# Verification Conditions Generated by *Escher Verification Studio*

## 1 Revision history

| Version | Date | Author | Description |
|---|---|---|---|
| 1.1 | 2002-05-02 | NE | Initial version |
| 1.2 | 2002-06-26 | NE | Updated to latest “that” and “any” obligations |
| 1.3 | 2002-08-09 | DC | Added introduction; made minor corrections; reformatted |
| 1.4 | 2003-12-10 | DC | Updated to version 2.10. Ordered the obligations alphabetically by name. |
| 1.5 | 2004-11-05 | DC | Updated for version 3.0 |
| 1.6 | 2010-03-03 | DC | Updated for version 4.02 |
| 1.7 | 2010-12-17 | DC | Updated for version 5 changes and Escher C Verifier |
| 1.8 | 2011-05-10 | DC | Further update for Escher C Verifier |
| 1.9 | 2011-10-03 | DC | Updated for Escher Verification Studio 5.0 |
| 1.10 | 2012-12-13 | DC | Reviewed for Escher Verification Studio 6.0 |

## 2 Introduction

This document lists all the verification conditions generated by *Escher Verification Studio* version 5.

In the following, each type of verification condition is introduced by name. The names of verification conditions in this document correspond to the names that *Escher Verification Studio* uses in its informational output and in the proof and unproven output files. For each verification condition, we give a description, a primary location (i.e. source file/row/column) and sometimes also a secondary location. The primary location is the location given at the start of the warning message. The secondary location is the one that appears in brackets, typically preceded by the words “defined at”.

## 3 List of verification conditions

### *3.1 All qualifying elements in operand of 'that' are equal*

Generated for every **that** expression to ensure that all elements in the collection that satisfy the predicate are equal. Primary location is the location of the **that** expression.

### *3.2 Arithmetic result of operator […] is within limit of type […]*

Used by *eCv* only. Generated for arithmetic operators whose result may overflow.

## 3.3 Assertion satisfies inherited assertion

Generated wherever: a method declared with **define** or **redefine** also declares an assertion, does not use the '**…**' form of assertion, and the original method was declared with an assertion. In this case the new assertion must imply the original assertion (i.e. be stronger). Primary location is that of the new assertion; secondary location is that of the original assertion.

## 3.4 Assertion valid

Generated for every **assert** statement, whether within an expression, a postcondition or a statement list (but not a post-assertion). The location is that of the assertion.

## 3.5 At least one guard is true

Used by *Perfect Developer* only. Generated for every conditional expression, conditional postcondition and **if** statement with no empty guard. The location is that of the expression or statement.

## 3.6 Cannot break constraint via selector

Used by *Perfect Developer* only. Generated for every selector that may return a variable of a constrained type, or a sub-component of a variable of constrained type, to check that all values of the type of the selector satisfy the constraint. The primary location is that of the selector; the secondary location is that of the declaration of the constraint.

## 3.7 Class invariant satisfied

Generated after the postcondition of every modifying member schema and constructor, and when **self after …** has been satisfied, for each class invariant declared in the enclosing class. Primary location is that of the final part of the postcondition; secondary location is that of the class invariant.

## 3.8 Data currently holds union member [name]

Used by *eCv* only. Generated whenever a member field of a value of union type is retrieved.

## 3.9 Declared number of array elements either matches number in initial value or is greater and element type can be default-initialized

Used by *eCv* only. Generated for any array declaration that has both a declared number of elements and an initializer, if the element type does not contain any fields that definitely cannot be default-initialized (e.g. non-nullable pointer fields). Verification conditions of this type are not generated when the initializer obviously has a suitable number of elements.

## 3.10 Declared number of array elements matches number in initial value (element type cannot be default-initialized)

Used by *eCv* only. Generated for any array declaration that has both a declared number of elements and an initializer, if the element type contains one or more fields that definitely cannot be default-initialized (e.g. non-nullable pointer fields). Verification conditions of this type are not generated when the initializer obviously has the correct number of elements.

## *3.11 Expressions modified by schema are independent*

Used by *Perfect Developer* only. Generated for each call to a schema that modifies more than one object, to check that these objects are independent of each other. The location is that of the schema call.

## *3.12 Guarded variable [name] is initialised, or its 'when' guard has not become true*

Used by *Perfect Developer* only. Generated every time a guarded data member of a class is accessed. Primary location is that of the call; secondary location is that of the guard condition.

## *3.13 History invariant satisfied*

Used by *Perfect Developer* only. Generated after the postcondition of every modifying member schema, for each history invariant declared in the enclosing class or inherited from a parent class, excluding those history invariants that declare the schema exempt. *[Added at version 4.02]*

## *3.14 Implementation has not changed abstract data*

Used by *Perfect Developer* only. Generated for every implementation of a method whose specification cannot modify **self** (e.g. a function), but whose implementation modifies internal data. The verification condition checks that the values of the abstract data are unchanged. The location is that of the end of the implementation.

## *3.15 Inherited precondition satisfies new precondition*

Used by *Perfect Developer* only. Generated wherever a method declared with **define** or **redefine** also declares a precondition. In this case the new precondition must be implied by the precondition of the original method (i.e. the new precondition must be the same as the old or be weaker). Primary location is that of the new precondition; secondary location is that of the original precondition

## *3.16 Intermediate object satisfies class invariant*

Generated every time an intermediate instance of **self** or **it** for which it is possible to break the class invariant is used in any way, except to access a variable member or as an operand of the equality operator. Primary location is that of the **self** or **it** expression; secondary location is that of the invariant.

## *3.17 Internal class invariant satisfied*

Used by *Perfect Developer* only. Generated at the end of every implementation that modifies internal data of a class, to check that each internal invariant is satisfied. The primary location is that of the end of the implementation; the secondary location is that of the invariant being checked.

## *3.18 Jump satisfies precondition at label [name]*

Generated at each **goto** statement to check that the precondition of the label being jumped to is true. The primary location is that of the **goto** statement; the secondary location is that of the label.

### 3.19 Left identity declared for operator [operator] is valid

Used by *Perfect Developer* only. Generated for any operator declaration that includes the declaration of a left identity, to verify that the expression given really is a left identity. The primary location is that of the operator property declaration.

### 3.20 Loop body establishes end condition or decreases variant

Generated at the end of a loop body to check that either the **until** condition is true (i.e. the loop has terminated) or the variant has decreased. The primary location is that of the end of the loop body; the secondary location is that of the variant.

### 3.21 Loop body establishes end condition or preserves validity of variant

Generated at the end of a loop body to check that either the **until** condition is true (i.e. the loop has terminated) or that each integer variant component is non-negative. The primary location is that of the end of the loop body; the secondary location is that of the variant component being checked.

### 3.22 Loop body only modifies objects in 'change' list

Used by *Perfect Developer* only. Generated at the end of a loop body to check that only objects or parts of objects specified in the **change** list were actually changed by the loop body. The primary location is that of the end of the loop body; the secondary location is that of the **change** list.

### 3.23 Loop body preserves loop invariant

Generated at the end of a loop body to check that the invariant is true after each iteration. The primary location is that of the end of the loop body; the secondary location is that of the invariant being checked.

### 3.24 Loop initialisation establishes end condition or a valid variant

Generated at every loop statement to check that when the loop statement is reached (i.e. before the body has been executed at all), either the **until** condition is true (i.e. the loop body will not be executed at all), or all integer components of the variant (**decrease** part) are non-negative. The primary location is that of the loop; the secondary location is that of the variant component.

### 3.25 Loop initialisation establishes loop invariant

Generated at every **loop** statement to check that the stated invariant is true when the loop statement is reached (i.e. before the body has been executed at all). The primary location is that of the **loop** statement; the secondary location is that of the invariant being checked.

### 3.26 Method assertion implies 'require' assertion

Used by *Perfect Developer* only. Generated where a template with **require** declarations is instantiated. We check that each of the methods' assertions are implied by the assertions declared by the actual class methods. The primary location is that of the point of instantiation of the template; the secondary location is that of the **require** declaration.

## *3.27 Method precondition is implied by 'require' precondition*

Used by *Perfect Developer* only. Generated where a template with **require** declarations is instantiated. We check that each of the actual class methods' preconditions is implied by the preconditions given in the **require** declaration. The primary location is that of the point of instantiation of the template; the secondary location is that of the **require** declaration.

## *3.28 Modified object is in current writes-clause*

Used by *eCv* only. Generated when a nonlocal variable is modified and the identity of the variable being modified is not static, for example when writing through a pointer.

## *3.29 Nullable pointer expression is not null*

Use by *eCv* only.Generated when a pointer expression whose type includes the null pointer is required to be non-nul, either because of a **not_null**(...) cast or because it is dereferenced.

## *3.30 Objects modified in parallel are independent*

Used by *Perfect Developer* only. Generated for parallel postconditions (i.e. where conditions are combined with ',' or '&', and **forall** postconditions). The verification condition checks that the objects modified by each component of the condition are independent of each other. The **forall** postcondition with **bag** or **seq** bound also generates the verification condition that the collection is unique. Location is that of the postcondition.

## *3.31 Only variables modified in specification are modified by implementation*

Used by *Perfect Developer* only. Generated for every schema with an implementation, to check that every variable and component of a variable not specified as changed in the postcondition (or specified as changed only in certain circumstances) is unchanged by the implementation (or is changed only in the same circumstances as the postcondition specifies). The primary location is that of the **done** statement (or the end of the implementation); the secondary location is that of the postcondition.

## *3.32 Operand of [that | any] has at least one qualifying element*

Generated at every **any** expression and every **that** expression with a condition. For a **that** or **any** with a condition, the verification condition checks that there exists an element of the collection that satisfies the condition. For an **any** with no condition we simply check that the collection is non-empty. The location is that of the **any** or **that** expression.

## *3.33 Operand of 'is' within specified type*

Used by *Perfect Developer* only. Generated for every **is** cast, to check that the type of the expression is as stated. Location is that of the cast.

## *3.34 Operand of 'over' has at least one element*

Generated for every *op* **over** expression if no left identity has been declared for *op*, to check that the collection is non-empty. The location is that of the expression.

## *3.35 Operator [operator] is associative*

Used by *Perfect Developer* only. Generated for every operator declaration which is declared as having the **associative** property to verify that *a op (b  c) = (a op b) op c*. The location is that of the property declaration.

## 3.36 Operator [operator] is commutative

Used by *Perfect Developer* only. Generated for every operator declaration which is declared as having the **commutative** property to verify that *a op b = b op a*. The location is that of the property declaration.

## 3.37 Operator [operator] is idempotent

Used by *Perfect Developer* only. Generated for every operator declaration which is declared as having the **idempotent** property to verify that *a op a = a*. The location is that of the property declaration.

## 3.38 Operator '~~' is 'total'

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ and declares it to be **total** to check the property **x~~y = same@rank ==> x = y**. Location is that of the operator declaration.

## 3.39 Operator '~~' is reflexive

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ to check the property **x~~x** = **same@rank**. Location is that of the operator declaration.

## 3.40 Operator '~~' is transitive

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ to check the property **x~~y = y~~z ==> x~~y = x~~z**. Location is that of the operator declaration.

## 3.41 Operator '~~' refines inherited definition

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ in a class with an ancestor that also defines the operator ~~ to check that the new definition is a refinement of the ancestor definition, i.e. for any pair of operands for which the ancestor definition returns **above@rank** or **below@rank**, the new definition returns the same value. The primary location is that of the operator declaration; the secondary location is that of the ancestor operator declaration.

## 3.42 Operator '~~' satisfies first symmetry condition

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ to check the properties:

**x~~y = same@rank ==> y~~x = same@rank**

Location is that of the operator declaration.

## 3.43 Operator '~~' satisfies second symmetry condition

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ to check the properties:

**x~~y = above@rank ==> y~~x = below@rank**

Location is that of the operator declaration.

## 3.44 Operator '~~' satisfies third symmetry condition

Used by *Perfect Developer* only. Generated wherever the user defines the operator ~~ to check the properties:

**x~~y = below@rank ==> y~~x = above@rank**

Location is that of the operator declaration.

### 3.45 Post-assertion valid

Generated whenever a post-assertion is declared (other than in a deferred method) or inherited by a method declaration. The primary location is that of the end of the postcondition or function definition; the secondary location is that of the post-assertion being checked.

### 3.46 Postcondition satisfied when function returns

Used by *eCv* only. Generated for C functions that have both a postcondition and a body. The primary location is that of the end of the function body; the secondary location is that of the postcondition being checked.

### 3.47 Postcondition specifies value for uninitialised data

Generated for all constructors and schemas with **out** parameters. A check is generated for each uninitialised data member or **out** parameter to ensure that it has a value specified for it before being used. The location is that of the postcondition.

### 3.48 Precondition at label [name] satisfied after preceding statement

Generated at each labelled statement that can be reached by fall-through from the preceding statement, to check that the precondition is true at this point. The location is that of the label.

### 3.49 Precondition of [name] satisfied

Generated at the point of call to any function, operator, selector, constructor or schema with a precondition. Primary location is that of the call; secondary location is that of the precondition.

### 3.50 Precondition of 'absurd' declaration is always false

Used by *Perfect Developer* only. Generated for every **absurd** declaration to check that the precondition inherited for that method is always false given the invariants in the class containing the declaration. The primary location is that of the declaration; the secondary location is that of the inherited precondition.

### 3.51 Property satisfied

Used by *Perfect Developer* only. Generated for every **property** declaration. Location is that of the assertion being checked.

### 3.52 Return value satisfies specification

Generated at every **value** statement to check that the given value satisfies the specification. The primary location is that of the value statement; the secondary location is that of the specification of the value.

### 3.53 Selector still returns assigned value after assignment

Generated for every selector declaration to check that changing the value returned cannot affect which object the selector should return (e.g. if the selector contains a conditional, the branch selected cannot be changed by changing the value returned by the selector). The location is that of the selector declaration.

### 3.54 Specification satisfied at ‘done’

Generated at every **done** statement to check that the postcondition has been achieved. The primary location is that of the **done** statement; the secondary location is that of the postcondition.

### 3.55 Specification satisfied at end of implementation

Generated at the end of every implementation that does not end with a **done** or **value** statement to check that the postcondition has been achieved. The primary location is that of the **done** or **value** statement; the secondary location is that of the postcondition.

### 3.56 Type constraint for [type] satisfied

Used by *Perfect Developer* only. Generated whenever a value is assigned to a variable with a more constrained type. The primary location is that of the postcondition; the secondary value is that of the declaration of the constraint.

### 3.57 Type constraint satisfied

Generated in various contexts where a value is required to conform to a more constrained type.

### 3.58 Type constraint satisfied in conversion from [type] to [type]

Generated for any explicit or implicit conversion to a constrained type.

### 3.59 Variable [name] is not accessed unless its ‘when’ guard is true

Used by *Perfect Developer* only. Generated at each access to a variable that was declared with a **when**-guard, to ensure that the guard is true. Primary location is that of the variable access; secondary location is that of the variable declaration.

### 3.60 Variant decreases

Generated at the point of each recursive call. Checks the variant of the called method is less than the variant of the calling method. Primary location is that of the recursive call; secondary location is that of the variant of the called method.

### 3.61 Variant non-negative

Generated for all recursion variants with integer components. Assumes the method precondition. Location is that of the variant component.

*End of document*