# Chapter 1

# SCJ Annotations User Manual

**Last edited by Ales Plsek, Date: 2011/06/19 22:38:02 .**

This document provides a simplified description of the metadata annotation system defined for Safety Critical Java. Further, the document describes the use of the **Checker** to verify annotated SCJ applications.

The key concepts of the annotation system and the checker are:

1. **Annotation Restating.** A user class/method that extends/overrides infrastructure or user code must restate any annotation present on the original class/method.
2. **Checker Validation Process.** A checker is provided to verify correctness of annotated SCJ applications. Command line arguments can be passed to checker to modify the verification process:
   (a) `-Alevel=0/1/2` — set the SCJ Level of the user application,
   (b) `-AnoScopeChecks` — disables checking of memory safety annotations.

This document is structured as follows:

1. Section 1.1 — **The Checker.**
   (a) Section 1.1.1 — Download and Installation Instructions.
   (b) Section 1.2 — Verifying the Installation.
   (c) Section 1.1.3 — Running the Checker.
2. Section 1.2 — **Annotations Overview.**
   (a) Section 1.2.1 — `@SCJAllowed` annotation.
   (b) Section 1.2.2 — `@SCJRestricted` annotation.
   (c) Section 1.2.4 — Memory Safety annotations.
3. Section 1.3 — **Annotating SCJ Applications — An Introduction.**
   (a) Section 1.3.1 — Annotating SCJ-given classes.
   (b) Section 1.3.2 — More on the use of `@Scope`.
   (c) Section 1.3.3 — Dealing with method invocation.

   (d) Section 1.3.4 — Using enterPrivateMemory() and executeInArea().
   (e) Section 1.3.5 — Default Annotations.
   (f) Section 1.3.6 — Dynamic Guards.
4. Section 1.4 — **More Resources.**

## 1.1 Checker

### 1.1.1 Download and Installation Instructions

Follow the instructions below to download and install the **Checker**.

1. **Download the Checker.**
   The Checker distribution can be downloaded from the `http://www.ovmj.net/oscj/checker/checker.html`. Execute:

```
$ wget http://scj−jsr302.googlecode.com/files/scjChecker.tar.gz
$ tar −zxvf scjChecker.tar.gz
$ export CHECK_HOME=‘pwd‘/checker/
```

   You can also check-out the latest sources from:

```
$ hg clone https://scj−jsr302.googlecode.com/hg/ scj
$ export CHECK_HOME=‘pwd‘/scj/oSCJ/tools/checker/
```

   The content of the installation directory is listed in Fig 1.1.

2. **Installation Instructions**
   Compiling the SCJ checker currently relies on some bash scripts, which means you probably need Cygwin to compile on Windows.
   (a) Enter the Checker directory:

```
$ cd $CHECK_HOME
```

   (all the paths in the remaining steps are relative to this directory)
   (b) Set the Properties
   Create a file named build.properties in the current directory:

```
$ touch build.properties
```

   Inside this file, add a single line in the format jdk.input.jar=<jdk jar>. The JDK JAR is the path to either rt.jar on Windows/Linux or classes.jar on OS X.
   On OS X, this *might* look like:

```
jdk.input.jar=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/ \
Contents/Classes/classes.jar
```

   On Windows, it *might* look like:

| *Dir/File name* | *Description* |
|---|---|
| doc/ | Documentation. |
| examples/ | Examples. |
| lib/ | **Checker** jar files. |
| localbin/ | Contains the Checker Framework JSR-302 distribution. |
| reports/ | Testsuite Report (generated by the ant tests). |
| spec/ | Annotated Java standard lib (contains e.g. java.lang,java.util). |
| src/ | **Checker** sources. |
| testsuite/ | TestSuite implementation and **Checker** test-cases. |
| build.properties | Ant build properties (created by user during the installation). |
| build.xml | Ant build file. |
| check.sh | Script for running the **Checker**. |
| README | Readme file. |

Figure 1.1: Content of the Distribution Directory.

```
jdk.input.jar=C:\Program Files (x86)\Java\jdk1.6.0_23\jre\lib\rt.jar
```

You may need to use the Cygwin path format on Windows. This would look something like:

```
jdk.input.jar=/cygdrive/c/Program\ Files\ \(x86\)/Java/jdk1.6.0_23/jre/lib/rt.jar
```

(c) Compile the **Checker** by running:

```
$ ant
```

### 1.1.2 Verifying the Checker Installation

The checker is distributed with a complex test-suite that verifies the correctness of the installation process and of the checker on a number of test cases. To run the SCJ checker tests, simply run ant tests. There are a few other test targets, but they do not usually need to be run.

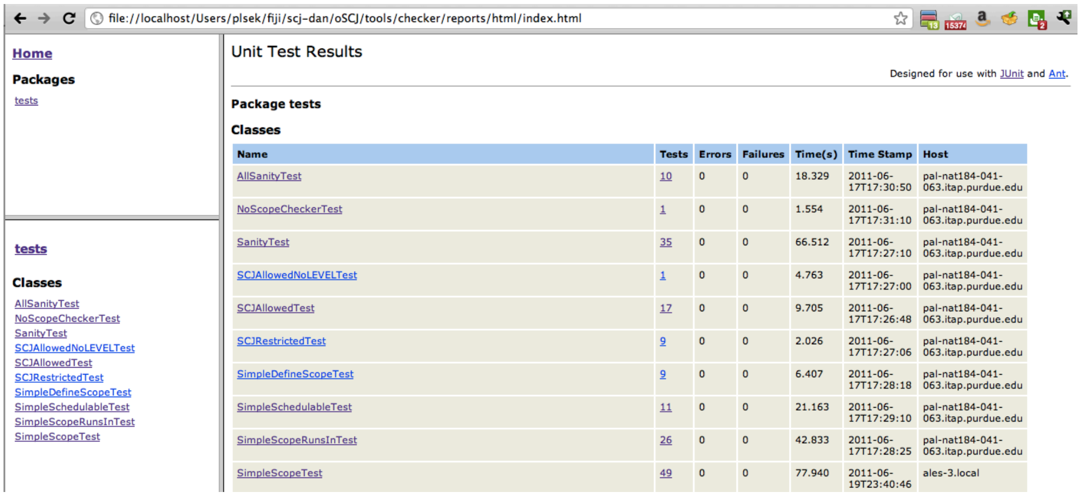
1. To run the tests:

```
$ ant tests
```

2. Then run the following to see the results of the tests:

```
$ open reports/html/index.html
```

This command will open a report of the test-cases, an expected report message should look similarly to the report displayed by Fig. 1.2.


file://localhost/Users/plsek/fiji/scj-dan/oSCJ/tools/checker/reports/html/index.html

Home

Packages

tests

tests

Classes

AllSanityTest
NoScopeCheckerTest
SanityTest
SCJAllowedNoLEVELTest
SCJAllowedTest
SCJRestrictedTest
SimpleDefineScopeTest
SimpleSchedulableTest
SimpleScopeRunsInTest
SimpleScopeTest

Unit Test Results

Designed for use with JUnit and Ant.

Package tests

Classes

| Name | Tests | Errors | Failures | Time(s) | Time Stamp | Host |
|---|---|---|---|---|---|---|
| AllSanityTest | 10 | 0 | 0 | 18.329 | 2011-06-17T17:30:50 | pal-nat184-041-063.itap.purdue.edu |
| NoScopeCheckerTest | 1 | 0 | 0 | 1.554 | 2011-06-17T17:31:10 | pal-nat184-041-063.itap.purdue.edu |
| SanityTest | 35 | 0 | 0 | 66.512 | 2011-06-17T17:27:10 | pal-nat184-041-063.itap.purdue.edu |
| SCJAllowedNoLEVELTest | 1 | 0 | 0 | 4.763 | 2011-06-17T17:27:00 | pal-nat184-041-063.itap.purdue.edu |
| SCJAllowedTest | 17 | 0 | 0 | 9.705 | 2011-06-17T17:26:48 | pal-nat184-041-063.itap.purdue.edu |
| SCJRestrictedTest | 9 | 0 | 0 | 2.026 | 2011-06-17T17:27:06 | pal-nat184-041-063.itap.purdue.edu |
| SimpleDefineScopeTest | 9 | 0 | 0 | 6.407 | 2011-06-17T17:28:18 | pal-nat184-041-063.itap.purdue.edu |
| SimpleSchedulableTest | 11 | 0 | 0 | 21.163 | 2011-06-17T17:29:10 | pal-nat184-041-063.itap.purdue.edu |
| SimpleScopeRunsInTest | 26 | 0 | 0 | 42.833 | 2011-06-17T17:28:25 | pal-nat184-041-063.itap.purdue.edu |
| SimpleScopeTest | 49 | 0 | 0 | 77.940 | 2011-06-19T23:40:46 | ales-3.local |

Figure 1.2: Checker Test-Cases report example.

### 1.1.3 Running the Checker

To check your own program with the SCJ checker, you may use the included check.sh script:

```
$ ./check.sh <program files to check...>
```

or do one of the following:

1. On UNIX:

```
$ $CHECK_HOME/localbin/javac −proc:only −cp $CHECK_HOME/lib/scj.jar: \
    $CHECK_HOME/lib/scj−checker.jar −processor checkers.SCJChecker \
        <∗.java program files to check...>
```

2. On Windows:

```
$ $CHECK_HOME/localbin\javac.bat −proc:only −cp $CHECK_HOME/lib\scj.jar;\
  $CHECK_HOME/lib\scj−checker.jar −processor checkers.SCJChecker \
        <∗.java program files to check...>
```

A typical output of the **Checker** is shown in Fig 1.3.

## 1.2 Annotation System Overview

### 1.2.1 @SCJAllowed Annotation

**Description**: Annotation restricts the visibility of the SCJ-infrastructure code and defines the level on which an SCJ application is executed.

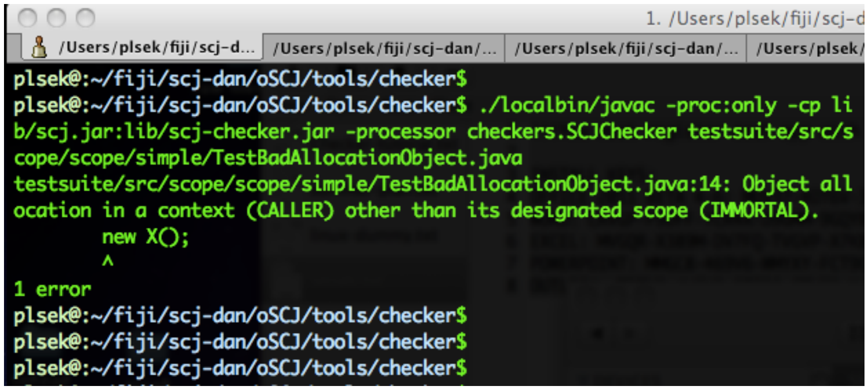



Figure 1.3: A Checker output example.

**Annotation**: @SCJAllowed(value=LEVEL_0,LEVEL_1/LEVEL_2,members=true/false)

**Command-line options**: Use a command-line argument -Alevel to define default SCJ level of user level classes: Example:

```
./check.sh −Alevel=0 ∗.java
```

**SUPPORT methods:** Every method that overrides a @SCJAllowed(SUPPORT) method code must restate this annotation.

### 1.2.2 @SCJRestricted Annotation

**Description:** The annotation restricts the behavior of a body of a method.

**Annotation Arguments:**

- phase: INITIALIZATION, EXECUTION, CLEANUP, ANY(default).
- mayAllocate: true (default), false.
- maySelfSuspend: true (default), false.

**Usage:** The following restrictions may by applied to a method, further, these restrictions should apply transitively also to all the methods invoked by that particular method.

- define a phase when a method can be executed,
- forbid allocation of a given method,
- forbid the method to be able to self-suspend.

### 1.2.3 @SCJAllowed and @SCJRestricted Annotations Example

The example presented in Fig. 1.4 and in Fig. 1.5 shows a simple LEVEL_1 application with one periodic event handler annotated with the @SCJAllowed and @SCJRe-

```
public class Level1Hello implements Safelet {

    @SCJRestricted(INITIALIZATION)
    @SCJAllowed(SUPPORT)
    public MissionSequencer getSequencer() {
        return new OneSequencer(new PriorityParameters(PriorityScheduler
                .instance().getNormPriority()), new StorageParameters(100000L,
                1000, 1000));
    }

    @SCJAllowed(SUPPORT)
    @SCJRestricted(INITIALIZATION)
    public void setUp() {}

    @SCJRestricted(CLEANUP)
    @SCJAllowed(SUPPORT)
    public void tearDown() {}

    static public class OneSequencer extends MissionSequencer {
        @SCJRestricted(INITIALIZATION)
        OneSequencer(PriorityParameters p, StorageParameters s) {
            super(p, s);
        }

        @Override
        @SCJAllowed(SUPPORT)
        protected Mission getNextMission() {
            return new OneMission();
        }
    }

    static public class OneMission extends Mission {

        @Override
        @SCJRestricted(INITIALIZATION)
        @SCJAllowed(SUPPORT)
        public void initialize() {
            PEH peh = new PEH(...).register();
        }

        @Override
        public long missionMemorySize() {
            return 1000L;
        }
    }
}
```

Figure 1.4: @SCJAllowed and @SCJRestricted annotations example — Safelet.

```
public class PEH extends PeriodicEventHandler {

  @SCJRestricted(INITIALIZATION)
  PEH(PriorityParameters p, PeriodicParameters r, StorageParameters s) {
    super(...);
  }

  @Override
  @SCJAllowed(SUPPORT)
  public void handleAsyncEvent() {}

  @Override
  @SCJRestricted(CLEANUP)
  @SCJAllowed(SUPPORT)
  public void cleanUp() {}
  }
}
```

Figure 1.5: @SCJAllowed and @SCJRestricted annotations example — Handler.

stricted annotations. The reader should notice mainly the following :

1. @SCJAllowed(SUPPORT) is restated for all the infrastructure overriden methods,
2. @SCJRestricted(INITIALIZATION/CLEANUP) are similarly restated.
3. Other @SCJAllowed annotations are omitted assuming that the -Alevel=0/1/2 command line argument was passed to the **Checker**.

### 1.2.4 Memory Safety Annotations

The three SCJ annotations for memory safety are summarized in Table 1.1.

**The important restrictions** of the Annotation System are:

1. A class implementing Schedulable can be instantiated only once per a mission.
2. Static fields are @Scope(IMMORTAL) and can be passed only where an IMMORTAL type is expected.
3. Schedulable-s can be instantiated only in the Mission.initialize() method.

**Checker Requirement:** To allow verification of the memory-safety annotations, all the source-code included by the application code must be provided.

| Annotation | Where | Arguments | Description |
|---|---|---|---|
| @DefineScope | Any | *Name* | Define a new scope. |
| @Scope | Class | *Name* | Instances are in named scope. |
| | | **CALLER** | Can be instantiated anywhere. |
| | Field | *Name* | Object allocated in named scope. |
| | | UNKNOWN | Allocated in unknown scope. |
| | | **THIS** | Allocated enclosing class' scope. |
| | Method | *Name* | Returns object in named scoped. |
| | | UNKNOWN | Returns object in unknown scope. |
| | | **CALLER** | Returns object in caller's scope. |
| | | THIS | Returns object in receiver's scope. |
| | Variable | *Name* | Object allocated in named scope. |
| | | UNKNOWN | Object in an unknown scope. |
| | | **CALLER** | Object in caller's scope. |
| | | THIS | Object in receiver's scope. |
| @RunsIn | Method | *Name* | Method runs in named scope. |
| | | CALLER | Runs in caller's scope. |
| | | **THIS** | Runs in receiver's scope. |

Table 1.1: Annotation summary. Default values in bold.

## 1.3 Annotating SCJ Applications — An Introduction

This section describes the process of annotating SCJ applications with the memory safety annotations. The process can be divided into several steps.

### 1.3.1 Annotate SCJ-given classes

In this first step, we annotate Safelet, MissionSequencer-s, their Mission-s, and Schedulable-s. Follow these simple rules:

1. Safelet
   (a) must be @Scope(IMMORTAL).
2. MissionSequencer
   (a) must have @DefineScope(name="A",parent="P"), where "A" is the scope where all the Missions of this sequencer will be allocated, and where "P" is the parent scope where the MissionSequencer is allocated.
   (b) annotate getNextMission with @RunsIn("A").
3. Mission
   (a) annotate with @Scope("A"), where is the scope defined by the MissionSequencer.

4. Schedulable
   (a) annotate @Scope("A"), where "A" is the scope of the Mission where the Schedulable lives.
   (b) annotate with @DefineScope(name="B",parent="A"), where "B" is the scope where the Schedulable is executed.
   (c) annotate the Schedulable 's execution method (e.g. handleAsyncEvent() for event handlers) with @RunsIn("B").

```
@Scope(IMMORTAL)
class Level1Hello implements Safelet {

    @Scope(IMMORTAL)
    @DefineScope(name = "OneMission", parent = IMMORTAL)
    static class OneSequencer extends MissionSequencer {

        @Override
        @RunsIn("OneMission")
        @SCJAllowed(SUPPORT)
        protected Mission getNextMission() {
            return new OneMission();
        }
    }

    @Scope("OneMission")
    static class OneMission extends Mission {
       // ...
    }

    @Scope("OneMission")
    @DefineScope(name = "PEH", parent = "OneMission")
    static class PEH extends PeriodicEventHandler {

        @Override
        @RunsIn("PEH")
        @SCJAllowed(SUPPORT)
        public void handleAsyncEvent() {
        }

        // ...
    }

    //...
}
```

Figure 1.6: A Level1 SCJ application example with memory safety annotations.

We present a simple LEVEL_1 SCJ application annotated with the memory safety annotations in Fig. 1.6. Only the elements that have some memory safety annotations are displayed in the example.

The reader should observe the following:

1. The scopes defined by the @DefineScope annotation form a tree rooted by the IMMORTAL scope. This scope is automatically defined by the checker.
2. The OneSequencer defines a new scope that is common to all the missions instantiated by this MissionSequencer. Further, its getNextMission() explicitly defines @RunsIn to declare that the method is running in this newly defined scope.
3. The PEH class representing a handler defines its own scope, the @RunsIn annotation on the handleAsyncEvent() method corresponds to this scope.

### 1.3.2 More on the @Scope Annotation

Once the SCJ-given classes are annotated, the @Scope annotation can be used to explicitly define allocation scope of user classes. For example, the Fig 1.7 shows a MissionData class that has @Scope("OneMission") annotation. By default, all the methods in this class must run in this scope, e.g. the method set().

```
@Scope("OneMission")
class MissionData {

  class Data value

  void set(int x, int y) {
    this.value = new Data(x,y);
  }

  @RunsIn(CALLER)
  void manipulate () {
    @Scope("OneMission") Data n_value = this.value;
    //...
    }

  @Scope("OneMission")
  @RunsIn(CALLER)
  Data getData() {
    return this.value;
  }
}

class Data {...}
```

Figure 1.7: Explicit @Scope annotation on classes

In contrast, the class Data is defined with no @Scope annotation, which defaults to @Scope(CALLER) meaning that the class can be instantiated anywhere (see Section 1.3.5 for more information about default values).

Furthermore, the @Scope annotation can be used also to explicitly annotate fields and variables. This is demonstrated in the method MissionData.manipulate() that can be called from any scope (at the @RunsIn(CALLER) annotation on the method declaration suggests). Since the method can be called from anywhere, we need to use the @Scope annotation explicitly in the declaration of the n_value variable to preserve the scope information.

Finally, the @Scope annotation can be also used on method declarations, to define the scope of the object returned by the method. This is demonstrated by the method getData(), which is again annotated @RunsIn(CALLER). Since the method can be invoked from any scope, we declare its return type as @Scope("OneMission") to preserve the scope information.

### 1.3.3 Dealing with Method Invocation

Every method declaration must contain an information about the scope in which the particular method is running, this is specified by the @RunsIn annotation. In case no @RunsIn is defined, the default value is assumed — @RunsIn(THIS), which means that the method is running in the same scope as its enclosing class. This was the case of the MissionData.set() method from Fig. 1.7.

```
@Scope("OneMission")
@DefineScope(name = "PEH", parent = "OneMission")
class PEH extends PeriodicEventHandler {

  MissionData data;

  @Override
  @RunsIn("PEH")
  @SCJAllowed(SUPPORT)
  public void handleAsyncEvent() {
    data.set(0,0);  // ERROR
    @Scope("OneMission") Data d = data.getValue();

  }
}
```

Figure 1.8: A method invocation example.

The @RunsIn annotation is then used when verifying each method invocation. In essence, the method invocation rule guarantees that the place of the method invocation and the invoked method run in the same allocation scope.

Looking at the example in Fig. 1.8, the user is not allowed to invoke MissionData.set() method from the PEH.handleAsyncEvent() method since both the methods are defined to run in different allocation scopes. However, the @RunsIn(CALLER) annota-

```
@DefineScope(name="EPM",parent="PEH")
class EnterPM implements Runnable {
    @RunsIn("EPM")
    public void run() {...}
}
```

Figure 1.9: A Runnable class for enterPrivateMemory().

```
@DefineScope(name="EPM",parent="PEH")
class ExecIA implements Runnable {
    @RunsIn("OneMission")
    public void run() {...}
}
```

Figure 1.10: Runnable class for executeInArea().

tion of the MissionData.getValue() method allows the user to invoke the getValue() method. Notice the @Scope annotation on the d variable which preserves the scope information for a newly returned object.

### 1.3.4 Use of enterPrivateMemory() and executeInArea()

The invocation of enterPrivateMemory() and executeInArea() methods must follow certain rules.

**Define the Runnable classes**

For enterPrivateMemory(), define a class implementing Runnable as follows:

1. use @DefineScope— since the enterPrivateMemory() call creates a new scope, we need to define the scope using @DefineScope.
2. use @RunsIn— to specify the scope in which the run() method runs.

Fig. 1.9 shows a class implementing the Runnable interface that can be used for the enterPrivateMemory() method calls.

For executeInArea():

1. same as for enterPrivateMemory(), but we do not need to use @DefineScope as the scope is already defined.
2. use the @RunsIn to define the scope of run().

Fig. 1.10 shows a class implementing the Runnable interface that can be used for the executeInArea() method calls.

**Using an instance of a ManagedMemory/MemoryArea:** ManagedMemory class instances must be annotated to :

1. define where the instance object is allocated (using @Scope), and
2. define which scope it represents (using @DefineScope).

```
@Override
@RunsIn("PEH")
@SCJAllowed(SUPPORT)
public void handleAsyncEvent() {
  @Scope("OneMission")
  @DefineScope(name="PEH",parent="OneMission")
  ManageMemory mem = ManagedMemory.getCurrentManagedMemory();

  EnterPM epm = new EnterPM();
  mem.enterPrivateMemory(1000,epm);

  @Scope(IMMORTAL)
  @DefineScope(name="OneMission",parent=IMMORTAL)
  ManagedMemory mem = (ManagedMemory) ManagedMemory.getMemoryArea(this);
  ExecIA eIA = new ExecIA();
  mem.executeInArea(eIA);
}
```

Figure 1.11: Annotating ManagedMemory object example.

Therefore, each declaration of a ManagedMemory field or variable must have two annotations — @Scope and @DefineScope. The example in Fig. 1.11 shows properly annotated instances of the ManagedMemory class.

### 1.3.5 Default Annotations

To reduce the annotation burden for programmers, annotations that have default values can be omitted from the program source.

Default annotation values and its meaning:

1. **Class:** @Scope(CALLER)— this means that when annotations are omitted classes can be allocated in any context (and thus are not tied to a particular scope).
2. **Local variables and arguments:** @Scope(CALLER).
3. **Fields:** — we assume by default that they infer to the same scope as the object that holds them, i.e. their default is @Scope(THIS).
4. **Instance methods:** @RunsIn(THIS)— this means that the class can be invoked from a scope that is the same as the scope of the receiver.

### 1.3.6 Dynamic Guards

Dynamic guards are our equivalent of dynamic type checks. They are used to recover the static scope information lost when a variable is cast to UNKNOWN, but they are also a way to side step the static annotation checks when these prove too constraining.

A dynamic guard is a conditional statement that tests the value of one of two pre-defined methods, allocatedInSame() or allocatedInParent() or, to test the scopes of a pair of references. If the test succeeds, the check assumes that the relationship between the variables holds. The parameters to a dynamic guard are local variables which must be final to prevent an assignment violating the assumption.

The following example (Fig. 1.12) shows how to store an unknown reference into a list allocated in the receiver's scope. Without the guard, the assignment statement

```
void method(@Scope(UNKNOWN) final List unk, final List cur) {
    if (ManagedMemory.allocatedInSame(unk, cur)) {
      cur.tail = unk;
    }
  }
```

Figure 1.12: Dynamic Guard Example.

would not be valid, since the relation between the objects' allocation contexts can not be validated statically.

## 1.4 More Resources

Further resources explaining in details the annotation system and the **Checker** are:

1. **Specification** – SCJ Specification, Chapter 9 contains a full specification of the annotation system and examples.
2. **A Static Memory Safety Annotation System for Safety Critical Java —** a research paper submitted to RTSS'11, available at `http://sss.cs.purdue.edu/projects/rtss11/RTSS11-extended-version.pdf`. Contains additional explanation of the annotation system, several case studies, one of them describes in the detail the annotations used.
3. **Additional Examples:**
   (a) Case Studies — the case studies contain more than 30KLOC of annotated code and are distributed with the checker, in $CHECKER_HOME/examples/.
   (b) Test-Cases — more than 170 test cases distributed with the checker, in $CHECKER_HOME/testsuite/src .
4. **The Checker distribution**, source-code, and its documentation can be found at:
   (a) oSCJ Project Web-Page: `http://www.ovmj.net/oscj/`.
   (b) Checker Repository: `http://code.google.com/p/scj-jsr302/`.
5. **Contact**
   (a) Authors: Daniel Tang, Ales Plsek, Jan Vitek.

(b) Checker Mailing List: `http://groups.google.com/group/SCJava?pli=1`.
(c) Purdue SCJ mailing list : `scj@cs.purdue.edu`.

# Bibliography