SRC Technical Note

2000-004

October 12, 2000

ESC/Java Quick Reference

Silvija Seres, June 1999

Revised by K. Rustan M. Leino and James B. Saxe, October 2000

Compaq Computer Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://research.compaq.com/SRC/

Abstract

This document is intended to be a non-detailed, trimmed-down version of the ESC/Java User's Manual, for people who would like to get an overview of the annotation language supported by the Compaq Extended Static Checker for Java (ESC/Java) without getting immersed in all its technical intricacies.

For more detailed information, please refer to the ESC/Java User's Manual. For information about the invocation of the ESC/Java checker, please see the escjava(1) man page included with the ESC/Java download available from http://research.compaq.com/SRC/esc/.

---

0. What does ESC/Java do anyway?

Type annotation and static type checking of programs have proved to be one of biggest engineering successes of computer science. Typing provides a coarse semantics for programs, since it pays no attention to the semantics of any language constructs that are not related to types. Nevertheless, the automatic type checking of programs weeds out already at compile time many of the most common programming errors, thus making them less costly for the developers. Also, typing forces a certain discipline upon the programmer, which results in better programs.

The Compaq Extended Static Checker for Java (ESC/Java) pushes the idea behind type checkers a few steps further. The class of errors that it looks for is much larger and more varied: it addresses, among others, the potential run-time errors that arise from illegal array operations, null-pointer dereferencing, deadlocks and race conditions of threads. Given a Java program, it automatically infers and checks a set of *verification conditions* that correspond to the described classes of errors. It also allows the programmer to record design decisions, and to influence the choice of verification conditions by annotating the program with a set of *pragmas*. These can be used to specify the pre- and postconditions of routines, properties of abstract data types, invariants of loops, and much more.

The errors that ESC/Java looks for are chosen on a pragmatic basis; they are the errors that, according to the engineering experience, occur often and are relatively cheap to find, but the ESC/Java system is flexible and can be extended to allow for checking of other types of errors. Currently, ESC/Java checks almost the entire Java 1.2 language, including all of Java 1.0.

In terms of program verification, ESC/Java is *unsound*, because it can miss real programming errors (from the targeted class), and it is *incomplete*, because it can give some spurious warnings. Some degree of inaccuracy is inevitable in a tool such as ESC/Java due to theoretical limits of decidability. Additionally, the design of ESC/Java intentionally sacrifices some accuracy in trade for efficiency of the tool. The user has some control over ESC/Java's unsoundness and incompleteness thanks to pragmas. These pragmas not only enable modular program checking, but are also a convenient formalism for recording programmers' design decisions and program specifications.

The remainder of this page gives a rough description of the kinds of pragmas available in the current ESC/Java, the *specification expressions* that can occur in those pragmas, and the kinds of warnings ESC/Java reports..

1. There are four syntactic categories of ESC/Java pragmas:

A pragma (annotation) is enlosed in a Java comment whose first character is an `@`. For example, `/*@ non_null */` is an ESC/Java pragma.

- **lexical** pragmas may occur in the same places as Java comments,
- **statement** pragmas may occur in the same places as Java statements,
- **declaration** pragmas may occur in the same places as Java declarations of class and interface members,
- **modifier** pragmas may occur in certain places within Java declarations of variables or routines.

All pragmas enclosed in a single Java comment must be of the same syntactic category.

2. The list of ESC/Java pragmas with their (syntactic, semantic) contexts:

For pragmas terminated by semicolon, the semicolon is optional if there are no further pragmas enclosed in the same comment.

- **`nowarn L;`** (lexical, general):
  `L` denotes a possibly empty comma-separated list of warning types; ESC/Java will suppress any warning messages of the types in `L` (or of all types, if `L` is empty) at the line where the pragma appears.
- **`assume E;`** (statement, general):
  `E` denotes a boolean specification expression; ESC/Java will assume that `E` is true whenever control reaches the pragma and ignores the remainder of all execution paths in which `E` is false.
- **`assert E;`** (statement, general):
  `E` denotes a boolean specification expression; ESC/Java will issue a warning if it cannot establish that `E` is true whenever control reaches the pragma.
- **`unreachable;`** (statement,general): semantically equivalent to `assert false;`
- **`requires E;`** (modifier, non-overriding routine):
  `E` denotes a boolean specification expression that is a precondition of the routine the pragma modifies; ESC/Java will assume that `E` holds initially when checking the implementation of the routine, and will issue a warning if it cannot establish that `E` holds at a call site.
- **`modifies S;`** (modifier, non-overriding routine):
  `S` denotes a nonempty comma-separated list of modification targets; ESC/Java will assume that calls to the routine modify only the modification targets in `S` and freshly allocated state components, but will not check the routine implementation correspondingly (that is, ESC/Java does not warn about implementations that modify more targets than `S` allows).
- **`ensures E;`** (modifier, non-overriding routine):
  `E` denotes a boolean specification expression that is a normal (i.e. non-exceptional) postcondition of the routine the pragma modifies; ESC/Java will assume that `E` holds just after each call site the routine, and will issue a warning if it cannot prove from the routine implementation that `E` holds whenever the routine terminates normally.
- **`exsures (T t) E;`** (modifier, non-overriding routine):
  `T` is a subtype of `java.lang.Throwable`, `t` is an (optional) identifier, and `E` denotes a boolean specification expression that is an exceptional postcondition of the routine the pragma modifies; ESC/Java will assume that `E` holds whenever the a call to the routine completes abruptly by throwing an exception `t` whose type is a subtype of `T`, and will issue a warning if it cannot prove from the routine implementation that `E` holds whenever the routine terminates completes abruptly by throwing an exception `t` whose type is a subtype of `T`.
- **`also_ensures E;`** (modifier, overriding routine):
  This pragma may modify only a method declaration that overrides another method declaration; otherwise,

it has the same meaning as `ensures E;`

- **`also_exsures (T t) E;`** (modifier, overriding routine):
  This pragma may modify only method declaration that overrides another method declaration; otherwise, it has the same meaning as `exsures (T t) E;`
- **`also_requires E;`** (modifier, overriding routine):
  This pragma may modify only a method declaration that occurs in a class declaration, overrides a method of a superinterface, and does not override a method of a superclass; otherwise, it has the same meaning as `requires E;`
- **`also_modifies S;`** (modifier, overriding routine):
  This pragma may modify only a method declaration that overrides another method declaration; otherwise, it has the same meaning as `modifies S;`
- **`non_null`** (modifier, data invariant):
  Modifies the declaration of a variable of a reference type, where the variable may be a static field, instance variable, local variable, or parameter; ESC/Java will check at each assignment to the variable that the value assigned is not null, and assume at each use that the value is not null.
- **`invariant E;`** (declaration, data invariant):
  `E` denotes a boolean specification expression that is an object invariant of the class within whose declaration the pragma occurs. If `E` does not mention `this`, the invariant is called a *static invariant*, and is assumed on entry to implementations, checked at call sites, assumed upon call returns, and checked on exit from implementations. If `E` mentions `this`, the invariant is called an *instance invariant*. An instance invariant is assumed to hold for all objects of the class on entry to an implementation and is checked to hold for all objects of the class on exit from an implementation. At a call site, an instance invariant is checked only for those objects passed in the parameters of the call and in static fields. A call is assumed not to falsify the instance invariant for any object.
- **`axiom E;`** (declaration, data invariant):
  ESC/Java assumes that `E` is true at the start of every routine body
- **`loop_invariant E;`** (statement, data invariant):
  This pragma may appear only just before a Java `for`, `while`, or `do` statement. ESC/Java will check that `E` holds at the start of each iteration of the loop.
- **`spec_public`** (modifier, variable referencing):
  This pragma may modify only non-public field declarations, and it will cause the fields in the declaration to be as accessible in pragmas as they would have been if the declaration had been public.
- **`readable_if E;`** (modifier, variable referencing):
  This pragma may modify only the declaration of a field or a local variable; `E` denotes a boolean specification expression that has to be true at any read access of the fields or variable.
- **`uninitialized`** (modifier, variable referencing):
  This pragma may modify only a local variable declaration that has an initializer; ESC/Java will check that no execution path accesses the variable without first performing an assignment (other than the initializer) to the variable.
- **`ghost M S v;`** (declaration, ghost variables):
  `S` is a specification type, `v` is an identifier, and `M` is a sequence of modifiers including `public`; this pragma is like an ordinary Java variable declaration `M S v;` except that it makes the declaration visible only to ESC/Java, and not to the compiler; such variables are called *ghost variables*.
- **`set D = E;`** (statement, ghost variables):
  `D` refers to a ghost field of some object or class and `E`is a specification expression containing no quantifiers or labels; this pragma has the analagous meaning to the Java assignment statement `D = E;`
- **`monitored_by SL;`** (modifier, synchronization):
  This pragma can be applied only to fields. The modified field is a shared variable monitored by the locks

in `SL`, which is a nonempty, comma-separated list of specification expressions. ESC/Java checks that the field is never read except by a thread holding at least one non-null lock in `SL` and that the field is never written except by a thread holding all non-null locks in `SL`, of which there must be at least one.

- **`monitored`** (modifier, synchronization):
  This pragma may modify only an instance variable declaration, and is the same as `monitored_by this;`

3. The ESC/Java specification expressions:

A specification type is either a Java type or one of the special types `\TYPE` or `\LockSet` (or an array of special types, for example `\TYPE[][]`). The specification type `\LockSet` cannot be mentioned explicitly in annotations.

Specification expressions must be free of subexpressions that may potentially have side effects, so they may not contain any assignment (`=`, `+=`, etc.), pre/post-increment/decrement (`++` or `--`), array or object creation (`new`), or method invocation (even for methods that have no side-effects).

The additional constructs that are allowed in specification expressions beyond those allowed in Java expressions are:

- **`\type(T)`**: `\TYPE`
  denotes the specification type `T`.
- **`\typeof(E)`**: `\TYPE`
  denotes the dynamic type of the value of specification expression `E`, where `E` is of a reference type.
- **`\elemtype(E)`**: `\TYPE`
  denotes the specification type `T` if `E` denotes an array type `T[]`, unspecified otherwise.
- **`S <: T`**: `boolean`
  denotes that `S` is a subtype of `T`, where `S` and `T` are specification expressions of type `\TYPE`.
- **`\lockset`**: `\LockSet`
  denotes the set of locks held by the current thread.
- **`S[L]`**: `boolean`
  denotes that `L` is a member of `S`, where `S` is a specification expression of type `\LockSet` and `L` is a specification expression of a reference type.
- **`E < F`**: `boolean`
  denotes that object `E` precedes object `F` in the locking order.
- **`E <= F`**: `boolean`
  denotes that object `E` precedes object `F` in the locking order or `E == F`.
- **`\max(S)`**: `Object`
  denotes the maximum element of `S` in the locking order, where `S` is a specification expression of type `\LockSet`.
- **`E ==> F`**: `boolean`
  denotes the condition that `E` implies `F`, where `E` and `F` are specification expression of type `boolean`.
- **`(\forall T V; E)`**: `boolean`
  denotes that `E` is true for all substitutions of values of type `T` for the bound variables `V`, where `T` is a specification type, `V` is a nonempty comma-separated list of identifiers, and `E` is a specification expression of type `boolean`.
- **`(\exists T V; E)`**: `boolean`
  denotes that `E` is true for some substitution of values of type `T` for the bound variables `V`, where `E`, `T` and `V` are as above.
- **`\nonnullelements(A)`**: `boolean`

denotes that `A` and all its elements are non-null, where `A` is a specification expression of a reference array type.

- **`\fresh(E)`**`: boolean`
  used in postconditions, denotes that `E` is non-null and was not allocated in the pre-state of the routine call, where `E` is a specification expression of a reference type.
- **`\result`**
  is a specification expression whose type is the return type of the non-void method in whose normal postcondition or modification target it appears, denoting the value returned by the method.
- **`\old(E)`**
  is a specification expression of the same type as the specification expression `E` and is used in a postcondition to denote the same thing as `E` except that (1) any occurrence in `E` of a target field of the routine is interpreted according to the pre-state value of the field, and (2) if any modification target of the routine has the form `X[i]` or `X[*]`, then all array accesses within `E` are interpreted according to the pre-state contents of arrays.
- **`E.owner`**`: Object`
  is a specification expression of type object, denoting the ``owner'' of object `E`. The standard specification library shipped with ESC/Java declares `owner` as a ghost field of `java.lang.Object`. The pragma `/*@ invariant f.owner = this; */` in the declaration of a type `T` is the conventional way to specify that the objects of type `T` do not share their `f` fields. All constructors have the implicit postcondition `this.owner != null`.

4. The ESC/Java modification targets (or specification designators):

- **a simple name** denoting a non-final field,
- **a field access** `O.f`, where `O` is a specification expression of a reference type `T` and `f` denotes one of the fields (possibly a ghost field) of `T`,
- an **array access** of the form `A[I]`, where `A` is a specification expression of an array type, and `I` is a specification expression of an integral type other than `long`, or
- an **array range** of the form `A[*]`, where `A` is a specification expression of an array type.

5. The ESC/Java warning types:

ESC/Java issues warnings for conditions that it regards as run-time errors, and that, so far is it can tell, might actually occur at run-time.

- **`ArrayStore`** warns that the control may reach an assignment `A[I] = E` when the value of `E` is not assignment compatible with the actual element type of `A`.
- **`Assert`** warns that control may reach a pragma `assert E` when the value of `E` is false.
- **`Cast`** warns that control may reach a cast `(T)E` when the value of `E` cannot be cast to the type `E`.
- **`Deadlock`** warns that control may reach a `synchronized` statement that would acquire a lock in violation of the locking order, or that the a `synchronized` method may start by acquiring a lock in violation of the locking order.
- **`Exception`** warns that a routine may terminate abruptly by throwing an exception that is not an instance of any type listed explicitly in the routine's throws clause.
- **`IndexNegative`** warns that control may reach an array access `A[I]` when the value of the index `I` is negative.
- **`IndexTooBig`** warns that control may reach an array access `A[I]` when `A.length <= I`.
- **`Invariant`** warns that some object invariant may not hold when control reaches a routine call, or that

some object invariant may not hold on exit from the current body.

- **`LoopInv`** warns that some loop invariant may not hold when it is supposed to.
- **`OwnerNull`** warns that a constructor may violate the implicit postcondition `this.owner != null`.
- **`NegSize`** warns of a possible attempt to allocate an array of negative length.
- **`NonNull`** warns of a possible attempt to assign the value null to a variable whose declaration is modified by a `non_null` pragma, or to call a routine with an actual parameter value of null when the declaration of the corresponding formal parameter is modified by (or inherits) a `non_null` pragma.
- **`NonNullInit`** warns that a constructor may fail to establish a non-null value for an instance field of the constructed object when the declaration of that instance field is modified by a `non_null` pragma.
- **`Null`** warns of a possible attempt to dereference null, for example, by field access `O.f`, an array access `O[i]`, a method call `O.m(...)`, a synchronized statement `synchronized (O) ...`, or a throw statement `throw O`, where `O` evaluates to `null`.
- **`Post`** warns that a routine body may fail to establish some normal postcondition (on terminating normally) or some exceptional postcondition (when terminating by throwing an exception of a relevant type).
- **`Pre`** warns that control may reach a routine call when some precondition of the routine does not hold.
- **`Race`** warns of a possible attempt to access a monitored field while not holding the requisite lock(s).
- **`Reachable`** warns that control may reach an `unreachable` pragma.
- **`Uninit`** warns that control may reach a read access to a local variable before execution of any assignment to the variable other than an initializer in a declaration modified by an `uninitialized` pragma.
- **`Unreadable`** warns that control may reach a read access of a field or variable `x` when the expression in a `readable_if` pragma modifying `x`'s declaration is false.
- **`ZeroDiv`** warns of a possible attempt to apply the integer division (`/`) or remainder (`%`) operator with zero as the divisor.
