SmartFrog Assertion Components

Introduction

The assertion components in SmartFrog aim to provide a way of validating parts of the deployment descriptor. They also enable one to verify that methods invoked on referenced component instances return true or false, which can be used to incorporate more checks into the system's operation.

These assertions are not only checked when deployment takes place, they are revalidated whenever the liveness of the assertions are checked. When an assertion is pinged by its parent, it re-resolves all its references, and revalidates all its assertions. Thus assertions declare not only what constitutes a valid state for deployment, they declare that this state must remain valid after deployment.

When an assertion fails, a `SmartFrogAssertionException` is thrown, a derivative class of `SmartFrogException`. Within the `sfStart` method, this exception is passed up directly. During a liveness check, the exception is wrapped into a `SmartFrogLivenessException`, as this and `RemoteException` instances are the only types of exception that liveness checks can throw.

Component Declarations

The component file is included with the following line

```
#include "org/smartfrog/services/assertions/components.sf"
```

This declares a schema for the assertion component, and the component itself

```
/**
 * the schema for assertions
 */
AssertSchema extends Schema {
    //a fact that must resolve to true
    isTrue extends OptionalBoolean;
    //a fact that must be fale
    isFalse extends OptionalBoolean;
    //a string message
    message extends OptionalString;
    //the name of a reference which does not have to be defined, but when it
    //is, the evaluatesTrue and evaluatesFalse checks apply
    reference extends OptionalCD;
    //name of a boolean test() method that returns true when called on the reference
    evaluatesTrue extends OptionalString;
    //name of a boolean test() method that returns false when called on the reference
    evaluatesFalse extends OptionalString;
    //flag to enable startup checking (default: true)
    checkOnStartup extends OptionalBoolean;
    //flag to enable liveness checking (default: true)
    checkOnLiveness extends OptionalBoolean;
}

/**
 * the assert component can be used to declare invariants which
 * must hold for the system to be valid
 */
Assert extends Prim {
    assertSchema extends Schema;
    sfClass "org.smartfrog.services.assertions.AssertComponent";
    isTrue true;
    isFalse false;
    checkOnStartup true;
    checkOnLiveness true;
}
```

Attributes

`isTrue`	A Boolean that must resolve to true, if declared
`isFalse`	A Boolean that must resolve to false, if declared
`message`	An optional text message
`reference`	A name of an (optional) reference to be used in the evaluatesTrue/False tests
`evaluatesTrue`	name of a method on the reference that should evaluate true
`evaluatesFalse`	name of a method on the reference that should evaluate to false
`attribute`	the name of an attribute that, if declared, must be resolvable from the reference.
`fileExists`	the name of a file, that, if declared, must exist and be a normal file. No attempt is made to actually open the file for reading or writing.
`dirExists`	the name of a directory, that, if declared, must exist.
`checkOnStartup`	A flag to enable checking on the initial deployment (specifically, the `sfStart()` method)
`checkOnLiveness`	A flag to enable checking on every liveness check

Simple use: validating true/false statements

This example sets a reference to false, and then asserts that the value of the reference it true

```
falseValue false;

sfConfig extends Assert {
    isTrue falseValue;
    message "false and true are unequal";
}
```

Needless to say, this configuration will not deploy, as the assertion does not hold. An exception will be thrown at deployment time which will prevent the deployment from completing, with the text message supplied as the message of the exception.

Simple use: checking for files

This is how we check that the `tempFile` component generates temporary files:

```
#include "/org/smartfrog/services/filesystem/components.sf"
#include "/org/smartfrog/services/assertions/components.sf"

sfConfig extends Compound {

    temp1 extends TempFile {
        deleteOnExit true;
        prefix "temp1";
        suffix ".txt";
    }

    assert extends Assert {
        fileExists LAZY temp1:filename;
    }
}
```

The temp1 component has a runtime attribute, `filename`, which is valid after the component is started; the assertion check will probe it for existing. This does not test the complete semantics of the component -we do not check the `deleteOnExit` behaviour, or that the prefix and suffix are adhered to. Those are checked in Java code after successfully deploying this configuration.

Advanced use: evaluating a method

Assume there is a component, here `BooleanValues`, that returns Boolean values, with methods all of the signature `boolean method()`. The interface for the component declare the remotable methods.

```
public interface BooleanValues extends Remote {
  public boolean getTrue() throws RemoteException;
  public boolean getFalse() throws RemoteException;
  public boolean getValue() throws RemoteException, SmartFrogResolutionException;
}
```

Implementations of the methods return Boolean values depending upon their state:

```
    public boolean getTrue() throws RemoteException {
        return true;
    }

    public boolean getFalse() throws RemoteException {
        return false;
    }

    public boolean getValue() throws RemoteException, SmartFrogResolutionException {
        boolean b=sfResolve("value",true,false);
        return b;
    }
```

Assertion components can be declared in a deployment descriptor, components that invoke the methods by reference and name.

```
#include "org/smartfrog/services/assertions/components.sf"
#include "org/smartfrog/test/system/assertions/components.sf"

sfConfig extends Compound {

    boolValue extends BooleanValues {
    }

    asserts extends Assert {
        reference boolValue;
        evaluatesTrue "getTrue";
        evaluatesFalse "getFalse";
    }
}
```

The methods are invoked on startup and on every liveness check; here the assertions are always valid, so this deployment will succeed.

Important: testing experiments imply that the methods are not invoked on startup, only on liveness checks. Presumably this means that the reference is not fully up and running,

Exceptions during evaluation

If, when evaluating a method, an exception is thrown, the behaviour varies depending upon the type of the exception.

1. Any `SmartFrogException`, or derivative, is passed on to the caller.
2. Any `RemoteException`, or derivative is passed on to the caller.
3. All other exceptions are wrapped in a `SmartFrogAssertionException`, which is a subclass of `SmartFrogException`, then rethrown.
4. `RuntimeException` instances may or may not get caught. That is a "we don't know, it depends" statement based on whether or not the reflection runtime catches all `Throwable` faults raised and wraps them in `InvocationTargetException` instances, or whether only

As explained in the introduction, the exact handling of `SmartFrogAssertionException` varies depending upon whether this is a liveness or startup check. Generally the most detailed exception that the signature of the calling method permits is thrown, with nesting when this is not possible. The effective result is that invoked methods can throw any type of exception they like, but that nesting may hide the underlying cause.

Advanced Use: evaluating attributes

Effective Use

The assertion component is not intended to replace well-written, domain specific liveness tests, though the evaluation mechanism can be used to provide extra checks if needed. For example, a component that provides extra read-only state information via boolean methods, can have those methods re-evaluated on a regular basis. A fully configurable liveness test is usually much superior.

The role of the assert component is better kept as that of validating the deployment descriptor itself, by probing for attributes and conditions that must be true or false, and providing meaningful text when this is the case.

Another intended use for the component is unit testing -the more of the system tests that can be described in the deployment descriptor, the less java-side coding that is needed to match the descriptor. To facilitate this use, the assert class may need to evolve to enable more effective assertions about the state of deployed components.

Futures

The evolution of this component is determined by what is useful in building and testing deployment descriptors. Better supports for conditions, including the basic equal/unequal tests would be very useful in many situations.