

## Reference constraints verification

Reference constraints verification was developed during Google Summer of Code 2010 and 2011 by Filip Rogaczewski, mentored by Suzette Person and Peter Mehltz.

### Motivation

The motivation for the project comes from the airline industry, which pays a particular attention to the code security. Java language provide only a very limited protection for protecting code from an unauthorized access in sort of all or nothing way. There are only four following protection levels:

- public – if a variable member or method is declared public, it means all other classes, regardless of the package they belong to, can access the member or call the method.
- protected – if a variable member or method is declared protected, it means all classes from the package in which protected member or method is declared, can access the member or call the method. Additionally, protected member may be accessed by a subclass even if the subclass is in a different package.
- default - if a variable member or method is declared protected, it means all classes from the package in which protected member or method is declared, can access the member or call the method.
- private – if a variable member or method is declared private, it means they can't be accessed by code in any class other than the class in which the private member or method is declared

While there are languages like Scala, which provides access modifiers augmented with qualifiers, this syntax is just an illusion of a reference safety. In fact, those modifiers do not give as any control of a reference flow in our system. The following example shows how easy it is to modify a protected reference and break a safety of whole system.

```
public class Temperature {
    private double degrees;
    public Temperature(double d);
    public double getDegrees();
    public void setDegrees(double degrees);
}

public class AirDensity {
    private Temperature airTemperature;
    public AirDensity(Temperature temperature);
    public Temperature getAirTemperature();
}

public class AircraftSensors {
    static AirDensity getAirDensity();
}

public class AircraftDataFacade {
    public AirDensity getAirDensity() {
        AircraftSensors.getAirDensity();
        ad.getAirTemperature().setDegrees(120.0d);
        return ad;
    }
}
```

As we can see, although the field degrees is private, we can change its value. Such action may affect the behavior of all other parts of the system holding a reference to the modified object.

@ConfinedField, @Self, etc... are annotations which provided means to verify leakages of sensitive references from SUT. Each Java construct **T** may specify a set of construct **S** which is permitted to perform certain set of instructions on **T**. The following annotations target at checking if: a method calls, an objects creations, throwing an exception, changing a reference or changing object's value is conducted from a specified scope.

### Syntax

Checkers which are part of jpf-aprop are dedicated to verify the following set of properties. Each of those properties may be defined with annotations and using jpf.properties.

### ConfinedType

Type constraints specify that all constructs of the type are confined. Annotation properties provide way to distinguish between static and instance members of the type. Members of the type may have their own constraint specified, however they are not allowed to widen the confined scope. The syntax of the annotation is:

```
@ConfinedType(instance = <boolean>, type = <boolean>, to = scope)
```

This safety property may be also configured in jpf.properties file, using the following syntax:

```
<fullyQualifiedTypeName> = @ConfinedType(type = <boolean>; instance = <boolean>; to = <scope>)
```

As default, both type and instance parameters are true.

### ConfinedField

Reference constraints specify that the reference to the annotated field is confined to the provided scope. It is possible to specify two parameters for a field constraint. The **value** indicates that it is restricted to modify the referenced object, **reference** parameter indicates that it is not possible to change or acquire a field's reference. The syntax of the annotation is:

```
@ConfinedField(value = <boolean>, reference = <boolean>, to = <CS>)
```

The property's syntax is:

```
<fullyQualifiedFieldName> = @ConfinedField(value = <boolean>; reference = <boolean>; to = <CS>)
```

As default, both value and reference parameters are true.

### Self

Reference constraint specifying that the reference to the annotated field is confined to the object which owns the reference. Like in @ConfinedField, it is possible to narrow the constraint, with **value** parameter allowing to modify only the referenced object and **reference** parameter, protecting the reference from being changed or acquired. The syntax of the annotation is:

```
@Self(value = <boolean>, reference = <boolean>)
```

The property's syntax is:

```
<fullyQualifiedFieldName> = @Self(value = <boolean>; reference = <boolean>)
```

As default, both value and reference parameters are true.

### ConfinedExecutable

Method and constructor constraint specifying that the method or constructor may be invoked only from the provided scope. The syntax of the annotation is:

```
@ConfinedExecutable(<CS>)
```

The property's syntax is:

```
<fullyQualifiedMethodName> = @ ConfinedExecutable (to = <CS>)
```

### ConfinedScope

ConfinedScope is a set of constructs allowed to perform certain instructions on a confined construct. ConfinedScope is express with the following syntax

- qualified names of a type, for instance **java.util.ArrayList**
- types from a package, for instance **java.util.\***
- types extending another type, for instance **? extends java.util.AbstractCollection**
- set of regions with their symbolic name prefixed with **#**.

## Region

Region is an abstract construct, which is a collection of related parts of the program. The constraint specify that the type and its methods belong to region provided as a parameter. Each type may belong to multiple regions. The syntax of the annotation is:

```
@Region(ids)
```

The property's syntax is:

```
<fullyQualifiedTypeName> = @Region(id = <regionID>)
```

## Running checkers

To explore the reference constraints verification run the examples from src/examples. To run the program, execute:

```
bin/jpf listener+=, .aprop.listener.ConfinedChecker ConfinedViolation
```

To let JPF check for violations of the specified properties, the following three checkers have to be configured:

- **ConfinedChecker** which verifies references constraints.
- **RegionChecker** which turns region's infrastructure.
- **DynamicRegionListener** which turns and manages dynamic region's infrastructure.

To configure them, just place the following lines in JPF configuration.

```
listener = .aprop.listener.ConfinedChecker  
listener += .aprop.region.RegionsChecker  
listener += .aprop.region.DynamicRegionListener
```