dbcbet: Object-Oriented Design by Contract with Declarative Bounded Exhaustive Testing

Christopher Coakley

and Peter Cappello

Overview

- Background
  - Design by Contract
  - Bounded Exhaustive Testing (Korat)
- dbcbet
  - *Object Oriented* Design by Contract
  - *Declarative* Bounded Exhaustive Testing

BACKGROUND

Defensive Programming

```
def sqrt(x):
    assert x >= 0
    result = <sqrt implementation>
    assert result*result == x
    return result
```

Defensive Programming

```
def sqrt(x):
    assert x >= 0
    result = <sqrt implementation>
    assert result*result == x
    return result
```

Precondition

Postcondition

Design by Contract is:

- Defensive Programming
- Extra Syntax
- Assignment of Blame
- Inheritance Rules

Design by Contract

- Est. 1986, Bertrand Meyer, Eiffel
- Precondition – Violation = bug in caller
- Postcondition – Violation = bug in callee
- Invariant – Violation = bug in callee
- Liskov Substitution Principle: Subclass may
  - *Weaken* preconditions
  - *Strengthen* postconditions & invariants

Design by Contract

- Native support for contracts:

  Spec#, Clojure, Racket, Eiffel

- Library/Tool support for contracts:

  C++, Java, Python, Ruby

- Eiffel makes guarantees about exception types

  These guarantees typically *are not part of contract*

- Contracts are *orthogonal to the type system*

Bounded Exhaustive Testing (BET)

- Test *a model* of the software:
  - TestEra
  - Alloy
- Test *the code*:
  - Korat

    Finitization – a finite bound on the input domain

How Korat Works

- Requires a finitization for all object fields
- Generates all possible field/value assignments

  Cross-product of field finitization sets

- If element in cross product passes class invariants, object is valid

How to Test Methods

- Korat delivers valid test *objects*
- Command Pattern: treat *methods* as *objects*
  - Self & parameters become fields
  - Call execute() on each valid Command object

DBCBET

Goals

- Design by Contract
  - *Reusable* contract *components* as 1st-class *objects*
  - Do *not* depend on the *metaclass*
    - Only 1 metaclass per class
    - SQLAlchemy uses metaclass
- Bounded Exhaustive Testing

  Declarative syntax

Use Python Decorators

Python *decorator*

```
@foo
def bar(): pass
```

Is *equivalent* to

```
def bar(): pass
bar = foo(bar)
```

foo can *redefine* bar

Syntax Example

```
@inv(invariant)
class BusMonitor( object ):
    @pre( precondition1 )
    @pre( precondition2 )
    @post( postcondition )
    @throws( IOError, CustomException )
    @finitize_method( [device(1), device(2)],range(-1, 10) )
    def attach( self, device, priority ):
        …
```

How it works

- First *applied* contract component (@pre, @post, @throws, @inv) wraps method with contract invoker
- Each contract component creates/appends list of preconditions, postconditions, invariants, & throws
- Inheritance is managed by @inv or @dbc
- Postcondition special parameter: *old*

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Invoker

```
def create_invoker(method):
    """invoker checks all contract components and invokes the method."""
    @wraps(method)
    def invoker(s, *args, **kwargs):
        check_preconditions(wrapped_method, s, *args, **kwargs)
        o = old(method, s, args, kwargs)
        try:
            ret = method(s, *args, **kwargs)
            check_postconditions(wrapped_method, s, o, ret, *args, **kwargs)
            check_invariants(wrapped_method, s, *args, **kwargs)
        except Exception as ex:
            if check_throws(wrapped_method, ex, s, *args, **kwargs):
                raise
        return ret
    return invoker
```

Contract Components are Objects

- Predicates can provide *custom error messages*
- Enables *stateful* guards
  - Ex. precondition: *Cannot call me more than once*
  - *State is contract's*, not guarded object's
- Composable & Resusable

  dbcbet.helpers library has *reusable primitive* contract components.

Finitization

- A python dict for classes

  { fieldname: [assignments] }

- A sequence of sequences for methods

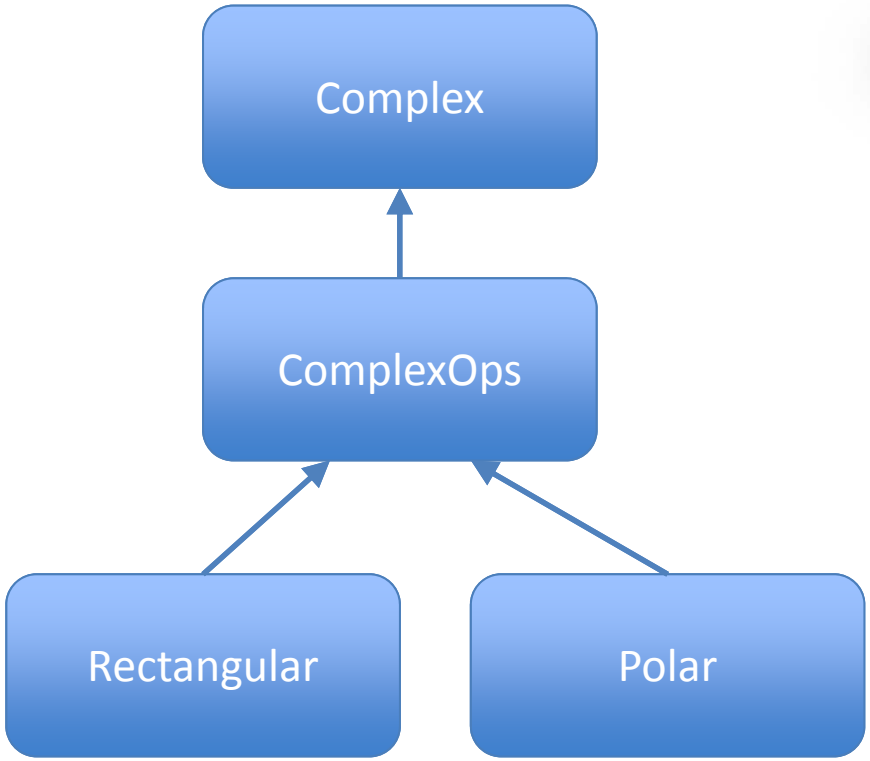
  Positional arguments

- Syntax:

  @finitize({'re':xrange(-1,1), 'img':[None,-1,0,1]})

  @finitize_method(enumerate(Person))

Ported JML Example



Some Contract Definitions

```
def real_part_post( self, old, ret ):
    return approx_equal( self._magnitude()
        * math.cos( self._angle() ), ret, tolerance )

def angle_post( self, old, ret ):
    return approx_equal( math.atan2( self._imaginary_part(),
        self._real_part()), ret, tolerance )

def arg_not_none( self, b ):
    """This is a custom error message"""
    return b is not None
```

Example

```
@dbc
class Complex( object ):
    @post( real_part_post )
    def real_part( self ): pass

    @post( imaginary_part_post )
    def imaginary_part( self ): pass

    @post( magnitude_post )
    def magnitude( self ): pass

    @post( angle_post )
    def angle( self ): pass
```

Example

```
@dbc
class Complex( object ):
    @post( real_part_post )
    def real_part( self ): pass

    @post( imaginary_part_post )
    def imaginary_part( self ): pass

    @post( magnitude_post )
    def magnitude( self ): pass

    @post( angle_post )
    def angle( self ): pass
```

Example

```
@dbc
class Complex( object ):
    @post( real_part_post )
    def real_part( self ): pass

    @post( imaginary_part_post )
    def imaginary_part( self ): pass

    @post( magnitude_post )
    def magnitude( self ): pass

    @post( angle_post )
    def angle( self ): pass
```

```
@dbc
class ComplexOps( Complex ):
    @pre( argument_types( Complex ) )
    @post( add_post )
    @finitize_method( complex_gen() )
    def add( self, b ):
        return Rectangular( self.real_part()
            + b.real_part(), self.imaginary_part() +
            b.imaginary_part() )

    @post( mul_post )
    @finitize_method( complex_gen() )
    def mul( self, b ):
        try:
            return Polar( self.magnitude() * b.magnitude(),
                self.angle() + b.angle() )
        except ValueError:
            return Rectangular( float( 'nan' ) ) )
```

```
@dbc
class ComplexOps( Complex ):
    @pre( argument_types( Complex ) )
    @post( add_post )
    @finitize_method( complex_gen() )
    def add( self, b ):
        return Rectangular( self.real_part()
            + b.real_part(), self.imaginary_part() +
            b.imaginary_part() )

    @post( mul_post )
    @finitize_method( complex_gen() )
    def mul( self, b ):
        try:
            return Polar( self.magnitude() * b.magnitude(),
                self.angle() + b.angle() )
        except ValueError:
            return Rectangular( float( 'nan' ) ) )
```

```
@dbc
class ComplexOps( Complex ):
    @pre( argument_types( Complex ) )
    @post( add_post )
    @finitize_method( complex_gen() )
    def add( self, b ):
        return Rectangular( self.real_part()
            + b.real_part(), self.imaginary_part() +
            b.imaginary_part() )

    @post( mul_post )
    @finitize_method( complex_gen() )
    def mul( self, b ):
        try:
            return Polar( self.magnitude() * b.magnitude(),
                self.angle() + b.angle() )
        except ValueError:
            return Rectangular( float( 'nan' ) ) )
```

```
@dbc
class ComplexOps( Complex ):
    @pre( argument_types( Complex ) )
    @post( add_post )
    @finitize_method( complex_gen() )
    def add( self, b ):
        return Rectangular( self.real_part()
            + b.real_part(), self.imaginary_part() +
            b.imaginary_part() )

    @post( mul_post )
    @finitize_method( complex_gen() )
    def mul( self, b ):
        try:
            return Polar( self.magnitude() * b.magnitude(),
                self.angle() + b.angle() )
        except ValueError:
            return Rectangular( float( 'nan' ) ) )
```

```
@dbc
class ComplexOps( Complex ):
    @pre( argument_types( Complex ) )
    @post( add_post )
    @finitize_method( complex_gen() )
    def add( self, b ):
        return Rectangular( self.real_part()
            + b.real_part(), self.imaginary_part() +
            b.imaginary_part() )

    @post( mul_post )
    @finitize_method( complex_gen() )
    def mul( self, b ):
        try:
            return Polar( self.magnitude() * b.magnitude(),
                self.angle() + b.angle() )
        except ValueError:
            return Rectangular( float( 'nan' ) ) )
```

```
@inv( polar_invariant )
@finitize( finitize_polar )
class Polar( ComplexOps ):
    @pre( argument_types( Number, Number ) )
    @finitize_method( [-1,0,1],
        [-math.pi,0,math.pi/4.0,math.pi/2.0] )
    @throws( ValueError )
    def __init__( self, mag, angle ):
        if math.isnan( mag ):
            raise ValueError()
        if mag < 0:
            mag = -mag;
            angle += math.pi;
        self.mag = mag;
        self.ang = standardize_angle( angle )

    def _real_part( self ):
        return self.mag * math.cos( self.ang )

    # specification inherited
    real_part = _real_part
```

```
@inv( polar_invariant )
@finitize( finitize_polar )
class Polar( ComplexOps ):
    @pre( argument_types( Number, Number ) )
    @finitize_method( [-1,0,1],
        [-math.pi,0,math.pi/4.0,math.pi/2.0] )
    @throws( ValueError )
    def __init__( self, mag, angle ):
        if math.isnan( mag ):
            raise ValueError()
        if mag < 0:
            mag = -mag;
            angle += math.pi;
        self.mag = mag;
        self.ang = standardize_angle( angle )

    def _real_part( self ):
        return self.mag * math.cos( self.ang )

    # specification inherited
    real_part = _real_part
```

```
@inv( polar_invariant )
@finitize( finitize_polar )
class Polar( ComplexOps ):
    @pre( argument_types( Number, Number ) )
    @finitize_method( [-1,0,1],
        [-math.pi,0,math.pi/4.0,math.pi/2.0] )
    @throws( ValueError )
    def __init__( self, mag, angle ):
        if math.isnan( mag ):
            raise ValueError()
        if mag < 0:
            mag = -mag;
            angle += math.pi;
        self.mag = mag;
        self.ang = standardize_angle( angle )

    def _real_part( self ):
        return self.mag * math.cos( self.ang )

    # specification inherited
    real_part = _real_part
```

```
@inv( polar_invariant )
@finitize( finitize_polar )
class Polar( ComplexOps ):
    @pre( argument_types( Number, Number ) )
    @finitize_method( [-1,0,1],
        [-math.pi,0,math.pi/4.0,math.pi/2.0] )
    @throws( ValueError )
    def __init__( self, mag, angle ):
        if math.isnan( mag ):
            raise ValueError()
        if mag < 0:
            mag = -mag;
            angle += math.pi;
        self.mag = mag;
        self.ang = standardize_angle( angle )

    def _real_part( self ):
        return self.mag * math.cos( self.ang )

    # specification inherited
    real_part = _real_part
```

```
@inv( polar_invariant )
@finitize( finitize_polar )
class Polar( ComplexOps ):
    @pre( argument_types( Number, Number ) )
    @finitize_method( [-1,0,1],
        [-math.pi,0,math.pi/4.0,math.pi/2.0] )
    @throws( ValueError )
    def __init__( self, mag, angle ):
        if math.isnan( mag ):
            raise ValueError()
        if mag < 0:
            mag = -mag;
            angle += math.pi;
        self.mag = mag;
        self.ang = standardize_angle( angle )

    def _real_part( self ):
        return self.mag * math.cos( self.ang )

    # specification inherited
    real_part = _real_part
```

```
@inv( polar_invariant )
@finitize( finitize_polar )
class Polar( ComplexOps ):
    @pre( argument_types( Number, Number ) )
    @finitize_method( [-1,0,1],
        [-math.pi,0,math.pi/4.0,math.pi/2.0] )
    @throws( ValueError )
    def __init__( self, mag, angle ):
        if math.isnan( mag ):
            raise ValueError()
        if mag < 0:
            mag = -mag;
            angle += math.pi;
        self.mag = mag;
        self.ang = standardize_angle( angle )
```

```
def _real_part( self ):
    return self.mag * math.cos( self.ang )

# specification inherited
real_part = _real_part
```

Helper Examples

```
def immutable( self, old, ret, *args, **kwargs ):
    """Object immutability was violated by the
  method call (did you forget to override
  __eq__?)"""
    return old.self == self

# use: @post( immutable )
```

```
class argument_types( object ):
    """DBC helper for reusable, simple predicates for
   argument-type tests used in preconditions"""
    def __init__( self, *typelist ):
        self.typelist = typelist
        self.msg = "implementation error in argument_types"

    def __call__( self, s, *args, **kwargs ):
        for typ, arg in zip( self.typelist, args ):
            if not isinstance( arg, typ ) and arg is not None:
                self.msg = "argument %s was not of type %s"
                    % ( arg, typ.__name__ )
                return False
        return True

    def error( self ):
        return self.msg

# use: @pre( argument_types( Number, Number ) )
```

```
class argument_types( object ):
    """DBC helper for reusable, simple predicates for
   argument-type tests used in preconditions"""
    def __init__( self, *typelist ):
        self.typelist = typelist
        self.msg = "implementation error in argument_types"

    def __call__( self, s, *args, **kwargs ):
        for typ, arg in zip( self.typelist, args ):
            if not isinstance( arg, typ ) and arg is not None:
                self.msg = "argument %s was not of type %s"
                    % ( arg, typ.__name__ )
                return False
        return True

    def error( self ):
        return self.msg

# use: @pre( argument_types( Number, Number ) )
```

```
class argument_types( object ):
    """DBC helper for reusable, simple predicates for
   argument-type tests used in preconditions"""
    def __init__( self, *typelist ):
        self.typelist = typelist
        self.msg = "implementation error in argument_types"

    def __call__( self, s, *args, **kwargs ):
        for typ, arg in zip( self.typelist, args ):
            if not isinstance( arg, typ ) and arg is not None:
                self.msg = "argument %s was not of type %s"
                    % ( arg, typ.__name__ )
                return False
        return True

    def error( self ):
        return self.msg

# use: @pre( argument_types( Number, Number ) )
```

```
class argument_types( object ):
    """DBC helper for reusable, simple predicates for
   argument-type tests used in preconditions"""
    def __init__( self, *typelist ):
        self.typelist = typelist
        self.msg = "implementation error in argument_types"

    def __call__( self, s, *args, **kwargs ):
        for typ, arg in zip( self.typelist, args ):
            if not isinstance( arg, typ ) and arg is not None:
                self.msg = "argument %s was not of type %s"
                    % ( arg, typ.__name__ )
                return False
        return True
```

```
    def error( self ):
        return self.msg
```

```
# use: @pre( argument_types( Number, Number ) )
```

Testing

```
>>> from dbcbet import bet
>>> for typ in [Polar, Rectangular]:
...     bet(typ).run()
Summary:
 Instance Candidates: 12
 Invariant Violations: 0
 Method Call Candidates: 180
 Precondition Violations: 0
 Failures: 0
 Successes: 180

Summary:
 Instance Candidates: 30
 Invariant Violations: 0
 Method Call Candidates: 570
 Precondition Violations: 0
 Failures: 0
 Successes: 570
```

42 instances
750 tests
real 0m0.286s
user 0m0.255s
sys 0m0.021s

258 instances
4854 tests
real 0m1.376s
user 0m1.338s
sys 0m0.022s

How bet.run works

- *Construct* an Object

  Class or constructor finitization

- For each method, *construct* an argument list
- If precondition fails, skip
- *Execute* method
- *Test succeeds*, if postcondition & invariant hold
- Do this for cross-product of

  objects X methods X arguments

Future Work

- Rewrite BET code to use object pooling

  Makes testing self-referential structures significantly easier

- Eliminate helpers like enumerate(Person)
- Higher-order contracts

  (e.g., if a method takes a function *f* as a parameter, issue contract on *f's* contract (e.g., *f must have postcondition* "must return None"))

References

- Meyer, Bertrand: *Design by Contract, TR-EI-12/CO,* Interactive Software Engineering Inc., 1986
- pyContract: http://www.wayforward.net/pycontract/
- pyDBC: http://www.nongnu.org/pydbc/
- Gary T. Leavens, Yoonsik Cheon. Design by Contract with JML., 2006
- Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In *Proceedings of the 29th international conference on Software Engineering (ICSE '07).*

Thank You

Code available at:

https://github.com/ccoakley/dbcbet

BACKUP SLIDES

Existing Contract Libraries

- Python
  - pyContract
  - pyDBC

pyContract

- Contracts are part of documentation strings
- PEP 316

```
def sort(a):
    """Sort a list *IN PLACE*.

    pre:
        # must be a list
        isinstance(a, list)

        # all elements must be comparable with all other items
        forall(range(len(a)),
               lambda i: forall(range(len(a)),
                                lambda j: (a[i] < a[j]) ^ (a[i] >= a[j])))

    post[a]:
        # length of array is unchanged
        len(a) == len(__old__.a)

        # all elements given are still in the array
        forall(__old__.a, lambda e: __old__.a.count(e) == a.count(e))

        # the array is sorted
        forall([a[i] >= a[i-1] for i in range(1, len(a))])
    """
```

pyDBC

- Metaclass based
  - Metaclasses are inherited properly
  - pyDBC inheritance works properly (it was fixed)
- Separate methods for contracts
  - non-reusable due to naming requirements

```
import dbc
  __metaclass__ = dbc.DBC

  class Foo:
      def __invar(self):
          assert isinstance(self.a, int)

      def __init__(self, a):
          self.a = a

      def foo(self, a):
          self.a *= a

      def foo__pre(self, a):
          assert a > 0

      def foo__post(self, rval):
          assert rval is None
```