BOBJ: A Tutorial

Lutz Hamel
Dept. of Computer Science and Statistics
University of Rhode Island
Kingston, RI 02881
hamel@cs.uri.edu

– DRAFT 9/22/06 –

Getting Started

BOBJ is a specification language based on equational logic and algebra.[1] Informally, in equational logic axioms are expressed as equations with algebras as models and term rewriting as operational semantics. Due to the 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.

Specifications

The basic specification unit in BOBJ is called an object. objects themselves consist of type definitions (called sorts in BOBJ lingo) and operator name definitions. Behavior is defined for the operators via a set of equations. The equations themselves can also contain variables. The following could be considered a typical specification:

```
obj NUMBERS is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  var M : Nat .
  var N : Nat .
  eq M + s(N) = s(M + N) .
  eq M + 0 = M .
endo
```

[1]Actually BOBJ is based on hidden (or behavioral), order-sorted equational logic, here we only discuss the many-sorted sublanguage of BOBJ.

This specification defines the object `NUMBERS`. Within this object we define the sort `Nat` as well as the three operator symbols: `0`, `s`, and `+`. The notation for the operator symbol declaration follows very closely standard mathematical practice: `op 0 :  -> Nat` defines a constant of type `Nat`, `op s :  Nat -> Nat` defines a unary function symbol that takes a value in domain `Nat` to its codomain `Nat`, and `op _+_ :  Nat Nat -> Nat` defines an infix binary operator. BOBJ admits mixfix operator declarations, that is, we can define prefix, infix, or postfix operator symbols. When no special action is taken during the definition of a function symbol it defaults to being a prefix function symbol. You can use underscores to tell the BOBJ parser how the operator is to appear in the target expressions. Following the operator declarations we have the variable declarations. Finally, we assign some behavior to the `+` operator with the equations appearing at the end of the object. The first equation states that computing the successor of `N` and then adding that to `M` is the same as first adding `M` and `N` and then computing the successor of the sum. The second equation states that adding zero to a value will not change the value.

Running BOBJ

Before you can load a specification into BOBJ your have to start BOBJ, usually by typing the keyword `bobj` at the Unix command prompt. The system will respond:

```
[lutz]% java -jar bobj.jar
                    \|||||||||||||||||/
                  --- Welcome to BOBJ ---
                    /|||||||||||||||||\
         BOBJ version 2.0  built: 2000 Jun 20 Tue 20:18:45
            Copyright 1988,1989,1991 SRI International
                   2002 Nov 21 Thu 7:33:22
BOBJ>
```

At this point you are ready to load specification files into BOBJ. Assume that the above specification is called “numbers.bob”, then in order to load this into BOBJ you type `in numbers`:

```
BOBJ> in numbers
==========================================
obj NUMBERS
BOBJ>
```

Once the file is loaded we might want to demonstrate that our specification behaves appropriately. BOBJ provides us with another keyword to accomplish that : `reduce`. This keyword activates BOBJ’s theorem proving engine. Consider that we want to show that the first equations holds during an actual computation. We can type in the following command:

```
BOBJ> reduce s(0) + s(s(0)) == s( s(0) + s(0) ) .
```

```
reduce in NUMBERS : s(0) + s(s(0)) == s(s(0) + s(0))
rewrites: 4
result Bool: true
BOBJ>
```

Note that we get back the result `true`. BOBJ has proven that the property indeed holds, i.e., that the behavior defined by the equation is what we expected. With the `==` symbol you are asking BOBJ to prove that the left term is equal to the right term.

We can also use BOBJ to simulate actual computations within the specification. Consider:

```
BOBJ> reduce s(0) + s(s(0)) .
reduce in NUMBERS : s(0) + s(s(0))
rewrites: 3
result Nat: s(s(s(0)))
BOBJ>
```

In this case we want to know what happens when we add the successor of zero to the successor of the successor of zero. As the results show, we get back what we expected, raising our confidence that our specification is indeed correct.

Built-ins

In the previous section we used the zero constant and successor function to simulate integers. BOBJ allows us to use regular integer notation via a built-in module. BOBJ includes a module INT and we can use it in our own specifications. Assume that we want to write a specification for a function that computes the Fibonacci numbers:

```
obj FIB is
  protecting INT .
  op fib : Int -> Int .
  var M : Int .
  eq fib(0) = 0 .
  eq fib(1) = 1 .
  ceq fib(M) = fib(M - 1) + fib(M - 2) if M =/= 0 and M =/= 1 .
endo
```

This specification introduces a number of new features in BOBJ. First, we have the `protecting` keyword which allows us to import other modules, in this case the built-in module `INT`. Within the equation section we have what is referred to as a *conditional equation* which starts of with the keyword `ceq` and includes an *if* clause to the right of its definition. Here we make sure that this equation is only applied when the variable `M` is neither 0 or 1. Again, we can use the BOBJ theorem prover to convince ourselves that the specification behaves appropriately:

```
BOBJ> reduce fib(10) == 55 .
reduce in FIB : fib(10) == 55
rewrites: 1058
result Bool: true
BOBJ>
```

Exercise 1 *Write a specification for a function that computes the factorial of a given integer. Use the* `reduce` *command to demonstrate that your function specification works correctly for* 0!, 1!, *and* 3!.

Exercise 2 *Write a specification for a list of integers representation (hint: cons function from Lisp; null to represent empty lists). Once you have the representation, write a function, say ladd, that returns the sum of the elements in an integer list. Use the* `reduce` *command to demonstrate that your function specification works correctly.*

Summary

In general, the structure of objects in BOBJ is as follows:

```
obj <object name> is
  <include directives>
  <type definitions>
  <operator name definitions>
  <variable declarations>
  <equations>
endo
```

It worthwhile pointing out that each statement within an object needs to be followed by a period. Also, the period needs to be separated from the text by a space.

As we pointed out above, equations can have a simple format

```
eq lhs  = rhs  .
```

or they can have the conditional format:

```
ceq lhs  = rhs  if cond  .
```

At the command line level, BOBJ provides two useful commands:

1. `in` – allows you to load object specifications into BOBJ.
2. `reduce` – activates the BOBJ theorem prover.

A Stack Specification

So far we looked at very simple specifications, but we are interested in specifying software systems. Let's take a look at a slightly more complicated specification. The following is a specification of a stack.

```
obj STACK is
  sort Stack .
  sort Element .
  op top : Stack -> Element .
  op push : Element Stack -> Stack .
  op pop : Stack -> Stack .
  var S : Stack .
  var E : Element .
  eq top(push(E,S)) = E .
  eq pop(push(E,S)) = S .
endo
```

The specification follows the pattern defined in the previous section. We define two sorts, one for the stack and one for the elements that can be pushed on the stack. Then we define our operation symbols: `top` is an operation that allows us to inspect the top of a stack, `push` allows us to push an element on a stack, and finally `pop` allows us to pop the top of the stack. We also declare two variables `S` and `E`. Finally, we give this specification our intended behavior with the two equations. The first equation states that looking at the top of the stack after something has been pushed will return the element that has been pushed. The second equation states that popping a pushed element returns the original stack.

Now we are ready to examine some properties of stacks. We introduce a new feature of BOBJ to do that: *proof scores*. You can think of a proof score as a program for BOBJ's theorem prover. The following is our proof score that demonstrates some basic properties of stacks.

```
*** Stack Proof Score ***

open STACK .

op s : -> Stack .
op e1 : -> Element .
op e2 : -> Element .

***> demonstrate some basic properties of stacks
reduce pop(pop(push(e1,push(e2,s)))) == s .
reduce top(pop(push(e1,push(e2,s)))) == e2 .

close
```

Proof scores usually begin with opening the object that we want to prove things about. This makes the operations and equations of the object available to the theorem prover. In our case our object of interest is STACK. Once we have access to our object we declare some constants as place holders for general values. Here, `s` stands for all stacks and `e1` and `e2` are place holders for element values. Once this is in place we can go ahead and demonstrate some system properties by first loading the stack object and then its proof score.

```
[lutz]$ java -jar bobj.jar
                     \|||||||||||||||||/
                   --- Welcome to BOBJ ---
                     /|||||||||||||||||\
            BOBJ version 2.0  built: 2000 Jun 20 Tue 20:18:45
               Copyright 1988,1989,1991 SRI International
                      2002 Nov 21 Thu 15:32:15
BOBJ> in stack
==========================================
obj STACK
BOBJ> in stackscore
==========================================
open STACK
==========================================
op s : -> Stack .
==========================================
op e1 : -> Element .
==========================================
op e2 : -> Element .
==========================================
***> demonstrate some basic properties of stacks
==========================================
reduce in STACK : pop(pop(push(e1,push(e2,s)))) == s
rewrites: 3
result Bool: true
==========================================
reduce in STACK : top(pop(push(e1,push(e2,s)))) == e2
rewrites: 3
result Bool: true
==========================================
close
BOBJ>
```

Exercise 3 *Build a proof score that uses the* `reduce` *command to demonstrate that a top of a pop of a push is just the top of the original stack, i.e., demonstrate that top(pop(push(e,s))) = top(s).*

A Queue Specification

Continuing with our notion of software specification, let's specify a queue. A queue is a list-like data structure but the operations on this list are constrained: you are only able to add elements to the end of the list and you can only remove elements from the beginning of the list. An BOBJ specification of a queue might look somehting like the following:

```
obj QUEUE is
  sort Queue .
  sort Element .
  op peek : Queue -> Element .
  op remove : Queue -> Queue .
  op add : Element Queue -> Queue .
  op cons : Element Queue -> Queue .
  op nil : -> Queue .
  var Q : Queue .
  vars E E1 E2 : Element .
  eq peek(cons(E,Q)) = E .
  eq remove(cons(E,Q)) = Q .
  eq add(E1,cons(E2,Q)) = cons(E2,add(E1,Q)) .
  eq add(E,nil) = cons(E,nil) .
endo
```

What is noteworthy about the specification is that we represent the internal list structure with the `cons` operation symbol. An empty queue is represented with the `nil` operation symbol. We have three functions that manipulate the queue, namely: `peek` which returns the first element in the queue, `remove` which removes the first element of the queue, and `add` which adds a new element to the end of the queue.

Like in the previous example, we do all our demonstration and proving in a proof score.

```
*** proof score
open QUEUE .

op q : -> Queue .
op q' : -> Queue .
op e : -> Element .
op e' : -> Element .

***> show some basic properties of queues
reduce add(e,cons(e',nil)) == cons(e',cons(e,nil)) .
reduce add(e,cons(e',q)) == cons(e',add(e,q)) .
reduce remove(add(e,cons(e',q))) == add(e,q) .

***> assume that all queues q under consideration are non-empty
```

```
eq q = cons(e,q') .

***> prove that remove and add can be swapped on non-empty queues
reduce remove(add(e,q)) == add(e,remove(q)) .

close
```

Here is a session with the queue specification and proof score:

```
[lutz]$ java -jar bobj.jar
                     \|||||||||||||||||/
                   --- Welcome to BOBJ ---
                     /|||||||||||||||||\
         BOBJ version 2.0  built: 2000 Jun 20 Tue 20:18:45
            Copyright 1988,1989,1991 SRI International
                   2002 Nov 21 Thu 16:18:04
BOBJ> in queue
==========================================
obj QUEUE
BOBJ> in queuescore
==========================================
open QUEUE
==========================================
op q : -> Queue
==========================================
op q' : -> Queue
==========================================
op e : -> Element
==========================================
op e' : -> Element
==========================================
***>  show some basic properties of queues
==========================================
reduce in QUEUE : add(e, cons(e', nil)) == cons(e', cons(e, nil))
result Bool: true
rewrite time: 43ms       parse time: 18ms
==========================================
reduce in QUEUE : add(e, cons(e', q)) == cons(e', add(e, q))
result Bool: true
rewrite time: 1ms       parse time: 29ms
==========================================
reduce in QUEUE : remove(add(e, cons(e', q))) == add(e, q)
result Bool: true
rewrite time: 1ms       parse time: 4ms
==========================================
***>  assume that all queues q under consideration are non-empty
==========================================
eq q = cons(e, q')
==========================================
***>  prove that remove and add can be swapped on non-empty queues
==========================================
reduce in QUEUE : remove(add(e, q)) == add(e, remove(q))
result Bool: true
rewrite time: 8ms       parse time: 5ms
==========================================
close
BOBJ>
```

Exercise 4 *Show that* `peek` *is invariant under the* `add` *operation for all non-empty queues.*

A Calculator Specification

As our final example we take a look at the specification of a postfix style calculator. The idea is that the calculator has four buttons: enter, add, sub, and display, which let you enter a value into the calculator, perform an addition, perform a subtraction, and display a value, respectively. The central part of the specification is the notion of a state that can remember values that were entered. Here is the specification of the calculator:

```
obj CALCULATOR is
  protecting INT .
  sort State .
  op remember : Int State -> State .
  op enter : Int State -> State .
  op display : State -> Int .
  op add : State -> State .
  op sub : State -> State .
  vars I I1 I2 : Int .
  var S : State .
  eq enter(I,S) = remember(I,S) .
  eq display(remember(I,S)) = I .
  eq add(remember(I1,remember(I2,S))) = remember(I1 + I2,S) .
  eq sub(remember(I1,remember(I2,S))) = remember(I2 - I1,S) .
endo
```

We see that entering a value simply means remembering this value in the internal state of the calculator. We also see that displaying a value is simply removing a remembered value from the internal state and printing it out. What is striking about adding and subtracting is the fact that as binary operations they remove the two most recently remembered values from the calculator state and then add the sum or difference of the two values, respectively, back to the calculator state.

As before we use the notion of a proof score to demonstrate properties of this specification:

```
*** proof score

open CALCULATOR .

op l : -> Int .
op m : -> Int .
op n : -> Int .
op s : -> State .

***> demonstrate some system properties
reduce display(enter(3,s)) == 3 .
reduce display(add(enter(3,enter(2,s)))) == 5 .
reduce display(add(enter(m,enter(n,s)))) == m + n .

***> prove associativity of add for all integer values l, m, n
reduce add(add(remember(l,remember(m,remember(n,s))))) ==
        add(remember(l,add(remember(m,remember(n,s))))) .
close
```

Here is a sample session with the calculator specification and proof score:

```
[lutz]$ java -jar bobj.jar
                     \|||||||||||||||||/
                   --- Welcome to BOBJ ---
                     /|||||||||||||||||\
         BOBJ version 2.0  built: 2000 Jun 20 Tue 20:18:45
            Copyright 1988,1989,1991 SRI International
                   2002 Nov 21 Thu 16:36:05
BOBJ> in calc
==========================================
obj CALCULATOR
BOBJ> in calcscore
==========================================
open CALCULATOR
==========================================
op l : -> Int
==========================================
op m : -> Int
==========================================
op n : -> Int
==========================================
op s : -> State
==========================================
***>  demonstrate some system properties
```

```
==========================================
reduce in CALCULATOR : display(enter(3, s)) == 3
result Bool: true
rewrite time: 40ms       parse time: 6ms
==========================================
reduce in CALCULATOR : display(add(enter(3, enter(2, s)))) == 5
result Bool: true
rewrite time: 7ms       parse time: 4ms
==========================================
reduce in CALCULATOR : display(add(enter(m, enter(n, s)))) == m + n
result Bool: true
rewrite time: 5ms       parse time: 29ms
==========================================
***>  prove associativity of add for all integer values l, m, n
==========================================
reduce in CALCULATOR : add(add(remember(l, remember(m, remember(n, s)))))
    == add(remember(l, add(remember(m, remember(n, s)))))
result Bool: true
rewrite time: 7ms       parse time: 5ms
==========================================
close
BOBJ>
```

Exercise 5 *Prove the commutativity of* `add`*.*