Synthesize it: from Design by Contract™ to Meaningful Test Input Data

Stefan J. Galler, Martin Weiglhofer and Franz Wotawa
Synthia generates jUnit tests that use automatically synthesized Fake objects instead of real objects as parameters.
Synthia generates test data based on Design by Contract specification

<table>
<thead>
<tr>
<th>White Box</th>
<th>PEX, JPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design by Contract</td>
<td>Spec Explorer, PEX, jTest</td>
</tr>
<tr>
<td>Black Box</td>
<td>TGV, Overture (VDM), ARTOO, JET</td>
</tr>
</tbody>
</table>

Test sequence | Test data |

Imagine we have to test a method that requires at least two elements on a Stack

```java
@Pre("stack.size()>=2")
@Post("stack.size()==1")
public void sumUp(Stack stack) {
    int sum = 0;
    while (stack.size() > 0) {
        sum += stack.pop();
    }
    stack.push(sum);
}
```

A Synthia Fake behavior is determined by the Design by Contract specification

Cut/MUT Parameter

Synthia Fake

jUnit Test

set initial values

Synthia is evaluated on two case studies

stackcalc BillingSoftware
Synthia generates test data based on Design by Contract specification

<table>
<thead>
<tr>
<th>White Box</th>
<th>Design by Contract</th>
<th>Black Box</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spec Explorer</td>
<td>TGV, Overture (VDM)</td>
</tr>
<tr>
<td></td>
<td>PEX, JPF</td>
<td>ARTOO, JET</td>
</tr>
<tr>
<td></td>
<td>PEX, jTest</td>
<td>Test data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test sequence</td>
</tr>
</tbody>
</table>
Synthia generates test data based on Design by Contract specification

<table>
<thead>
<tr>
<th>White Box</th>
<th>Design by Contract</th>
<th>Black Box</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spec Explorer</td>
<td>TGV, Overture (VDM)</td>
</tr>
<tr>
<td></td>
<td>PEX, JPF</td>
<td>ARTOO, JET</td>
</tr>
</tbody>
</table>

Test sequence | Test data
Design by Contract is a software concept similar to legal contracts in real world.

Design by Contract is an incomplete specification for classes and methods.

A Fake Object is the most realistic replacement for a real object

- **Dummy**
  - Passes bogus input values around

- **Stub**
  - Returns hard coded values

- **Mock**
  - Focuses on Interaction

- **Spy**
  - Focuses on the objects state

- **Fake**
  - Swaps out real implementation
Imagine we have to test a method that requires at least two elements on a Stack

```java
class Stack {
    @Post("size()==@Old(size())+1")
    public void push(int p1) {
    }
    @Pure
    @Post("@Return>=0")
    public int size() {
    }
}

@Pre("stack.size()>=2")
@Post("stack.size()==1")
public void sumUp(Stack stack) {
    int sum = 0;
    while(stack.size()>0) {
        sum += stack.pop();
    }
    stack.push(sum);
}
```

```java
public class Stack {
    @Post("size()==@Old(size())+1")
    public void push(int p1) {
    }
    @Pure
    @Post("@Return>=0")
    public int size() {
    }
}
```
2,5 out of 1000 randomly generated Stack objects have at least 2 elements

```java
@Test
public void testSumUp() {
    Operator receiver = SumUpOp();
    Stack stack = new Stack();
    stack.push(54);
    stack.push(8432);
    receiver.sumUp(stack);
}
```
Manually testing `sumUp()` would feature mock objects to satisfy precondition

```java
@Test
public void testSumUp() {
    Operator receiver = SumUpOp();
    Stack stack =
        new EasyMock.create(Stack);
    expect(stack.size()).andReturn(2);
    replay(stack);
    receiver.sumUp(stack);
}
```

Static behaviour of mock leads to false failing test cases

```java
@Pre("stack.size()>=2")
@Post("stack.size()==1")
public void sumUp(Stack stack) {
    int sum = 0;
    while(stack.size()>0) {
        sum += stack.pop();
    }
    stack.push(sum);
}

expect(stack.size()).and Return(2)
```

- **Precondition satisfied**
- **while expression satisfied**
- **Update sum**
- **While expression satisfied**
- **...**
Synthia synthesizes the behaviour of the mock from Design by Contract specification

```java
@Test
public void testSumUp() {
    Operator receiver = SumUpOp();
    Stack stack =
        new Synthia<Stack>();
    Map init = {size=2};
    stack.setInitialState(init);
    receiver.sumUp(stack);
}
```
Synthia generates Fakes which replace objects in unit tests

A Synthia Fake behavior is determined by the Design by Contract specification

- **CUT/MUT**
- **Parameter**
- **jUnit Test**
- **Synthia Fake**

Set initial values
A Synthia Fake calculates the (public observable) state change at runtime

size() := 2
...
pre
size()
size() == @Old(size()) - 1

post
size() := 1
...

CUT/MUT
Synthia Fake
SMT Solver (yices, Z3)
Synthia Fake handles incomplete specification written in Java

- **Framing**
  - Unreferenced variables keep their values

- **Chaining**
  - Method calls within specification

**Design-by-Contract keywords**
- @Pure
- @Return
Synthia Fake assumes that values are kept if not stated otherwise

```java
@Pure
@Post("@Return>=0")
public int size() {...}
```

```java
@Pure
@Post("@Return>=0 && size()==@Old(size())")
public int size() {...}
```
Chaining includes specification of referenced methods

```java
@Post("size() == @Old(size()) - 1")
public void pop(int p1){...}
```

```java
@Pure
@Post("@Return>=0")
public int size() {...}
```

```
size() >= 0 ;
size() == @Old(size()) - 1
; size() >= 0
```
Synthia uses the chained framed specification for behavior calculation.

```java
@Post("size()==@Old(size())-1")
public void pop(int p1){...}

@Pure
@Post("@Return>=0")
public int size() {...}

size()>=0 && size()==@Old(size()) ;
size()==@Old(size())-1 ;
size()>=0 && size()==@Old(size())
```
Synthia is evaluated on two case studies:

- stackcalc
- BillingSoftware
The stackcalc case study was developed and later specified by students

- 868 NCSS
- 34 out of 147 methods require non-primitive data types as parameters
The BillingSoftware is provided by our industry partner

- 1456 NCSS
- 71 out of 249 methods require non-primitive data types as parameters
- First specified then implemented
- Non-blocking queue had to be replaced
We use a random approach (JET) to benchmark Synthia

- **create**
  - Create input data for each method of all classes of the case study

- **classify**
  - Execute test online to classify the result
    - Meaningless tests: precondition violation

- **export**
  - Export all non-meaningless tests to jUnit
Each data point is an average over 300 measurements.

Union over 4 tries to generate meaningful test input data.
Synthia is able to test more methods in less time.

[Graph showing that Synthia is able to test more methods in less time than Random, with a -30% improvement in generation time for a given level of function coverage.]
Strengthening preconditions of constructors boosted Synthia results

function coverage

+100%

Random (1) and (2)
Synthia (1)
Synthia (2)

0 1000 2000
generation time in ms
Test execution time depends on the interaction of the test with Synthia Fakes

<table>
<thead>
<tr>
<th>Test execution</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Random [sec]</td>
<td>0.24</td>
<td>0.53</td>
</tr>
<tr>
<td>Synthia [sec]</td>
<td>0.47</td>
<td>0.47</td>
</tr>
</tbody>
</table>

-11% to +95%
Synthia tests need about 24% more time for generation

<table>
<thead>
<tr>
<th>Test generation</th>
<th>Random [sec]</th>
<th>Synthia [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random [sec]</td>
<td>1.77</td>
<td>4.1</td>
</tr>
<tr>
<td>Synthia [sec]</td>
<td>2.12</td>
<td>5.11</td>
</tr>
</tbody>
</table>

~ +24%
Thank you for your attention!

A Synthia Fake behavior is determined by the Design by Contract specification.

- CUT/MUT
- Parameter
- jUnit Test
- Synthia Fake

Strengthening preconditions of constructors boosted Synthia results.