

[TracNav](#)

- [JPFWiki](#) - Welcome Page

[Introduction...](#)

[Installing JPF...](#)

[User Guide...](#)

[Developer Guide...](#)

[Projects](#)

- [jpf-core](#)
- [jpf-actor](#)
- [jpf-awt](#)
- [jpf-awt-shell](#)
- [jpf-concurrent](#)
- [jpf-cv](#)
- [jpf-delayed](#)
- [jpf-guided-test](#)
- [jpf-mango](#)
- [jpf-racefinder](#)
- [jpf-rtembed](#)
- [jpf-statechart](#)
- [net-iocache](#)
- [jpf-aprop](#)
- [jpf-numeric](#)
- [jpf-symbc](#)
- [jpf-concolic](#)
- [jpf-symbc-load?](#)
- [jpf-extended-test-gen](#)
- [jpf-parallel-spf?](#)
- [eclipse-jpf](#)
- [netbeans-jpf](#)
- [jpf-inspector](#)
- [jpf-shell](#)
- [jpf-template](#)
- [jpf-trace-server](#)
- [standard NB example](#)
- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

[About...](#)

- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

jpf-aprop

Introduction

The jpf-aprop project contains [Java annotation](#) based program property specifications, together with corresponding [listeners](#) to check them.

The general idea is to use annotations so that SUT behavior outside JPF execution is not modified, and the SUT can be run normally outside JPF. Ideally, such annotations are useful program documentation, and can be processed by a variety of verification tools (e.g. static analysis tools), like the `@Nonnull` example below, which is part of the [JSR-305 proposal](#) and can also be checked by [FindBugs™](#) or the [Checker Framework](#) which implements the upcoming [JSR-308](#).

```
class X {...
  @Nonnull Y computeY() { // this is never allowed to return null, so callers don't have to check
    ...
  }
}
```

The scheme is always the same - each annotation is paired with a corresponding checker (usually a listener, but there are also standalone tools). The annotations are also made available in a `jpf-aprop-annotations.jar` (which exclusively contains annotation types) so that SUTs can be compiled without having to include any executable code from JPF projects. This is especially important if you have to ensure that SUT behavior outside JPF is not modified by using such annotations.

Once your SUT has been compiled with property annotations, all you have to do to execute the corresponding checks is to specify the related listeners:

```
> bin/jpf +listener=gov.nasa.jpf.aprop.listener.NonnullChecker ...
```

Repository

The Mercurial repository can be found on <http://babelfish.arc.nasa.gov/hg/jpf/jpf-aprop>

Annotations and Listeners

Currently, the following annotations and annotation groups are supported:

- [@Nonnull](#) - check method return values and field assignments for null values
- [@Const](#) - check for object modifications within method scope
- [@SandBox](#) - check for modification of external fields from sandbox objects
- [@GuardedBy](#) - lock policy specifications for fields
- [@NonShared?](#) - check for concurrent use of non-threadsafe objects
- [@Requires](#), [@Ensures](#) and [@Invariant](#) - pre-/post-conditions and invariants
- [@Sequence](#), [@SequenceEvent](#), [@SequenceMethod](#), [@SequenceObject?](#) - automatic UML sequence diagram creation
- [@Test?](#) - in-source method test specifications
- [@Confined](#) - check the safety of sensitive references, values and methods
- [@Region?](#) - group classes into a virtual hierarchy

This list is supposed to grow, and some of the annotations use embedded languages for expression arguments (e.g. `@Test`, `@Requires`, `@Ensures` and `@Invariant`), i.e. represent whole categories of annotations. Apart from the more stable ones, expect to see experimental annotations in the source tree.

JPF Options

Property annotations require JPF to find the related listeners, which can be done in two different ways: (a) with explicit `listener` specification, or (b) by configuring "autolisteners".

(1) Explicit listener

This is the preferred method if you just want to check one specific property annotation, e.g. for experimental annotations or checks that do not coexist with others. This method just uses the plain `listener` property, which can be set from the commandline, a project property (`jpf.properties`) or an application property (`*.jpf`) file. Please see the [JPF configuration](#) page for details.

```
> bin/jpf +listener=.aprop.listener.NonNullChecker ...
```

(2) Autolister

Use this method if you want to check for a number of annotations simultaneously, and only load the listeners on demand, i.e. if your execution contains one of the listed annotations. Autolisters are specified by including the annotation type in the `listener.autoload` list, and providing corresponding annotation-type/listener class pairs, as in:

```
listener.autoload=\
  javax.annotation.Nullable,\
  ...
listener.javax.annotation.Nullable=.aprop.listener.NonNullChecker
...
```

Autolister specifications are typically kept in SUT project property files (`jpj.properties` within the SUT root directory).

Please Note:

many of the annotation checkers use the JPF attribute system, which does not yet support multiple attributes per entity. For now, only use autolisters if you know there are not several different attribute types used at the same time

Purpose and categories of property annotations

Property annotations should not be band aids for wished-for but missing language features - if there is a static, type-safe and efficient way to achieve what the property is asking for, a suitable design is a better solution than to depend on additional tools. Ideally, property annotations should therefore be mostly self-explaining type qualifiers that are orthogonal to the existing language.

Looking at proposed annotations, the emerging functional categories seem to be

- null-ness (e.g. `@NonNull`)
- primitive subtypes (e.g. `@Negative`)
- thread safety (`@ThreadSafe`, `@Immutable`, `@GuardedBy` etc.)
- method invocation context (e.g. `@CheckReturnValue`)
- method execution side effects (object mutability like `@Const`, resource management like `@WillClose`)

The majority of existing work uses static analysis for property verification, either as compiler extensions (`javac` annotation processors) or standalone tools (e.g. [FindBugs™](#)). Such tools are relatively easy to use and work on non-executable, incomplete artifacts, i.e. can be applied very early in the development process. JPF on the other hand is a heavy weight execution environment, which requires suitable test cases and configuration effort. Why do we use JPF for property annotations then?

There are mainly three areas where the known, concrete execution context within JPF is helpful

- property locality
- value based properties
- instance based properties

(1) Property locality

Static analyzers work well on properties that can be checked in a narrow scope, e.g. within a method body. A `@Const` annotation might serve as an example

```
class Z {..
  int d;
  void baz () {}
}

class X extends Z {
  int d; ..
  @Const void foo(Y y) {
    d = 1;           // (1) violation easy to detect
  }
}
```

```

    y.bar(this); // (2) not so easy
} ..
void baz () {
    d = 0;
}
}

class Y { ..
    void bar(Z z) { ..
        z.baz();
    }
}

```

Defect (1) is easy to spot for a static analyzer, it only requires simple intraprocedural analysis. Defect (2) is much more expensive since it involves different methods and even compile units. In fact, it is so expensive that static analysis resorts to some three way logic (e.g. const/non-const/unknown-const) in order to avoid false positives or missed errors, which means we need a lot more annotations. C++ programmers know this "ripple effect" (e.g. when using the C++ `const`). JPF only checks if the `const` property is violated during a concrete execution of `x.foo()`, and therefore can precisely detect violations without false positives. Of course all that JPF tells us is if the property holds in an executed test case (leave alone JPF's symbolic execution mode), but introducing the property checks for these test cases is less invasive - and hence expensive - than having to annotate everything upfront that can be executed from within the checked context. JPF allows incremental introduction of annotation properties.

Another locality aspect is the outbound semantic scope of properties. Assume a property annotation like

```

class X {
    void foo (@NonNull Y y) {
        ..
    }
}

```

The essence of this property is more about caller responsibilities, and less about usage of `y` within the scope of the `foo()` body - it is a classical precondition that needs to be checked and reported for each calling context of `foo()`. Again, this is much easier to achieve by JPF (in a concrete execution) than it is for a static analyzer (for all possible calls).

(2) value based property expressions

The previous precondition example can also be thought of as a value based expression `y != null`. It is obvious that such expressions can quickly become so complex that their evaluation is out of reach for static analysis and does require execution of the program:

```

#!java
class X {
    double d;
    ...
    @Requires("a > b && d >= 0")
    @Ensures("d > old(d)")
    void foo (int a, int b) {
        ..
    }
}

```

While preconditions, postconditions and invariants have been deliberately left out in the Java language specification, `assert` based expressions as an alternative are too tedious and error prone in the context of polymorphism and class hierarchies (see [Design-by-Contract™](#)), and simple type qualifiers like `@NonNull` seem to be too restricted from a functional perspective.

With JPF, we can use annotations that represent fully inheritance-aware contracts that use embedded languages for value based expressions. Moreover, evaluation of such expressions can be done strictly in the execution environment, which can guarantee absence of side effects for normal execution - a problem that has long hampered more complex use of assertions.

(3) instance based properties

This can be thought of as a special case of value based properties, where we care about object identities instead of numeric or lexical values. This seems to be especially important for more complex temporal properties like

```
@CallSequence("open {^open} close")
class CheckedFile {
    void open() {...}
    void close() {...}
    byte read() {...}
    ..
}
```

This is an example of a protocol specification which is instance aware - we just care for the call sequences with respect to the same object:

```
..
CheckedFile a = new CheckedFile(..);
CheckedFile b = new CheckedFile(..);
a.open();
b.open(); // fine, different object
a.close();
b.open(); // violation of @CallSequence for object 'b'
..
```

Again, this is easy to implement in JPF, e.g. by using the [JPF attribute system](#) from a dedicated checker listener, but would be hard to achieve with a static analyzer.

The bottom line is that JPF is a good implementation platform for properties that are too expensive for static analysis, and can also allow more gradual introduction of properties into SUTs. The downside is that concrete JPF execution needs suitable test cases. We therefore still see static analyzers and simple type qualifier annotations as a scalable and easy to use first line of defense.

Related Work

There are two JSR efforts that are related to this work:

(1) [JSR-305 "Annotations for Software Defect Detection"](#) Thematically, this effort is the best fit for `jpf-aprop`, since it is focused on the semantics of annotations that can be used across tools to detect general defects. As of this writing (01/2010) JSR-305 seems to have become somewhat sidelined by other efforts (like JSR-308), and the proposed `javax.annotation` types are not yet part of the OpenJdk (jdk7) code base. However, distributions of the [FindBugs™](#) static analyzer do contain such types in a separate `jsr305.jar`, and include code to check for related properties.

(2) [JSR-308 "Type Annotations"](#) This effort is aimed at allowing Java annotations for all type specifications, not just for classes, fields, methods and method arguments (as in Java 6). The specification is not focused on particular annotations or uses (like `@NonNull`), but seems to emerge as the primary testbed and integration platform for new annotation proposals through the [Checker Framework](#), which is the JSR-308 reference implementation. This framework includes not only verification related annotation types, but also corresponding `javac` compiler plugins to check the respective properties at compile time. These checkers are implemented as standard [javac annotation processors](#), which makes them available in the context of a production compiler. While this does not introduce new tools, it effectively is a modification of the bytecode compiler - the most essential component of the tool chain. JSR-308 is slated to become part of JDK 7, but it remains to be seen how much of the Checkers framework will be incorporated.

The associated tools are static analyzers, but the related annotations are general enough to make them useful in a runtime tool like JPF, provided these annotations have a `RUNTIME` retention policy (as opposed to `CLASS` or `SOURCE`, which would be sufficient for a static analyzer).