degrees of formality and still derive a benefit.

1. Validation – write the specifications in a formal specification language and then only informally validate them – no proofs.
2. Validation with some Verification – validate the formal specification and only prove critical components of the system correct.
3. Validation with full Verification – validate formal specifications and then prove all aspects of the system correct – VERY EXPENSIVE – this only reserved for the most critical of systems.

Equational Logic

The formalism we are using is called equational logic. In equational logic axioms are written as equations and we use equational deduction to reason about these axioms.

Example: A simple equational specification of the computation of a power of two.

```
begin
  include Integer
  oper square : Integer -> Integer
  var  I:Integer
  equ  square(I) = I * I
end
```

In this specification we include the integers so that they are available to us to use for our own specification. We then proceed to define our 'square' operation which takes an integer as an argument and produces an integer as a result. After declaring a variable 'I' of type integer we define some behavior for the operations which given a value simply multiplies this value with itself and returns that as the result.

We could envision executing this specification by saying something like this:

```
> execute square(5)

Result: 25
```

This is precisely what the equational specification language BOBJ does. It allows you to write equational specifications and then gives you the environment to actually execute these specifications.

Getting Started with BOBJ

BOBJ is a specification language based on equational logic and algebra. Informally, in equational logic axioms are expressed as equations with algebras as models and term rewriting as operational semantics. Due to this efficient operational semantics, BOBJ specifications are executable; once you have specified your system, you can actually run the specification and see if it behaves correctly. Furthermore, due to the fact that BOBJ is rigorously based on mathematical foundations you can use BOBJ to prove things about your specifications.

Lets revisit the specification from before, but now write it as a BOBJ specification.

```
obj SQUARE is
  protecting INT .
  op square : Int -> Int .
  var I : Int .
  eq square(I) = I * I .
endo
```

There are some minor syntactical differences to our intuitive specification from before. If you assume that this is saved in the file 'square.bob', then we can load this file into BOBJ and run the specification:

```
$ java -jar bobj.jar
> in square.bob
> reduce square(5) .
       Result: Int: 5
>
```

Another simple example, specify the computation of the smaller of two values.

```
obj MIN is
  protecting INT .
  op min : Int Int -> Int .
  var X : Int .
  var Y : Int .
  ceq min(X, Y) = X if X ≤ Y .
  ceq min(X, Y) = Y if X > Y .
endo
```

To execute this specification:

```
$ java -jar bobj.jar
> in min.bob
> reduce min(3,5) .
       Result: Int: 3
> reduce min(5,3) .
```

```
        Result: Int: 3
>
```

For more information, see the bobj-quickstart document.