Formal Verification Tutorial

INTRODUCTION page 1

Software testing can reveal some errors, but there is no way to prove that the software is absolutely correct in every case. However, there is a way to **verify** programs by *mathematically proving that the program's post-condition* will hold as long as the precondition holds.

Included with the code for the method to be verified will be the specification for the code that shows the pre- and post-conditions. Along with that, the formal specification for the model being used will be required in order to complete the verification process.

A simple example is shown next. (- - > next page)

page 2

The method shown below declares a variable `x` of type `Object` to be used to store the result of a pop from a Stack, and then pushes that object back onto the Stack. Since this code involves a Stack, the formal specification for a Stack is required in order to verify the code, so it is helpful to have that to refer to.

Also note the pre- and post-conditions that are specified. The pre-condition requires that the beginning stack is non-empty; the post-condition says that when the code finishes, the resulting stack should be the same as it was before the code executed.

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      x = s.pop();
      s.push(x);
}
```

There are two approaches to code verification:

1. Table Approach
2. Goal-Oriented Approach

(- - > user chooses one of these tutorials)

CHOICE 1: TABLE APPROACH page 3

With the Table Approach, the first thing to do is to label states. State 0 is the state after the first line of code is executed. After the second line of code is executed would be State 1, etc. The code from the previous page with the states labeled is shown below:

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      // STATE 0: after first line of code

      x = s.pop();
      // STATE 1: after second line of code

      s.push(x);
      // STATE 2: after third line of code
}
```

(- - > next page)

After the states are labeled, we now know that there are 3 states in all, so we know how big to make the table:

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0		
State 1		
State 2		

The next step is to start filling in the table.

(- - > next page)

page 5

Right away, you can fill in the very first box (with what is stated in the pre-condition) and the very last box (with what is shown in the post-condition):

- In *State 0* under the *Facts*, the pre-condition `|s| > 0` can be filled in, including the state number
- Also in *State 0* under the *Facts*, we have the variable `x` which has been declared, so it can be stated that x now exists or that x is null, and since it is State 0, we will denote it as either x_0 exists or $x_0 = null$.
- In *State* 2 under the *Obligations*, the post-condition `s = #s` can be filled in. That can be understood to mean that the final stack is equal to the beginning stack. Now that we know that *State 2* is the final state, we can rewrite the post-condition slightly so as to include the correct state numbers: $s_2 = s_0$.

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      // STATE 0: after first line of code

      x = s.pop();
      // STATE 1: after second line of code

      s.push(x);
      // STATE 2: after third line of code
}
```

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0	**$\|s_0\| > 0$ (pre-condition)** **$x_0 = null$**	
State 1		
State 2		**$s_2 = s_0$ (post-condition)**

(- - > next page)

page 6

Before moving into *State 1*, which involves a `pop()`, the next box to fill in is the first box under *Obligations*.

Since a `pop()` operation has a pre-condition (also called a requires clause) that says that the length of the stack must be greater than 0, that is what must be shown to be true before moving into *State 1*. This is where having the specification for Stack comes in handy. That part of that specification is shown below:

```
T pop();
      /*!
       *  precondition:
       *    |self| > 0
       *  postcondition:
       *    #self = self * <pop()>
      !*/
```

Again, the code that we are working with is restated:

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      // STATE 0: after first line of code

      x = s.pop();
      // STATE 1: after second line of code

      s.push(x);
      // STATE 2: after third line of code
}
```

The addition to the table – the first box under *Obligations* - is shown in red below. The item under *Obligations* that needs to be proved turns out to be the same as the pre-condition that was given, so it is sufficient to just re-state it.

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0	$\|s_0\| > 0$ (pre-condition) $x_0 = \text{null}$	**$\|s_0\| > 0$ (given)**
State 1		
State 2		$s_2 = s_0$

(- - > next page)

The line of code which will result in *State 1* involves a `pop()`. Again, this is where the specification for Stack comes in handy. You will notice that after a `pop()`, a post-condition exists:

```
T pop();
      /*!
       *  precondition:
       *    |self| > 0
       *  postcondition:
       *    #self = self * <pop()>
      !*/
```

This is what will be put in the table under the *Facts* for *State 1*, rewritten to reflect the code that we are validating: the previous self, which would be s_0, equals the current self, or s_1, concatenated with the string containing the item being popped off the stack, which would be $\langle x_1 \rangle$, or: $s_0 = s_1 * \langle x_1 \rangle$ (shown in the table below in red).

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      // STATE 0: after first line of code

      x = s.pop();
      // STATE 1: after second line of code

      s.push(x);
      // STATE 2: after third line of code
}
```

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0	$\|s_0\| > 0$ (pre-condition) x_0 = null	$\|s_0\| > 0$ (given)
State 1	**$s_0 = s_1 * \langle x_1 \rangle$**	
State 2		$s_2 = s_0$

(- - > next page)

page 8

The line of code which will result in *State 2* involves a `push()`. Again, the specification for Stack comes in handy. You will notice that for a `push()`, there is no pre-condition:

```
void push (T e);
      /*!
       *
       * preserves: e
       * postcondition:
       *    self = #self * <e>
       *
      !*/
```

so, for the *Obligation* before *State 2*, you only need to write "true", as shown in red in the table below, meaning that there are no requirements that need to be met before pushing something on to the Stack.

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      // STATE 0: after first line of code

      x = s.pop();
      // STATE 1: after second line of code

      s.push(x);
      // STATE 2: after third line of code
}
```

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0	$\|s_0\| > 0$ (pre-condition) x_0 = null	$\|s_0\| > 0$ (given)
State 1	$s_0 = s_1 * \langle x_1 \rangle$	**true**
State 2		$s_2 = s_0$

(- - > next page)

Now that the *Obligation* before *State 2* has been met, we can move on to the *Fact* for *State 2*. Looking again at the specification for a `push()`:

```
void push (T e);
      /*!
       *
       * preserves: e
       * postcondition:
       *    self = #self * <e>
       *
      !*/
```

we can see that there is a post-condition, so after a `push()` – what would go in the *Fact* box in the table below - is the following: the current self equals the previous self concatenated with the string containing the item that is being pushed onto the stack: $s_2 = s_1 * \langle x_2 \rangle$ (as shown in red in the table below).

Also, since nothing has changed with the value of `x` itself, we will denote that as well by saying that $x_2 = x_1$, meaning the value of `x` in the current state is the same as it was in the previous state. This is known as the ***Frame Property***. The Frame Property can be used for any item that exists for which there was no change. It may or may not be needed in proving the other *Obligations*, but it is ok to have more than what is absolutely necessary. It is better to include under the *Facts* the things you think you may need for each *Obligation* in case it is needed.

```
public static void doNothing (Stack s) {
      /*
       *  pre:  |s| > 0
       *  post:  s = #s
       *
       */

      Object x;
      // STATE 0: after first line of code

      x = s.pop();
      // STATE 1: after second line of code

      s.push(x);
      // STATE 2: after third line of code
}
```

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0	$\|s_0\| > 0$ (pre-condition) $x_0 =$ null	$\|s_0\| > 0$ (given)
State 1	$s_0 = s_1 * \langle x_1 \rangle$	true
State 2	**$s_2 = s_1 * \langle x_2 \rangle$** **$x_2 = x_1$ (frame property)**	$s_2 = s_0$

(- - > next page)

Now, for the last box. In this example, the post-condition for the `doNothing()` method is that the final self is equal to the beginning self: `s = #s`, which we have already rewritten as: $s_2 = s_0$.

Remember we can only use *Facts* from the table when proving any of the *Obligations*, and we do not have $s_2 = s_0$. But, we do have the following: $s_2 = s_1 * \langle x_2 \rangle$, so we'll start with what we have. With substitutions and the use of results from string theory, we hope we can get to what we are trying to prove. The rest of the proof is explained in the last box of the table below:

STATE #	**FACTS** (things we know to be true)	**OBLIGATIONS** (what must be proven using previously known facts before moving on to the next state)
State 0	$\|s_0\| > 0$ (pre-condition) $x_0 = \text{null}$	$\|s_0\| > 0$ (given)
State 1	$s_0 = s_1 * \langle x_1 \rangle$	true
State 2	$s_2 = s_1 * \langle x_2 \rangle$ $x_2 = x_1$ (frame property)	**$s_2 = s_0$ (what we are trying to prove)** **$s_2 = s_1 * \langle x_2 \rangle$ (from State 2)** **since $x_2 = x_1$ (from State 2), we can substitute for x_2:** **$s_2 = s_1 * \langle x_1 \rangle$** **from State 1, we have:** **$s_0 = s_1 * \langle x_1 \rangle$** **therefore, $s_2 = s_0$**

Notice that alternatively, we could have done the following instead (either way would be correct):

$s_2 = s_1 * \langle x_2 \rangle$ (from State 2)

from State 1, we have:
$s_0 = s_1 * \langle x_1 \rangle$

substituting here for x_1, since $x_2 = x_1$:
$s_0 = s_1 * \langle x_2 \rangle$

therefore, $s_2 = s_0$