Coverage for ISO/IEC 8652:2012 and subsequent corrections in ACATS 3.x and 4.x

Clauses 4.1.3 – 4.5.2

A Key to Kinds and subkinds is found on the sheet named Key. Tests new to ACATS 3.0 are shown in **bold**; ACATS 3.1 in ***bold italic***; ACATS 4.0 in **blue bold**; ACATS 4.1 in ***blue bold italic***. ACATS 4.2 in ***green bold italic***.

Clause	Para.	Lines	Kind	Subkind	Notes	Tests	New	Objective's Priority	Objective Text	Objective notes	Submitted tests (will need work).
4.1.3	(1)		Redundant								
	(2)		Syntax								
	(3)		Syntax								
	(4)		Definitions	Widely Used	Expanded Name						
	(5)		NameRes					6	Check that if the prefix of a selected component denotes an enclosing construct, it is not interpreted as a component reference.	C-Test. Try F.C inside a a function F that returns a record R with a component C, while the function has an object C of the same type. This should resolve and not make a recursive call.	
								6	Check that if the prefix of a selected component denotes an enclosing protected type, it is not interpreted as an external reference to a protected entry or subprogram.	B-Test (?)	
								6	Check that if the prefix of a selected component denotes an enclosing construct, it is not interpreted as a prefix view.	B-Test (?) or a C-Test like the one described above.	
	(6)		NameRes	Portion	Lead-in for next rule.						
	(7)		NameRes			C41301A			Check that for the reference L.R, if R represents a component or discriminant of a record type, then L can represent an object or value of that type.		
				Negative				4	Check that for the reference L.R, if R represents a discriminant of a private, task, or protected type, then L can represent an object or value of that type.	C-Test. Try cases like those found in C41301A. Simple cases probably exist in many other ACATS tests, thus the low priority.	
				Negative		B940005 contains a single example.		4	Check that for the reference L.R, if R represents a component of a protected type, and L represents an object or value of that type, the reference is illegal.	B-Test. Try many kinds of prefixes.	
	(8)		NameRes	Portion	Lead-in for next rule.						
	(9)		NameRes			C41306B (func, access-to-task), C41306C (func, access-to-task), ***C413006*** (not access)			Check that for the reference L.R, if L represents a task value or object, R can represent a task entry or family.		
								5	Check that for the reference L.R, if L represents a protected value or object, R can represent a protected entry or subprogram.	C-Test. Simple cases are scattered throughout the ACATS; we mainly need to test examples like those in C41306x.	
	(9.1/2)		NameRes	Portion	Lean-in for next rule.						
	(9.2/3)	1	NameRes		These objectives mostly cover the first three lines.	**C413001**	All		Check that for the reference L.R, if L represents an object or value of a tagged type T, that R may represent a subprogram with a first parameter of the type T that is declared immediately in the declarative region of an ancestor of T.		
						C413002	All		Check that for the reference L.R, if L represents an object or value of an access type designating a tagged type T, that R may represent a subprogram with a first parameter of the type T that is declared immediately in the declarative region of an ancestor of T.		

			C413001	All		Check that for the reference L.R, if L represents an object or value of a tagged type T, that R may represent a subprogram with a first parameter of a class-wide type that covers T that is declared immediately in the declarative region of an ancestor of T.	
			C413002	All		Check that for the reference L.R, if L represents an object or value of an access type designating a tagged type T, that R may represent a subprogram with a first parameter of a class-wide type that covers T that is declared immediately in the declarative region of an ancestor of T.	
			C413003	All		Check that for the reference L.R, if L represents an object or value of a tagged type T, that R may represent a subprogram with a first access parameter that designates T that is declared immediately in the declarative region of an ancestor of T.	
			C413004	All		Check that for the reference L.R, if L represents an object or value of an access type designating a tagged type T, that R may represent a subprogram with a first access parameter that designates T that is declared immediately in the declarative region of an ancestor of T.	
			C413003	All		Check that for the reference L.R, if L represents an object or value of a tagged type T, that R may represent a subprogram with a first access parameter that designates a class-wide type that covers T that is declared immediately in the declarative region of an ancestor of T.	
			C413004	All		Check that for the reference L.R, if L represents an object or value of an access type designating a tagged type T, that R may represent a subprogram with a first access parameter that designates a class-wide type that covers T that is declared immediately in the declarative region of an ancestor of T.	
			C413007	All		Check that for the reference L.R, if L represents an object or value of an access type designating a tagged type T with the value null, and R represents an appropriate subprogram for a prefixed view, that Constraint_Error is raised when the name L.R is evaluated.	
	Negative		**B413004**	Part	3	Check that for the reference L.R, if L represents an object or value of an non-access untagged type T or an access type designating an untagged type T, and R represents a subprogram with a first parameter of T, the reference is illegal even if the subprogram is primitive for T.	B-Test. Try other types, including protected, task, limited record, float, fixed, decimal, modular, enum. But this isn't very important.
	Negative		**B413004**	Part	2	Check that for the reference L.R, if L represents an object or value of an non-access untagged type T or an access type designating an untagged type T, and R represents a subprogram with a first access parameter designating T, the reference is illegal even if the subprogram is primitive for T.	B-Test. Try other types, including protected, task, limited record, float, fixed, decimal, modular, enum. But this isn't very important.
2	Negative		***B413005***	All		Check that for the reference L.R, if L represents an object of a tagged type T or an access type designating a tagged type T, and R represents a subprogram with a first parameter of the type T or a class-wide type that is covered by T that is not declared immediately in the declarative region of an ancestor of T, the reference is illegal.	Could try other tagged types, including limited, private, task, and protected. But unlikely to find anything.

3		Negative		***B413005***	All		Check that for the reference L.R, if L represents an object of a tagged type T or an access type designating a tagged type T, and R represents a subprogram with some parameter other than the first parameter of the type T and a first parameter of a non-access untagged type that is declared immediately in the declarative region of an ancestor of T, the reference is illegal.	Could try other tagged types, including limited, private, task, and protected. But unlikely to find anything.
4				**B413001**	All		Check that the reference L.R is not interpreted as a prefixed view if the designator R represents a component of the type T visible at the point of the reference.	
		Negative				3	Check that the reference L.R can be interpreted as a prefixed view if the designator R represents a component of the type T that is not visible at the point of the reference.	C-Test. B431001 includes this case, which is why the priority is low.
5		Widely Used	A new rule in Ada 2012, necessary to allow ordinary Ada83-style prefix calls to tagged task and protected operations. We don't need to test this separately as any test of tagged task or protected types will necessarily make prefix calls.					
6	StaticSem	Subpart	Prefixed view calls are tested in 6.4(10.1/2).					
				C413005	All		Check that a prefixed view is the name of a subprogram (with the first parameter omitted from the profile) that can be renamed and passed as a generic formal parameter.	
		Negative		***B413005***	All		Check that a call of a prefixed view cannot repeat the first parameter in the parameter list.	Could try other tagged types, including limited, private, task, and protected. But unlikely to find anything.
6	Definitions		Prefixed view.					
(10)	NameRes	Portion	Lead-in for the following rules.					
(11)	NameRes	Subpart	Tested in the next two rules.					
(12)	NameRes					3	Check that for the reference L.R, if L represents the name of a package, then R can name any visible declaration in the package.	C-Test: commonly used but no obvious test to report here.
				C41320A (enum), C41321A (derived Boolean), C41322A (signed integer), C41323A (float), C41324A (fix), C41325A (array), C41326A (access), C41327A (private), C41328A (inherited subs, derived type)		2	Check that for the reference L.R, if L represents the name of a package, then R can name any implicitly declared declarations in the visible part of the package.	C-Test. Need modular types and decimal fixed types; maybe types derived from interfaces as well.
						4	If L represents the name of a package, check that for the reference L.R given in the private part or body of package L or the private part or body of a public child of L or in a private child of L, then R can name any declaration in the package private part of the package.	C-Test. Expanded names don't appear in the relevant Section 10 tests, so we need them here.
						5	If L represents a renaming of a package P, check that for the reference L.R, R can name any visible declaration in the package P.	C-Test. An untested Ada 83 objective.
		Negative				4	Check that for the reference L.R, if L represents the name of a package, then R cannot name any declaration of the package not visible at the point of the reference.	B-Test.

			Negative					3	Check that for the reference L.R, if L represents the name of an enclosing construct, then R cannot name an entity declared other than in that enclosing construct.	B-Test. Try items declared in nested and outer scopes.
	(13)	1	NameRes			C41307D		4	Check that an expanded name can reference a declaration in a callable construct, type declaration, accept statement, block statement, or loop statement if it is given within that construct.	C-Test (need to test a protected function, protected procedure, and entry body).
								5	Check that for reference L.R, L can be an operator symbol if R is a declaration in that operator and the reference occurs within the operator.	C-Test. An untested Ada 83 objective.
			Negative					4	Check that an expanded name is illegal if it tries to reference a declaration inside of a callable construct, accept statement, block statement, or loop statement outside of that construct by naming the construct in its prefix.	B-Test.
			Negative					4	Check that an expanded name is illegal if it tries to reference a declaration inside of a type declaration outside of the type declaration by naming the type in its prefix.	B-Test.
			Negative					4	Check that a family index is not allowed in an expanded name for an access statement or entry body.	B-Test.
		2						6	Check that if expanded name occurs within a callable construct, and the prefix of an expanded name denotes more than one enclosing callable construct, the expanded name is illegal.	B-Test.
	(13.1/5)		Legality	Subpart	Prefixed view calls are tested in 6.4(10.1/2).					
			Negative			**B413002**	All		Check that the prefixed view L.R is illegal if the first parameter of R is an access parameter and L is not an aliased view of an object.	
	(13.2/2)		Legality	Subpart	Prefixed view calls are tested in 6.4(10.1/2).					
			Negative			**B413003**	All		Check that the prefixed view L.R is illegal if the first parameter of R is a parameter with mode in out or out and L does not denote a variable.	
			Negative			**B413003**	All		Check that the prefixed view L.R is illegal if the first parameter of R is a parameter with an access-to-variable type and L does not denote a variable.	
	(14)		Dynamic	Widely Used	Testing the evaluation of the name will necessarily test evaluation of the prefix.					
						C41304A			Check that L.R raises Constraint_Error when L has the access value null.	
	(15)		Dynamic			C41304B			Check that L.R raises Constraint_Error when L denotes a record object with discriminant values such that component R does not exist.	
	(16)		NonNormative		Start of examples.					
	(17/2)		NonNormative							
	(18)		NonNormative							
	(19)		NonNormative		End of examples.					
4.1.4	(1)		Redundant							
	(2/5)		Syntax		AI12-0262-1 adds reduction attributes, but that is tested elsewhere.					

					C41404A ('Image'First, etc.)	2	Check that the prefix of an attribute can be another attribute.	C-Test. Check that T'Class'something and T'Base'something work. These may be covered by other existing tests scattered throughout the test suite.
(3/2)		Syntax						
(4)		Syntax						
(5)		Syntax						
(6/5)	1	NameRes		AI12-0242-1 changes this rule so it doesn't interfere with the definition of reduction attributes.	C41401A (Callable, Terminated, First, Last, Length, Range)	4	Check that the prefix of attributes that do not apply to objects of an access types can be interpreted as an implicit dereference.	C-Test. Test Component_Size, Constrained, Tag, and Valid. Test by determining that a prefix with the value null raises Constraint_Error (and resolves).
					C41402A (Address, Size, First_Bit, Last_Bit, Position)	4	Check that the prefix of attributes that apply to objects of an access types are not interpreted as an implicit dereference.	C-Test. Test Access, Alignment, Storage_Size, Unchecked_Access. Test by determining that a prefix with the value null does not Constraint_Error.
	2					5	Check that the prefix of attributes that apply to objects but not functions is interpreted as a parameterless function call.	C-Test. Test Callable, Terminated, First, Last, Length, Range, Size, First_Bit, Last_Bit, Position, Alignment, Storage_Size, Component_Size, Constrained, Tag, Valid.
						5	Check that the prefix of attributes that apply to both objects and functions is never interpreted as a parameterless function call.	C-Test. The Address attribute: check that the function is not called. B-Test: The prefix of Access can't be a function for a access-to-object type. (Is this a good idea?)
			Negative	These are the only rules for most attributes.	B87B26A (First_Bit, Last_Bit, Position, Callable, Terminated, First, Last, Length, Range, Count)	4	Check that information about the kind of entity expected as the prefix of attributes without extra resolution rules is not used to resolve the prefix.	B-Test(s). Test any untested attributes for which an example can be made (all of the ones mentioned/tested for previous objectives are candidates; there probably are others). AARM 4.1.4(6.d-h) give some examples for the Valid attribute.
(7)		NameRes				1	Check that the expression of a First, Last, Length, or Range attribute can be resolved even if there are interpretations of a non-integer type.	C-Test. I believe this objective cannot be tested because of the requirement that this expression be static (we need user-defined functions to get interesting overloading). It could be tested in Ada 2022, but still not very likely to occur.
(8)		Legality	Negative		B36201A has a single test case for Length.	2	Check that the expression of a First, Last, Length, or Range attribute must be static.	B-Test. Test the other three attributes, and try cases where the non-staticness isn't obvious (as for a constant defined of a generic formal integer type).
(9/3)	1	StaticSem	Subpart	This is untestable by itself; it will be tested as part of testing each attribute.				
	2			Added by AI05-0006-1.		4	Check that a First, Last, First_Valid, or Last_Valid attribute can be used as the expression of a case statement, and coverage is required for the base subtype of its type.	B-Test.
	3			Added by AI05-0006-1.		4	Check that a Pred, Succ, Val, or Input attribute can be used as the expression of a case statement, and coverage is required for the base subtype of its type.	B-Test.
(10)		Redundant						
(11/5)	1	Dynamic	Subpart	Changed by AI12-0262-1, just to avoid conflicts with 4.5.10.		3	For attributes designating objects, check that evaluating the attribute evaluates the prefix.	C-Test. This is slightly covered by the tests for the objectives of 4.1.4(6), thus the low priority.
	2	Redundant		Should be tested by tests for 4.5.10.				

	(12/1)		Impl-Def	Not Testable	The only effect of this is that a test checking that Small is not defined for floating point types is incorrect.					
	(13)		NonNormative		A note.					
	(14/2)		NonNormative		Another note.					
	(15)		NonNormative		Start of examples.					
	(16)		NonNormative		End of examples.					
4.1.5	(1/3)		StaticSem	Portion	This entire subclause is new in Ada 2012. The rule is tested below.					
	(2/3)		StaticSem	Negative		***B415001***	All	Check that the name given for an Implicit_Dereference aspect must be that of an access discriminant for the associated type.		
	(3/3)		Definitions	Subpart	This will necessarily be used in any C-Tests testing other rules here.					
	(4/3)		Syntax							
	(5/3)		NameRes	Subpart	This will necessarily be used in any C-Tests testing other rules here.					
	(5.1/4)		StaticSem	Subpart	Added by AI12-0138-1. The rules are enumerated in 13.1.1(18.2-5/4), and the objectives are there.					
	(6/3)		StaticSem		Not clear that any semantic effect of this beyond those caused by the dynamic rules (there doesn't seem to be any other sensible meaning).					
	(7/3)	1			Note: The “if not overridden” wording doesn't need to be tested, as 5.1/4 makes it illegal to override.			7	Check that a generalized reference can be used for an object of a derived type that inherits the Implicit_Dereference aspect from its parent type.	C-Test.
		2						6	Check that a generalized reference for a derived type refers to the new discriminant when that discriminant constrains an inherited reference discriminant.	C-Test.
		3		Redundant	Even though this is redundant (because it follows from the rules for constraining an inherited discriminant), we test it here as it's unlikely that such a combination would be tested elsewhere.			6	Check that a generalized reference for a derived type whose inherited reference discriminant is constrained refers to the constrained value.	C-Test.
	(8/3)	1	Dynamic					3	Check that the reference_object_name is evaluated by the evaluation of a generalized reference.	C-Test. Try a name containing a function call. Low priority because it's hard to imagine a compiler getting this wrong, being the same as evaluating any other name that's part of an expression.
		2	Dynamic			**C415001**	All		Check that Constraint_Error is raised by a generalized reference whose discriminant value is null.	
		3	Dynamic	Portion	The rest of the rule is in the previous paragraph.					
		4	Dynamic			**C415001**	Part	4	Check that a generalized reference denotes the object or subprogram designated by the value of the reference discriminant.	Need a C-Test that tries this for an access-to-subprogram discriminant (but much less important).
						C415001	All		Check that the object denoted by a generalized reference can be modified if the discriminant has an access-to-variable type.	

							B415002	All	Check that the object denoted by a generalized reference cannot be used as a variable if the discriminant has an access-to-constant type.	
	(9/3)		NonNormative							
	(10/3)		NonNormative							
	(11/3)		NonNormative							
	(12/3)		NonNormative							
	(13/3)		NonNormative							
	(14/3)		NonNormative							
	(15/3)		NonNormative							
4.1.6	(1/3)		StaticSem	Negative	This entire subclause is new in Ada 2012.		**B416001**	All	Check that a Constant_Indexing or Variable_Indexing aspect can only be specified on a tagged type declaration.	
	(2/3)	1	StaticSem	Negative			**B416001**	Part	4 Check that name of a Constant_Indexing aspect cannot denote an entity other than a function declared in the same declaration list as the type declaration.	B-Test. Should try denoting the wrong kind of entity (procedures in particular).
									6 Check that the name of a Constant_Indexing aspect can denote entities in other scopes so long as at least one qualifying function exists in the same declaration list as the type declaration.	C-Test. We don't want other visible things to cause issues. Test in a child package where the parent makes conflicting things visible.
									4 Check that the name of a Constant_Indexing aspect can denote other kinds of entities in the same declaration list so long as at least one qualifying function exists in the same declaration list as the type declaration.	C-Test. (Procedures in particular.) Low priority as it doesn't seem particularly likely to occur in practice.
		2		Negative			**B416001**	All	Check that the name specified by a Constant_Indexing aspect cannot denote a function with zero or one parameters.	
				Negative			**B416001**	All	Check that the name specified by a Constant_Indexing aspect cannot denote a function whose first parameter has a type other than T or T'Class or an access-to-constant designating T or T'Class.	
							B416001, *C416A01*	All	Check that the name specified for a Constant_Indexing can refer to a set of overloaded functions.	
							B416001, *C416A01*	All	Check that the name specified for a Constant_Indexing can have more than two parameters.	
	(3/3)	1	StaticSem	Negative			**B416001**	Part	4 Check that name of a Variable_Indexing aspect cannot denote an entity other than a function declared in the same declaration list as the type declaration.	B-Test. Should try denoting the wrong kind of entity (procedures in particular).
									6 Check that the name of a Variable_Indexing aspect can denote entities in other scopes so long as at least one qualifying function exists in the same declaration list as the type declaration.	C-Test. We don't want other visible things to cause issues. Test in a child package where the parent makes conflicting things visible.
									4 Check that the name of a Variable_Indexing aspect can denote other kinds of entities in the same declaration list so long as at least one qualifying function exists in the same declaration list as the type declaration.	C-Test. (Procedures in particular.) Low priority as it doesn't seem particularly likely to occur in practice.
		2		Negative			**B416001**	All	Check that the name specified by a Variable_Indexing aspect cannot denote a function with zero or one parameters.	
				Negative			**B416001**	All	Check that the name specified by a Variable_Indexing aspect cannot denote a function whose first parameter has a type other than T or T'Class or an access-to-variable designating T or T'Class.	

	3		Negative		**B416001**	All		Check that the name specified by a Variable_Indexing aspect cannot denote a function that returns a type other than a reference type for an access-to-variable.	
					B416001, *C416A01*	All		Check that the name specified for a Variable_Indexing can refer to a set of overloaded functions.	
					B416001, *C416A01*	All		Check that the name specified for a Variable_Indexing can have more than two parameters.	
(4/3)	1	StaticSem			***C416A02***	All		Check that a generalized indexing can be used for an object of a derived type that inherits the Constant_Indexing or Variable_Indexing aspect from its parent type.	
	2	StaticSem	Redundant	This sentence is deleted by AI12-0104-1.					
(5/3)		Definitions							
(5.1/4)		StaticSem	Subpart	Added by AI12-0138-1. The rules are enumerated in 13.1.1(18.2-5/4).					
(6/4)		Deleted		Removed by AI12-0138-1 (Corrigendum).					
(7/4)		Deleted		Removed by AI12-0138-1.					
(8/4)		Deleted		Removed by AI12-0138-1.					
(9/4)		Deleted		Removed by AI12-0138-1.					
(10/3)		Syntax							
(11/3)		NameRes					2	Check that the prefix of a generalized indexing resolves if it denotes overloaded functions where one option is a function returning an indexable container type and the other function is not.	C-Test. Low priority because the other overloaded function has to return a type that cannot be any kind of prefix (else it probably would ambiguous for other reasons). It's also not clear that requiring this level of resolution is a good idea.
(12/3)		NameRes	Portion	This is lead-in text.					
(13/3)		NameRes					5	Check that a generalized indexing calls the Constant_Indexing function if no Variable_Indexing is specified, even in a variable context.	C-Test. To try variable contexts, a Constant_Indexing function that returns an access-to-variable type is needed; a dereference then can be used in a variable context. Assume that the object being indexed is some sort of handle.
					B416002	All		Check that a generalized indexing is illegal in a variable context if no Variable_Indexing is specified and Constant_Indexing specifies a function returning an ordinary object.	
							5	Check that a generalized indexing is illegal in a variable context if no Variable_Indexing is specified and Constant_Indexing specifies a function returning a reference type with an access constant discriminant.	B-Test. Not very important (unlikely the previous would work and this would fail), but could make a version of the existing test.
(14/3)		NameRes			***C416A01***	Part	6	When both Constant_Indexing and Variable_Indexing are specified, check that the variable indexing function is called in variable contexts if the prefix is a variable, and that the constant indexing function is called in all other cases (including variable indexing contexts when the prefix is a constant).	C-Test. To try variable contexts, a Constant_Indexing function that returns an access-to-variable type is needed; a dereference then can be used in a variable context. Assume that the object being indexed is some sort of handle. (Existing test checks that the right routine is called when the prefix is a variable.) C4160RB
					B416A01	All		When both Constant_Indexing and Variable_Indexing are specified, check that a generalized indexing is illegal if it is called in variable contexts when the prefix is a constant and Constant_Indexing specifies a function returning an ordinary object.	

									When both Constant_Indexing and Variable_Indexing are specified, check that a generalized indexing is illegal if it is called in variable contexts when the prefix is a constant and Constant_Indexing specifies a function returning a reference	B-Test. Not very important (unlikely the previous would work and this would fail, also partially covered by the containers tests), but could make a version of the existing test.
								5	type with an access constant discriminant.	
									When only a Variable_Indexing is specified, check that a	B-Test. (There's no fallover in this case, like there is in the others.)
								6	generalized indexing with a prefix of a constant is illegal.	
	(15/3)		NameRes	Portion	Included in the other test objectives.					
	(16/3)		NameRes	Portion	Included in the other test objectives.					
	(17/3)		NameRes	Subpart	This is necessarily tested in any C-Test that uses a generalized indexing.					
	(18/4)		NonNormative		Added by AI12-0104-1. We test these cases here as there is no other natural point to do so, and they're important.	***C416A02***	All		Check that if a function used by an inherited Constant_Indexing or Variable_Indexing is overridden, the overridden function is called by a generalized indexing.	
						C416A02	All		Check that if a function used by an inherited Constant_Indexing or Variable_Indexing is overloaded (with a different profile), the overloaded function can be called by a generalized indexing.	
	(19/3)		NonNormative		Example; the paragraphs were renumbered by AI12-0104-1.					
	(20/3)		NonNormative							
	(21/3)		NonNormative							
	(22/3)		NonNormative							
4.2	(1)	1	Redundant							
		2	Definitions	Widely Used	Literal					
	(2/2)		Deleted							
	(3)		NameRes					4	Check that the value of a character literal is not used to determine its type.	B-Test. Try a call of two overloaded procedures taking parameters of different character types, only one of which has the appropriate literal. This is marked as untested in ACATS 2.x.
								3	Check that an overloaded call can be resolved when an actual parameter is a character literal and only one of the subprograms has a character type parameter.	C-Test. This is marked as untested in ACATS 2.x.
								3	Check that a character literal can be used as the actual for an appropriate formal function.	C-Test. This is marked as untested in ACATS 2.x.
						B46002A (type conversion)		2	Check that a character literal is illegal in a context that does not identify a single type.	B-Test. Try other contexts that require a single type (if there are any that allow characters).
	(4/5)		NameRes			C87B27A			Check that an overloaded call can be resolved when an actual parameter is a string literal and only one of the subprograms has a string type parameter.	
						B46002A (type conversion)		2	Check that a string literal is illegal in a context that does not identify a single type.	B-Test. Try other contexts that require a single type (if there are any that allow strings).
								2	Check that an expression will not resolve if the expected type of an integer literal is not an integer type or one that has Integer_Literal specified.	B-Test. Try many other kinds of types: float, fixed, enumeration, access, record, array, task, protected, etc.
								2	Check that an expression will not resolve if the expected type of a real literal is not an float or fixed type or one that has Real_Literal specified.	B-Test. Try many other kinds of types: integer, enumeration, access, record, array, task, protected, etc.

	(5)		Legality	Widely Used	Any character literal.			
				Negative		2	Check that a character literal is illegal if it is not a value of the expected type.	B-Test. This is marked as untested in ACATS 2.x.
	(6/5)		Legality	Widely Used	Any string literal. Text added by AI12-0295-1 to exclude user-defined string literals from this requirement. That will be tested by any legal user-defined string literal.			
				Negative		4	Check that a string literal is illegal if any character is not a value of the component type of the expected type.	B-Test. This is marked as untested in ACATS 2.x. Also see 4.3.3(19).
	(7/2)		Deleted					
	(8/5)		Definitions	Widely Used	Types of literals (most moved to paragraph 4 by AI12-0373-1).			
				Negative		2	Check that an expression will not resolve if the expected type of null is not an access type.	B-Test. Try many other kinds of types: integer, float, fixed, enumeration, access, record, array, task, protected, etc.
	(9/5)	1	Dynamic	Widely Used	Nothing will work if the values of literals are wrong. AI12-0249-1 adds wording to exclude user-defined numeric literals.			
		2	Dynamic	Widely Used	Nothing will work if null isn't the null value.			
		3	Dynamic	Portion	The actual test is in 4.2.1; added by AI12-0249-1.			
	(10/5)		Dynamic		Text added by AI12-0295-1 to exclude user-defined string literals from this definition.	C42007E	Check that the bounds of non-null string literals are determined properly.	Also see 4.3.3(26).
						3	Check that if the upper bound of a non-null string literal is outside of the appropriate index subtype, Constraint_Error is raised.	C-Test.
						2	Check that the bounds of null string literals are determined properly when the upper bound is in the index base type.	C-Test.
	(11/5)	1	Dynamic		Text added by AI12-0295-1 to exclude user-defined string literals from this check.	C42006A	Check that if any character of a string literal does not belong to the dynamic component subtype of the expected type, Constraint_Error is raised.	
						2	Check that if any character of a static string literal does not belong to the static component subtype of the expected type, the literal is illegal.	B-Test. This happens because of 4.9(34).
		2	Dynamic			C420001	Check that non-static null string literals whose upper bound is not in the index base type raise Constraint_Error.	
						B420001	Check that static null string literals whose upper bound is not in the index subtype are illegal.	This happens because of 4.9(34).
	(12)		NonNormative		A note.			
	(13)		NonNormative		An example.			
	(14)		NonNormative					
4.3	(1)		Definitions	Subpart	Aggregate, test the individual types.			
	(2/5)		Syntax		Delta aggregate is added by AI12-0127-1; container aggregate by AI12-0212-1.			

	(3/5)		NameRes		Rules expanded by AI12-0307-1 to cover delta aggregates and container aggregates.	B43005A (array dimensions), B43005B (number of components), B43005F (mixed notation), B43105C (type of expressions), B432221A (completeness of array), B43221B (length of array), B43223A (others choice)		2	Check that the contents of an aggregate are not used to resolve it.	B-Tests. Additional cases (extension aggregates must be tagged, null record must be record) should be tried here.
								5	Check that limitedness is not used to resolve expressions containing aggregates	B-Test. Try a subprogram overloaded on limited and nonlimited record types.
								5	Check that an overloaded call can be resolved when an actual parameter is an aggregate and only one of the subprograms has a record or array parameter.	C-Test. Try record, array, and extension. Contrast to private types completed by record or arrays (which are not considered). Not tested in ACATS 2.x.
						B46002A (type conversion)		4	Check that an aggregate is illegal in a context that does not identify a single type.	B-Test. Try an aggregate as the ancestor of an extension aggregate. Try delta aggregates.
	(4/5)		Legality			B430001			Check that an aggregate cannot be of a class-wide type.	
	(5)	1	Dynamic	Widely Used	Any aggregate does this.					
		2	Dynamic	Not Testable	This says that no order can be depended upon.					
		3	Dynamic	Widely Used	Any aggregate will test this.					
	(6)		Dynamic					4	Check that an aggregate of a tagged type that does not belong to the first subtype of its type raises Constraint_Error.	C-Test. A constrained first subtype with inherited discriminants is necessary, and an ancestor object that is not in that subtype.
4.3.1	(1)		Redundant							
	(2)		Syntax							
	(3)		Syntax							
	(4/2)		Syntax							
				Negative	This grammar may be ambiguous.	**B431001**	All		Check that a positional component association in a record aggregate cannot have a <> rather than an expression.	
						B431002	All		Check that a positional component association in an extension aggregate cannot have a <> rather than an expression.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
	(5)		Syntax							
	(6)	1	Definitions	Widely Used	Any aggregate will test one or the other.					
		2	Legality			C43106A			Check that positional components can precede any named components in a record aggregate.	
								4	Check that positional components can precede any named components in an extension aggregate.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
				Negative		B43002K			Check that named components cannot precede any positional components in a record aggregate.	
						B431002	All		Check that named components cannot precede any positional components in an extension aggregate.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
		3	Legality	Subpart	Other tests will test the others choice.					

			Negative		B43002F, B43002H		Check that an others component association cannot appear anywhere in a record aggregate other than last.	Note: This also prevents multiple others association, since one of them is not last.
					B431002	All	Check that an others component association cannot appear anywhere in an extension aggregate other than last.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
(7)		NameRes		This is not really a syntax rule, but rather a resolution one, because of the ambiguity in the syntax.	C87B29A		Check that an expression surrounded by parens is interpreted as a parenthesized expression, not a record aggregate.	
(8/2)		NameRes	Widely Used	This simply determines the type of the aggregate for reference to other rules; real resolution issues are tested for 4.3(3/2).				
					C431A01	All	Check that a record aggregate can have a limited type.	We only test this objective because it is a change from Ada 95.
(9/5)	1	Definitions	Subpart	Needed - tested by other aggregate tests.				
	2	NameRes	Subpart	Tested by any named notation. Clarified that this does not apply to delta aggregates.				
							Check that the selector names in a record aggregate can only name components and discriminants of the record type, and cannot name components of other variants.	
			Negative		B43101A			
							Check that the selector names in an extension aggregate can only name components and discriminants of the record extension, and cannot name components of other variants or of the type of the ancestor part..	
					B431004	All		Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
(10)		NameRes	Portion	Lead-in for following bullets				
(11)		NameRes			C87B30A		Check that overloaded expressions can be resolved in positional associations of a record aggregate because the type of the associated component is known.	
	3						Check that overloaded expressions can be resolved in positional associations of an extension aggregate because the type of the associated component is known.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
(12)		NameRes			C43105A, C43105B, C87B30A		Check that overloaded expressions can be resolved in named associations of a record aggregate because the type of the associated component is known.	
	3						Check that overloaded expressions can be resolved in named associations of an extension aggregate because the type of the associated component is known.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
(13)		NameRes					Check that overloaded expressions can be resolved in others associations of a record aggregate because the type of the associated component is known.	
	3							C-Test.
							Check that overloaded expressions can be resolved in others associations of an extension aggregate because the type of the associated component is known.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
(14)		Legality			C431001		Check that a record aggregate can be for a record extension if it is not descended from any private types.	
			Negative		**B431003**	All	Check that the type of a record aggregate cannot be for a record extension that is descended from any private type or private extension.	Not tested in ACATS 2.x.
			Negative	AI05-0115-1 changed the definition of descended from to make it clear visibility is involved.	***B431007***	All	Check that the type of a record aggregate cannot be a derived type that has an ancestor for which the current view of the parent of the derived type is not a descendant of the full view of the ancestor.	

(15/3)		Legality		These objectives use the approved correction of AI05-0016. AI12-0127-1 rewords so that it is clear this doesn't apply to delta aggregates.	B430001, C431001			Check that **null record** may appear in place of component associations if no components are needed in a record aggregate.	This isn't the primary objective of these tests, but it is tested.
					B430001, C432001, C432004			Check that **null record** may appear in place of component associations if no components are needed in an extension aggregate.	This isn't the primary objective of these tests, but it is tested.
			Negative		B430001			Check that **null record** cannot appear in place of component associations if any components are needed in a record aggregate.	
					B430001			Check that **null record** cannot appear in place of component associations if any components are needed in an extension aggregate.	
(16/4)	1	Legality	Subpart	Tested in every legal aggregate. Wording was clarified that it only applies to record and extension aggregates by AI12-0127-1.					
				We test others => <> separately because it is new, and because it is different than other associations.	**C431A01, *C431002***	All		Check that a component association of others => <> in a record aggregate may have any number of associated components, including none.	
					C431A01, *C431002*	All		Check that a component association of others => <> in an extension aggregate may have any number of associated components, including none.	
			Negative		B43101A (others)		2	Check that a component association (other than others => <>) in a record aggregate is illegal if it does not have an associated component.	B-Test: Test too many positional components.
			Negative		**B431004**	All		Check that a component association (other than others => <>) in an extension aggregate is illegal if it does not have an associated component.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
			Negative		B43101A			Check that a record aggregate is illegal if it has needed components that are not associated with any component associations.	
			Negative		**B431004**	All		Check that an extension aggregate is illegal if it has needed components that are not associated with any component associations.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
			Negative		B43101A (two named choices, one positional and one named)			Check that a record aggregate is illegal if it has a needed component that is associated with more than one component association.	
			Negative		**B431004**	All		Check that an extension aggregate is illegal if it has a needed component that is associated with more than one component association.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
	2			Sentence added by AI12-0127-1. Tested by every legal record delta aggregate.					
(16.1/5)	1	Legality	Subpart	Test should be checked when the number of expression evaluations is tested. This was separated into a stand-alone paragraph by AI12-0127-1.					
				We test others => <> and A|B => <> separately because they are new, and because they are different than other associations.	**C431A01**	All		Check that a component association in a record aggregate with a <> may have two or more associated components of different types.	
					C431A01	All		Check that a component association in an extension aggregate with a <> may have two or more associated components of different types.	These rules also apply to extension aggregates, so we test them for those as well.

				Allowed by AI05-0199-1			7	Check that a component association in a record aggregate may have two or more associated components with anonymous access types that statically match.	C-Test.
				Allowed by AI05-0199-1			6	Check that a component association in an extension aggregate may have two or more associated components with anonymous access types that statically match.	C-Test.
			Negative		B43101A			Check that a component association in a record aggregate with an expression cannot have two or more associated components of different types.	
					B431004	All		Check that a component association in an extension aggregate with an expression cannot have two or more associated components of different types.	Note: These rules also apply to extension aggregates, so we test the rules for them as well as record aggs.
	2			From approved AI12-0046-1.	***B431005***	All		Check that Legality Rules are enforced for all associated components, even when the results vary.	
(17/5)		Legality	Subpart	Any aggregate for a variant record will test. Modified by Ai12-0127-1, but no meaning change.					
			Negative	Modified by AI05-0220-1; the previous rule could be circular. Modified again by AI12-0086-1.	***B431006***	All		Check that if a variant part is not nested in an unselected variant, the value of the governing discriminant of a variant in a record aggregate cannot be non-static.	Note: This objective is wrong, strictly speaking, after the post-Corrigendum document is issued, but the test is OK even when AI12-0086-1 is in effect.
			Negative				3	Check that if a variant part is not nested in an unselected variant, the value of the governing discriminant of a variant in an extension aggregate cannot be non-static unless the subtype is static and the subtype covers only one variant part.	B-Test. Low priority because mixing variants and extensions is rare.
			Double Negative		C43103A, C43102B		3	Check that if a discriminant does not govern a variant part, or the variant part is nested within a variant that is not selected, the value in a record aggregate can be non-static. Probably should correct the objective to be similar to the next one, but that's not a priority.	C-Test: check that the last aggregate in the question of AI05-0220-1 is legal (already tried in B431006). The other case is adequately tested by the existing tests.
			Double Negative				3	Check that if a discriminant does not govern a variant part, or the variant part is nested within a variant that is not selected, the value in an extension aggregate can be non-static.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
(17.1/2)		Legality			***C431003***	All		Check that the association in a record aggregate for a discriminant with a default can be given by <>.	
					C431004	All		Check that the association in an extension aggregate for a discriminant with a default can be given by <>.	These rules also apply to extension aggregates, so we test them for those as well.
			Subpart	Discriminant associations with expressions are tested by many legal aggregates.					
			Negative		***B431008***	All		Check the association in a record aggregate for a discriminant without a default cannot be given by <>.	
			Negative		***B431009***	All		Check the association in an extension aggregate for a discriminant without a default cannot be given by <>.	These rules also apply to extension aggregates, so we test them for those as well.
(18)		Dynamic	Not Testable	Defines basic execution.					
(19)	1	Dynamic	Widely used	Basic execution is tested by any record aggregate.					
					C43004A, C43004C		2	Check that each expression of a record aggregate is converted to the appropriate subtype and appropriate checks are made.	C-Test. Try array conversions (length checks).

								3	Check that each expression of an extension aggregate is converted to the appropriate subtype and appropriate checks are made.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
	2							8	Check that per-object constraints are elaborated in a record aggregate, and that happens before the associated expression is evaluated and after the value of the discriminant is evaluated.	C-Test. Be sure to check cases where the per-object constraint raises an exception. Marked as not tested in ACATS 2.x.
								7	Check that per-object constraints are elaborated in an extension aggregate, and that happens before the associated expression is evaluated and after the value of the discriminant is evaluated.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
(19.1/2)	1	Dynamic	Widely used	Any record aggregate.						
	2	Dynamic			**C431A02**	All			Check that for each association with a <> in a record aggregate, if the associated component has a default expression, that expression is used and not the default initialization of the type of the component.	
					C431A02	All			Check that for each association with a <> in an extension aggregate, if the associated component has a default expression, that expression is used and not the default initialization of the type of the component.	These rules also apply to extension aggregates, so we test them for those as well.
					C431A02	All			Check that for each association with a <> in a record aggregate, if the associated component does not have a default expression, the component is initialized by default.	
					C431A02	All			Check that for each association with a <> in an extension aggregate, if the associated component does not have a default expression, the component is initialized by default.	These rules also apply to extension aggregates, so we test them for those as well.
					C431A01, C431A03	All			Check that if a <> in a record aggregate has multiple associated components, each one is appropriately initialized (either from the default expression or the initialized by default). In particular, check that if these components have the same type and default expression, the expression is evaluated for each one.	
					C431A01, C431A03	All			Check that if a <> in an extension aggregate has multiple associated components, each one is appropriately initialized (either from the default expression or the initialized by default). In particular, check that if these components have the same type and default expression, the expression is evaluated for each one.	These rules also apply to extension aggregates, so we test them for those as well.
(20)		Dynamic			C43107A				Check that each expression in a record aggregate is evaluated once for each associated component when there are multiple associated components.	
								4	Check that each expression in an extension aggregate is evaluated once for each associated component when there are multiple associated components.	C-Test. These rules also apply to extension aggregates, so we test them for those as well.
(21)		NonNormative	A note.							
(22)		NonNormative	Start of examples...							
(23)		NonNormative								
(24)		NonNormative								
(25)		NonNormative								
(26)		NonNormative								
(27/2)		NonNormative								
(28)		NonNormative								
(29)		NonNormative								
(29.1/2)		NonNormative								

	(29/2/2)		NonNormative							
	(30)		NonNormative							
	(31)		NonNormative		...end of examples.					
4.3.2	(1)		Redundant							
	(2)		Syntax							
	(3)		Syntax							
	(4/2)	1	Legality		Resolution tested under 4.3(3/2).					
				Negative	This rule is not a Name Resolution rule; that's in 4.3(3/2).	**B432001**	All		Check that an extension aggregate cannot be of a tagged record type or of a private extension.	Not tested in ACATS 2.x.
		2	NameRes	Subpart	Any ancestor part expression will test.					
						C432005	All		Check that the ancestor expression type in an extension aggregate can be limited.	
				Negative		**B432001**	All		Check that the type of an extension aggregate is not used to resolve an ancestor part expression.	Not tested in ACATS 2.x.
	(5/3)	1	Legality	Subpart	Any ancestor part subtype will test.					
						C432001 (private and private extension), C432004 (abstract)			Check that the subtype of an ancestor part of an extension aggregate can be a private type, a private extension, or an abstract type.	These are the odd cases.
								7	Check that the ancestor subtype mark in an extension aggregate can be an interface type.	C-Test. This is a new capability in Ada 2005.
						C432005	All		Check that the ancestor subtype mark in an extension aggregate can be limited.	
				Negative		**B432001**	All		Check that the subtype of an ancestor part of an extension aggregate cannot be a class-wide subtype, nor an untagged type.	
		2	Legality	Subpart	Any ancestor part expression will test.					
				Negative		**B432001**	All		Check that the expression of an ancestor part cannot be dynamically tagged.	New rule in the Amendment.
		3	Legality		This rule was substantially modified by AI05-0115-1.	C432001, C432004			Check that the type of an extension aggregate is a descendant of from the type of the ancestor, through one or more record extensions.	The only change was to replace "derived from" to "a descendant of". That changes the wording of the objective, but only makes a difference in obscure visibility cases (we have a separate objective for them).
				Negative		**B432001**	All		Check that the type of an extension aggregate cannot be unrelated from the type of the ancestor part.	Same as the first objective for this line.
						B432001	All		Check that the type of the ancestor part cannot be the same as or descended from the type of the extension aggregate.	Same as the first objective for this line.
						B432001	All		Check that the type of an extension aggregate cannot be descended from the type of the ancestor through any private extensions.	Same as the first objective for this line.
						B432002	All		Check that the type of an extension aggregate cannot be a type such that the current view of the parent of the aggregate type is not a descendant of the full view of the ancestor type.	
		4	Legality		Added by AI05-0115-1.	***B432003***	All		Check that the subtype_mark of an extension aggregate cannot denote a view of a type that has unknown discriminants.	
	(5.1/5)		Legality		Added by AI05-0067-1 and AI05-0244-1. Turned into the entire rule by AI12-0317-1.			6	Check that the ancestor_part of a limited extension aggregate can have a constituent that is a function call (even if the function has an unconstrained nominal subtype) if the extension part has no needed components.	C-Test. Try parenthesizing, qualifying, and view conversions of the function call. (Complex consistuents have their own objectives.)

					7	Check that the ancestor_part of a limited extension aggregate can have constituents that are objects that have an unconstrained nominal subtype, even if the extension part has components.	C-Test. Try parameters, class-wide objects, and array objects. Try parenthesizing, qualifying, and view conversions of the objects. (Complex constituents have their own objectives.)
		Negative			8	Check that the ancestor_part of a limited extension aggregate cannot be have a constituent that is a function call of a function with an unconstrained result subtype unless the aggregate has no needed extension components.	B-Test. Try parenthesizing, qualifying, and view conversions of the function calls. (Complex constituents have their own objectives.)
					5	Check that the ancestor_part of a limited extension aggregate can be a conditional expression that has a dependent expression that has an unconstrained nominal subtype if the extension part has no needed components.	C-Test.
					5	Check that the ancestor_part of a limited extension aggregate can be a conditional expression where one or more dependent expressions are objects that have an unconstrained nominal subtype, even if the extension part has components.	C-Test.
		Negative			7	Check that the ancestor_part of a limited extension aggregate cannot be a conditional expression that has a dependent expression that is a function call of a function with an unconstrained result subtype unless the aggregate has no needed extension components.	B-Test. Try parenthesizing and qualifying the conditional expression and the dependent expression.
(5.2/5)	Deleted		Added by AI05-0067-1 and AI05-0244-1; folded into main rule by AI12-0317-1.				
(5.3/5)	Deleted		Added by AI05-0067-1 and AI05-0244-1; folded into main rule by AI12-0317-1.				
(5.4/3)	Deleted		Added by AI05-0067-1 and AI05-0244-1; folded into main rule by AI12-0317-1.				
(6)	StaticSem	Subpart	Any extension aggregate will test normal cases.				
					6	Check that inherited discriminants are needed by an extension aggregate if the ancestor subtype mark denotes an unconstrained subtype.	C-Test. (?)
		Negative			6	Check that values cannot be given for components included in the ancestor expression or subtype of an extension aggregate.	B-Test.
		Negative		C432002 (not given).	4	Check that inherited discriminants are not needed by an extension aggregate if the ancestor part is an expression or constrained subtype.	B-Test (try to give them).
			Extra and missing needed components are tested in 4.3.1(16/2).				
(7)	Dynamic			C432003, C432004		Check that the components associated with an ancestor part subtype mark in an extension aggregate are initialized by default.	
				C432001		Check that the components associated with an ancestor part expression in an extension are initialized by the expression.	
			Order of discriminant evaluation tested in 4.3.1(19).				

	(8/3)		Dynamic	This check was broadened by AI05-0282-1.	C432002, C432003			If the type of the ancestor part has discriminants that are not inherited by the type of an extension aggregate, then check that the values of the discriminants are checked and constraint_error raised if needed.	This is the Ada 95 objective.
							5	If the type of the ancestor part has discriminants and the ancestor_part is not an unconstrained subtype name, then check that the values of the discriminants are checks and Constraint_Error is raised if needed.	C-Test. Test cases not covered in the existing tests: discriminants that come from a constrained ancestor. See the example in the AI05-0282-1. Be sure to test the inconsistency mentioned in 4.3.2(13.d/3).
	(9)		NonNormative	A note.					
	(10)		NonNormative	Another note.					
	(11)		NonNormative	Start of examples...					
	(12)		NonNormative						
	(13)		NonNormative	...end of examples.					
4.3.3	(1)	1	Redundant						
		2-3	Definitions	Positional and named array aggregates.					
	(2/5)		Syntax	Changed by AI12-0306-1 to add null_array_aggregate.					
	(3/5)		Syntax	Changed by AI12-0212-1 and AI12-0306-1 to allow square brackets.					
	(3.1/5)		Syntax	Added by AI12-0306-1.					
	(4/5)		Syntax	Changed by AI12-0127-1 to allow sharing with delta aggregates, there no effect on programs. AI12-0212-1 adds square brackets as a possibility.					
	(4.1/5)		Syntax	Changed by AI12-0127-1 to allow sharing with delta aggregates, there no effect on programs.					
	(5/5)		Syntax	Changed by AI12-0061-1.					
				Since the syntax of array and record aggregates has to be shared, this is likely to be a check outside of the syntax.					
					Negative	B43201A		Check that an array aggregate cannot have mixed positional and named notation (excepting **others**).	
						B433001	All	Check that <> is not allowed in positional array aggregates other than in an **others** choice.	
	(5.1/5)		Syntax	Added by AI12-0061-1 and AI12-0212-1, post Corrigendum 1.					
	(6)		Definitions	How n-dimensional array aggregates work.					
	(6.1/5)		Definitions	Added by AI12-0061-1, post Corrigendum 1. “Index Parameter”.					
	(7/2)	1	NameRes	This simply determines the type of the aggregate for reference to other rules; real resolution issues are tested for 4.3(3/2).	Widely Used				
		2	NameRes			C87B31A (one-dim)	2	Check that array component expressions in an array aggregate can be resolved because they must have the component type of the array type of the aggregate.	C-Test: check multi-dimensional arrays.
					Negative	B43201D		Check that array component expressions in an array aggregate must have have the component type of the array type of the aggregate.	

				C433A01 (one-dim), **C433A02** (two-dim)	All		Check that the array component expressions in an array aggregate may have a limited type.	
(8)				C87B31A (one-dim)		2	Check that the choices in a named array aggregate can be resolved because they must have the corresponding index type of the array type of the aggregate.	C-Test: check multi-dimensional arrays.
		Negative		B43201D			Check that the choices in a named array aggregate must have the corresponding index type of the array type of the aggregate.	
(9)	Legality	Subpart	All M-dimensional array aggregates will test.					
		Negative				3	Check that an n-dimensional array aggregate cannot be written as if it has some other dimensionality.	B-Test. Try writing 2-dim arrays as 1-dim positional aggs.
(10)	Legality	Portion	Lead-in for following bullets					
		Negative		B43202C (operators, membership)			Check that an others choice in an array aggregate is not allowed in contexts not described by 4.3.3(11-15).	Are there any other such contexts?? I can't think of any, if there are any I've missed they should be tested.
(11/4)	Legality		The positive objectives really belong to 4.3.3(25), but here we can spread them out more clearly.	C43204A, C433001		2	Check that the constraint of the constrained array subtype of an explicit actual parameter of a subprogram or entry call is used to determine the bounds of an array aggregate with an **others** choice.	C-Test. Try entry calls and others => <>.
				C43204C			Check that the constraint of the constrained array subtype of an explicit actual parameter of an instantiation is used to determine the bounds of an array aggregate with an **others** choice.	
				C433005	All		Check that the constraint of the constrained array subtype of a function return is used to determine the bounds of an array aggregate with an **others** choice in the return expression of a return statement.	
			Added by AI12-0157-1 (although always assumed).	***C433006***	All		Check that the constraint of the constrained array subtype of a function return is used to determine the bounds of an array aggregate with an **others** choice in the return expression of an expression function.	
				C43204E, C433001			Check that the constraint of the constrained array subtype of an object declaration (including constants) is used to determine the bounds of an array aggregate with an **others** choice in the initializing expression of the object.	
				C43204E			Check that the constraint of the constrained array subtype of a component declaration is used to determine the bounds of an array aggregate with an **others** choice in the default expression of the component.	
				C43204F (subprogram), C43204G (entry), C43204H (generic unit)			Check that the constraint of the constrained array subtype of a formal parameter is used to determine the bounds of an array aggregate with an **others** choice in the default expression of the parameter.	
		Negative		B43202A			Check that an others choice is not allowed in an array aggregate that is an explicit actual parameter of a subprogram or entry call if its subtype is unconstrained.	
		Negative		B43202A			Check that an others choice is not allowed in an array aggregate that is an explicit actual parameter of an instantiation if its subtype is unconstrained.	
		Negative		B43202A (simple return, others with value), ***B433003***	All		Check that an others choice is not allowed in an array aggregate in the expression of a return statement if the subtype of the function return is unconstrained.	

			Negative	Added by AI12-0157-1 (although always assumed).	***B433003***	All		Check that an others choice is not allowed in an array aggregate in the return expression of an expression function if the subtype of the function return is unconstrained.		
			Negative		B43202A		3	Check that an others choice is not allowed in an array aggregate in the initializing expression of an object declaration if the subtype of the object is unconstrained.	B-Test. Try a variable declaration and others => <>.	
			Negative				3	Check that an others choice is not allowed in an array aggregate in the default expression of a component declaration if the subtype of the component is unconstrained.	B-Test. Try others => <>.	
			Negative		B43202A			Check that an others choice is not allowed in an array aggregate in the default expression of a formal parameter if the subtype of the parameter is unconstrained.		
(12)		Legality		The positive objectives really belong to 4.3.3(25), but here we can spread them out more clearly.	C43204I, C433001			Check that the constraint of the target array object of an assignment statement is used to determine the bounds of an array aggregate with an **others** choice in the source expression.		
					C433007			Check that an aggregate with an others choice can be assigned to an array variable, even if the variable has an unconstrained nominal subtype.		
			Negative	This is impossible, as an array object must be constrained.						
(13)		Legality		The positive objectives really belong to 4.3.3(25), but here we can spread them out more clearly.	C433001			Check that the constraint of the array subtype of a qualified expression is used to determine the bounds of an array aggregate with an **others** choice in the qualified expression.		
			Negative		B43202A			Check that an others choice is not allowed in an array aggregate in the expression of a qualified expression if the subtype_mark in the qualified expression is unconstrained.		
(14)		Legality					1	Check that the constraint of the component array subtype of an aggregate component is used to determine the bounds of an array aggregate with an **others** choice which is used as a component expression in a larger aggregate.	C-Test. This is not usefully testable, as it is not possible to find the bounds used for the component separate from the array object it is nested in.	
			Negative	This is impossible, as components have to be definite.						
(15/3)		Legality		A punctuation change by AI05-0147-1.	***C433006*** (expression functions)	Part	3	Check that the constraint of the applicable index constraint of a parenthesized expression is used to determine the bounds of a parenthesized array aggregate with an **others** choice.	C-Test. Check parenthesized expressions in all of the other contexts noted under 4.3.3(11-14.1, 15.1).	
			Negative		***B430003*** (return statements, expression functions)	Part	3	Check that an **others** choice in a parenthesized array aggregate is not allowed in contexts where the appropriate subtype (as described by 4.3.3(11-15)) is unconstrained.	B-Test. Check parenthesized expressions in all of the other contexts.	
(15.1/3)		Legality		Added by AI05-0147-1 (Ada 2012).	***C433008*** (object declarations, parameter passing, assignments, expression function, simple return statement)	All		Check that the constraint of the applicable index constraint of a conditional_expression is used to determine the bounds of a dependent array aggregate with an **others** choice.	Could test remaining 4.3.3(11-14.1) contexts, but not a lot of value in doing that.	
			Negative		***B433004*** (return statements, expression functions, qualified expressions, parameter passing, object declarations)	All		Check that an others choice in an array aggregate is not allowed in a conditional expression used in contexts where the appropriate subtype (as described by 4.3.3(11-15.2)) is unconstrained.	Could test remaining 4.3.3(11-14.1) contexts, but not a lot of value in doing that.	
(16)	1	Definitions		Constraint that applies.						

	2	Legality	Negative				6	Check that an others choice is not allowed in an array aggregate that is an explicit actual parameter of a subprogram call to a generic formal subprogram even if its subtype is constrained.	B-Test. This was a fix for a contract model violation in Ada 83, and it should be tested.
							6	Check that an others choice is not allowed in an array aggregate in the default expression of a formal parameter of a generic formal subprogram even if the subtype of the parameter is constrained.	B-Test. This was a fix for a contract model violation in Ada 83, and it should be tested.
(17/5)		Legality		AI05-0153-1 modifies the syntax terms used here, but the effect is unchanged. AI12-0061-1 adds a parenthetical remark. AI12-0127-1 adds rules for delta aggregates.	C43207D (range), C43208A (range), C43208B (range), C43224A (range attribute)		2	Check that a single nonstatic choice is allowed in an array aggregate.	C-Test. Check that the single choice can be an expression, and after the next document (Amendment or Revision) is issued, also check that the choice can be an iterated_component_association.
			Negative				5	Check if an array aggregate contains more than one choice or component association, it is illegal for any (other than a **others** choice) of them to be nonstatic.	B-Test. This is marked as untested in ACATS 2.x.
(18/3)		Legality	Widely Used	Any named notation array aggregate will test this.					
			Negative		B43201C			Check that a named array aggregate cannot contain two choices that cover the same value.	
				New cases added by AI05-0262-1.	B43201C, **B433002**	All		Check that a named array aggregate is illegal if it does not cover a contiguous set of index values.	
(19)		Legality			C43209A			Check that a bottom level subaggregate of an array aggregate can be a string literal.	
							2	Check that a string literal cannot be used as the bottom level subaggregate of an array aggregate if the component type is not a character type.	B-Test. Hard to imagine an implementation getting this wrong.
(20)		StaticSem					4	Check that a string literal used as the bottom level subaggregate of an array aggregate is illegal if any character is not a value of the component type of the aggregate.	B-Test. This is marked as untested in ACATS 2.x. (So is 4.2(7)).
					B43209B			Check that a string literal used as a bottom level subaggregate of an array aggregate cannot be enclosed in parentheses.	
(21)		Dynamic	Portion	Lead-in for following bullets.					
(22)		Dynamic	Not Testable	Arbitrary order is untestable. The conversion could fail, but any case that did would also fail 4.3.3(28), so those tests dominate.					
(23)		Dynamic	Widely Used	Arbitrary order is untestable. Normal evaluation is tested by every array aggregate.					
					C43004A		2	Check that Constraint_Error is raised if a component expression fails the conversion to the component subtype of an array aggregate.	C-Test. Try composite types (discriminant checks, array length checks).
					C43207D (2-dim range), C43208A (1-dim range), C43208B (2-dim range), C43210A (1-dim & 2-dim, named & others)			Check that a component expression in an array aggregate with multiple associated components is evaluated once for each associated component.	
(23.1/4)	1	Dynamic	Widely Used	Any array aggregate					

	2				**C433A01** (task, PO, lim-rec for 1-dim); **C433A02** (task, PO, lim-rec for 2-dim); **C433A04** (non-lim cases); ***C433003*** (scalar types with Default_Value)	All	Check that for each association with a <> in an array aggregate, the component is initialized by default.		
					C433A03	All	Check that for a <> in an array aggregate with multiple associated components, each associated component is default initialized individually.		
				Part added by AI12-0084-1.	***C433004***	All	Check that for each association with a <> in an array aggregate whose type has a Default_Component_Value, the component is initialized to the Default_Component_Value.		
(24)		Dynamic	Portion	Lead-in for following bullets.					
(25)		Dynamic		Tested under 4.3.3(11-15).					
(26)		Dynamic			C43205A (subprogram, unconstrained), C43205G (subprogram, constrained), C43214B (subprogram, constrained, string literal)	3	Check that the constraint (or lack of one) the array subtype of an explicit actual parameter of a subprogram or entry call is used to determine the lower bound of a positional array aggregate or string literal.	C-Test. Try entry calls and string literals in unconstrained contexts.	
					C43205B (unconstrained), C43205H (constrained), C43214C (constrained, string literal)	2	Check that the constraint (or lack of one) of the array subtype of an explicit actual parameter of an instantiation is used to determine the lower bound of a positional array aggregate or string literal.	C-Test. Try string literals in unconstrained contexts.	
					C43205C (unconstrained), C43205I (constrained), C43214D (constrained, string literal)	2	Check that the constraint (or lack of one) of the array subtype of a function return is used to determine the lower bound of a positional array aggregate the expression of a return statement or string literal.	C-Test. Try string literals in unconstrained contexts.	
					C43205D (constant, unconstrained), C43205J (objects, constrained), C43214E (objects, constrained, string literal)	4	Check that the constraint (or lack of one) of the array subtype of an object declaration (including constants) is used to determine the lower bound of a positional array aggregate or string literal in the initializing expression of the object.	C-Test. Try an unconstrained variable, and string literals in any unconstrained contexts.	
						3	Check that the constraint (or lack of one) of the array subtype of a component declaration is used to determine the lower bound of a positional array aggregate or string literal in the default expression of the component.	C-Test.	
					C43205J (subprogram, generic; constrained)	3	Check that the constraint (or lack of one) of the array subtype of a formal parameter is used to determine the lower bound of a positional array aggregate or string literal in the default expression of the parameter.	C-Test. Try entry declarations, unconstrained contexts, string literals.	
						1	Check that the constraint of the target array object of an assignment statement is used to determine the lower bound of a positional array aggregate or string literal in the source expression.	C-Test. Not usefully testable because the bounds slide on the assignment.	
						3	Check that the constraint (or lack of one) of the array subtype of a qualified expression is used to determine the lower bound of a positional array aggregate or string literal in the qualified expression.	C-Test.	
					C43205K, C43214F (string literal), C460010		Check that the constraint of the component array subtype of an aggregate component is used to determine the lower bounds of a positional array aggregate or string literal which is used as a component expression in a larger aggregate.		

				C42007E (string literal), C43205E (string literal)		3	Check that the constraint of the applicable index constraint of a parenthesized expression is used to determine the lower bound of a parenthesized positional array aggregate or string literal.
						3	Check that the lower bound of a positional array aggregate or string literal in a membership is always that of the index subtype, even if the subtype is constrained.
						2	Check that the lower bound of a positional array aggregate or string literal used as the operand of a predefined operator is always that of the index subtype.
(27)	Dynamic			C43206A (null).		3	Check that the bounds of a named array aggregate without others are determined by the choices.
							Check that Constraint_Error is raised if the upper bound of a positional array aggregate without an others choice would be outside of the index subtype.
(28)	Dynamic			C43215A, C43215B			
				C43207B (ranges), C43211A (ranges), C43214A (ranges)		3	Check that Constraint_Error is raised if any non-null choice of a named array aggregate is outside of the index subtype.
							Check that Constraint_Error is raised if any choice of an aggregate with an others clause specifies a component outside of the bounds of the aggregate.
(29/3)	Dynamic			C433001			
			Approved AI05-0037 mandates this.	***C433002***	All		Check that Constraint_Error is raised if any <> choice of an aggregate with an others clause specifies a component outside of the bounds of the aggregate.
							Check that all subaggregates of a multidimensional array aggregate that correspond to the same index have the same bounds.
(30)	Dynamic			C43212A, C43212C			
(31)	Dynamic	Portion	Defines the exception to raise for the previous rules.				
(32/5)	Impl-Def		AI12-0212-1/AI12-0250-1. Do not assume that the iteration is perfromed twice (but it might be). [All of the following paragraph numbers were changed.]				
(33/5)	NonNormative		A note.				
(34/5)	NonNormative		Another note, added by AI12-0061-1.				
(35)	NonNormative		Start of examples...				
(36)	NonNormative						
(37)	NonNormative						
(38)	NonNormative						
(39)	NonNormative						
(40)	NonNormative						
(41)	NonNormative						
(42)	NonNormative						
(43)	NonNormative						
(44)	NonNormative						
(45)	NonNormative						
(46/5)	NonNormative		A new example added by AI12-0061-1.				
(47/5)	NonNormative		A new example added by AI12-0312-1.				
(48/2)	NonNormative						
(49/5)	NonNormative		End of examples. Corrected by AI12-0178-1.				

	C-Test.
	C-Test.
	C-Test. Try aggregates.
	C-Test. Try non-null aggregates.
	C-Test. Try single choices.

4.4	(1/3)	General		Modified by AI05-0147-1.					
	(2)	Syntax							
	(2.1/3)	Syntax		Added by AI05-0158-1. Note that the important effects of these changes are tested in 3.8 (for choices) and 4.5.2 (for memberships)					
	(2.2/3)	Syntax		Added by AI05-0158-1.					
	(3/4)	Syntax		Added by AI05-0158-1 and AI12-0022-1; corrected by AI12-0039-1.					
			Negative		B45205A		Check that expressions of the form A < B < C aren't allowed.	We only record this because of the existing test.	
	(3.1/3)	Syntax		Added by AI05-0158-1.					
	(3.2/4)	Syntax		Added by AI05-0158-1; corrected by AI12-0039-1.	***B440001***	All	Check that the operands of a membership test are simple_expressions, not choice_expressions.	We check this syntax change because it is an incompatibility with the original definition of Ada 2012.	
	(4)	Syntax							
	(5)	Syntax							
	(6)	Syntax							
	(7/5)	Syntax		Various syntax additions over the years, not interesting since we don't test syntax.					
				We test precedence separately, because it is possible for the syntax to be flattened.	C44003D (float), C44003F (enum), C44003G (Boolean)		2 Check that the precedence of operators is correct.	C-Test. Try integer, modular, and ordinary and decimal fixed point types. But not likely to be wrong.	
	(8)	NameRes					3 Check that a primary can be resolved because it must denote an object or value.	C-Test. Try a function overloaded with a procedure; use in an expression must resolve to the function call (even without parameters).	
			Negative		B44001B (procedure), B44002B (tasks, entries), B44002C (exception)		2 Check that names that do not denote an object or value are not permitted as primaries.	B-Test. Try type and subtype names, package names, single protected object names, block and loop names. But not likely to be wrong.	
	(9)	Definitions		The type of an expression.					
	(9.1/5)	Definitions		“Parenthesized expression”, added by AI12-0317-1.					
	(9.2/5)	Definitions		“constituent”, “evaluated constituent”, added by AI12-0317-1. Tested on uses, 4.3.2, 6.2, esp. 7.5.					
	(9.3/5)	Definitions	Portion	Part of above, added by AI12-0317-1.					
	(9.4/5)	Definitions	Portion	Part of above, added by AI12-0317-1.					
	(9.5/5)	Definitions	Portion	Part of above, added by AI12-0317-1.					
	(9.6/5)	Definitions	Portion	Part of above, added by AI12-0317-1.					
	(9.7/5)	Definitions		“newly constructed”, added by AI12-0317-1. Tested when used, see 7.5.					
	(10)	Dynamic	Widely Used	Any object name used in an expression tests this.					
	(11)	Impl-Def	Not Testable	Either something happens, or it doesn't. Can't test that.					
	(12)	NonNormative		Start of examples...					
	(13)	NonNormative							
	(14)	NonNormative							
	(15/2)	NonNormative		...end of examples.					

4.5	(1)		General					
	(2)		Syntax					
	(3)		Syntax					
	(4)		Syntax					
	(5)		Syntax					
	(6)		Syntax					
	(7)		Syntax					
	(8)	1	Redundant	We test precedence under 4.4(7/3).				
		2	Widely Used	Almost any use of parens will test.				
	(9)	1	Definitions	Definition of “predefined”.				
		2	StaticSem		C45251A (comparisons) – also several others, C45331A (adding), C45503A (mod/rem), C45631A (abs), C45672A (not)	2	Check that the parameters of a predefined operator are named Left and Right.	C-Tests for multiply, divide, exponentiation, and concat. Not very important, unlikely to get wrong.
		3	Redundant	Normatively in 6.6(2); judged not necessary to test there.				
		4	Redundant	Also in 6.6(2).				
		5	General					
	(10)	1	Redundant	Normatively in 3.5.4(20), tested for individual operators.				
		2	Redundant	Normatively in Annex G.				
	(11)		Impl-Req	Not Testable	One can't guess when an implementation would do the wrong thing; also widely used so that a mistake would be unlikely to be missed.			
	(12)		Impl-Req	Not Testable	One can't guess when an implementation would do the wrong thing; also widely used so that a mistake would be unlikely to be missed.			
	(13)		Impl-Perm	Not Testable	Tests need to avoid cases that could be changed by the permission.			
	(14)		NonNormative		A note.			
	(15)		NonNormative		An example.			
	(16)		NonNormative		An example.			
	(17)		NonNormative		An example.			
4.5.1	(1)		NameRes			B45121A		Check that the operands of a short-circuit form cannot have a non-Boolean type.
						C87B33A		Check that overloading resolution uses the rules that a short-circuit control form and its operands are boolean expressions.
	(2)		StaticSem	Portion	Lead-in for following definition			
	(3)		StaticSem	Widely Used	Just defines the operators.			

				Negative		B45102A (record, integer, character, array of integer), B45116A (multidim arrays, non Boolean arrays)		Check that the operands of the logical operands cannot be any type other than a boolean, modular, or array of boolean type.	We could try additional types, but that would be of little added value.
(4)		StaticSem	Widely Used	We could check that the logical operators produce the correct results, but if they didn't, many ACATS tests would fail.					
(5)		StaticSem				C450001		Check that the modular logical operators do a bit-by-bit operation, and subtract the modulus if necessary.	
(6)	1	StaticSem				C45114B (packed)		Check that the array logical operators do a component-by-component operation.	
	2	StaticSem				C45112A, C45112B		Check the lower bound of the result of an array logical operator is that of the left operand.	
(7)		Dynamic	Widely Used	We could check that the short circuit forms produce the correct results and don't evaluate the right operand, but if they didn't, many ACATS tests would fail.					
(8)	1	Dynamic				C45113A		Check that Constraint_Error is raised when the array operands of a logical array operator are different lengths.	
	2	Dynamic					2	Check that Constraint_Error is raised if the result of a logical array operator has a component outside of the component subtype.	C-Test. This is very unlikely, so its barely worth testing.
	3	Dynamic	Portion						
(9)		NonNormative		A note.					
(10)		NonNormative		A note.					
(11)		NonNormative		An example.					
(12)		NonNormative		An example.					
(13)		NonNormative		An example.					
(14)		NonNormative		An example.					
4.5.2 (1)	1	Definitions		“equality operators”					
	2	Definitions		“ordering operators”					
	3	Definitions		“discrete array types”					
(2/3)	1	Definitions		“membership test”					
	2	Redundant							
(3/3)	1	Definitions		“tested type”					
	2	NameRes	Widely Used	Any membership will test.					
						C452002 (enumerations)	Part	4 Check that the requirement that membership choices are all the same type can be used to resolve a membership test.	C-Test. Try overloaded composite functions.
			Negative					7 Check that multiple membership choices cannot resolve to different composite types.	B-Test. Try objects of related types (T and T'Class, for instance), or one object and one type.
								7 Check that multiple membership choices cannot resolve to unrelated elementary types.	B-Test. Try objects and functions of types related by derivation.
(3.1/4)	1	NameRes		AI12-0039-1 introduced “tested_simple_expression”.				7 For a tagged tested type, check that the tested expression can be a type that is not the tested type but is convertible to it.	C-Test. Probably ought to test interface cases.

2		Widely Used	All legal memberships will test usual cases. Note: All resolution rules are applied at once, thus these rules can be recursive.			5	For an untagged tested type, check that the expression can be resolved based on the expected type being the tested type.	C-Test. Try a call on an overloaded function for the tested expression. This is normal resolution, so middle priority.
				C452002 (enumerations)	Part	6	For an untagged type, check that the tested expression can determine the tested type.	C-Test. The tested expression could be something not overloaded, while the choices contain overloading. This objective is for Ada 2012 cases with object choices or multiple ranges.
			This is the Ada 83 case; it's not a very interesting test but many existing tests fall here.	B45209A (enum), B45209B (signed int), B45209C (enum), B45209D (char enum), B45209E (Boolean), B45209F (ordinary fixed), B45209G (float), B45209H (array), B45204I (record), B45204J (access), B45204K (private)			For a membership test with a single choice and an untagged tested type, check that the tested expression and the type of the subtype or range bounds is the same.	We could add the missing cases: modular, task, and protected, but this series has fairly minimal value to begin with.
						3	For an tagged type, check that the tested expression can determine the tested type.	C-Test. The tested expression could be something not overloaded, while the choices contain overloading. Note that a realistic test is hard to construct here, since the types would have to be completely unrelated. Thus the low priority.
						8	Check that a membership with a range choice can be resolved even when the tested expression and both range choices are overloaded.	C-Test – This is the Ada 83 test case.
(4/4)	Legality		AI12-0039-1 introduced “tested_simple_expression”.			7	Check that if the tested expression is a tagged class-wide type, the tested type has to be visibly tagged.	B-Test. See the AARM note for how such types can exist. The test would have to be in a body that can see the full definition of the private type as well as the derived type (possibly a child?).
(4.1/5)	Legality		AI12-0039-1 introduced “tested_simple_expression”.	***B452001***	All		Check that if the tested expression has a limited type and the membership has any choice expressions, the tested type must have a visible equality operator.	
(5)	StaticSem	Widely Used	Any legal membership test will check.					
		Negative		B45204A (local boolean-alike), B45221A (derived Boolean)		6	Check that a membership test cannot be used without conversion with boolean types other than Boolean.	B-Test.
(6)	StaticSem	Widely Used	Any legal equality will check.					
		Negative		B45207A, B45207G		5	Check that no equality operators are predefined for a limited private type.	B-Test. Still to try a generic formal limited private type.
				B45207B, B45207C, B45207D, B45207H, B45207I, B45207J, B45207N, B45207O, B45207P, B45207T, B45207U, B45207V			Check that no equality operators are predefined for a composite type with a component of a limited type.	
						7	Check that no equality operators are predefined for a limited record type.	B-Test.

						7	Check that no equality operators are predefined for a limited interface type.
					B45207M, B45207S		Check that no equality operators are predefined for a task type.
						6	Check that no equality operators are predefined for a protected type.
						4	Check that no equality operators are predefined for an anonymous access type.
(7)	StaticSem	Widely Used	The equality operators are widely used.				
					C45232B (signed integer), C45242B (float), C45252B (ordinary fixed)	7	Check that the operands of a predefined equality operator have subtype T'Base.
						5	Check that the parameter names for a predefined equality or inequality function are “Left” and “Right”.
					B45221A (derived Boolean)	4	Check that the result of predefined equality operators is always Boolean (and not some other boolean type).
					B45206C		Check that the operands of a predefined equality operator are of the same type.
(7.1/2)	StaticSem					7	Check that the equality operators for anonymous access types exist and can be used.
		Negative				7	Check that the access equality operators in Standard cannot be used with named access types.
(7.2/2)	StaticSem	Subpart	Mostly tested above.			5	Check that the parameter names for the access equality and inequality functions declared in Standard are “Left” and “Right”.
(8)	StaticSem					5	Check that ordering operators are predefined for enumeration types.
					C45231A, C45232B		Check that ordering operators are predefined for signed integer types.
					C450001		Check that ordering operators are predefined for modular types.
					C45242B		Check that ordering operators are predefined for float types.
						5	Check that ordering operators are predefined for fixed types.
						5	Check that ordering operators are predefined for decimal fixed types.
						5	Check that ordering operators are predefined for discrete array types.
					C45232B (signed integer), C45242B (float), C45252B (ordinary fixed)	6	Check that the operands of a predefined ordering operator have subtype T'Base.
		Negative			B45208C, B45208I	5	Check that ordering operators are not predefined for access types.
					B45208C, B45208I		Check that ordering operators are not predefined for record types.
					B45208B, B45208H, B45208N, B45208T, B45261B, B45261C, B45261D		Check that ordering operators are not predefined for non-discrete array types.

B-Test.
B-Test.
B-Test. Careful about the universal operator! Must check that no operators allow selected notation at the point of the type declaration. (This is unlikely in practice, thus the low priority.)
C-Test. Still need to try modular, decimal fixed type. Note that this is not usefully testable for enumeration types.
C-Test(?)
B-Test. Note: Only need a few examples.
We have this objective mainly to cover the existing test.
C-Test. Possibly combine with other tests for this feature.
B-Test (?)
C-Test(?)
C-Test.
C-Test.
C-Test.
C-Test.
C-Test.
C-Test. Still need to try modular, decimal fixed type. Note that this is not usefully testable for enumeration types.
B-Test. Still to try: generic formal access types.

						B45261A		Check that ordering operators are not predefined for multi-dimensional arrays.	
						B45208M, B45208S		Check that ordering operators are not predefined for task types.	
							5	Check that ordering operators are not predefined for protected types.	B-Test.
						B45208A (limited private), B45208G (derived lp)	5	Check that ordering operators are not predefined for private types.	B-Test. Still to try both non-limited private types, as well as formal private types (both kinds).
						B45221A (derived Boolean)	4	Check that the result of predefined ordering operators is always Boolean (and not some other boolean type).	B-Test. Note: Only need a few examples.
						B45206C		Check that the operands of a predefined ordering operator are of the same type.	We have this objective mainly to cover the existing test.
(9)		StaticSem	Subpart	Tested with previous paragraph.					
(9.1/2)	1	NameRes			***B452002***	All		Check that a universal access equality operator does not resolve if neither operand is a specific anonymous access type.	
	2						7	Check that the universal access equality is not used if either operand is designated a type D such that D has a primitive equality with one or more access parameters designating D.	B-Test. Possibly define a primitive equality with only one access parameter, so that compare of an access against null cannot resolve.
							7	Check that even if a type D has a primitive equality with one or more access parameters designating D, the universal access equality can be used with the Standard prefix.	C-Test.
							8	Check that for a type D with primitive equality with two access-to-D parameters, the primitive equality is used when comparing anonymous access-to-D against null.	C-Test. This is the point of these rules, to make it possible for such operators to be used as intended. Check a case where the equality is declared in the spec for a private type and the usages are in the body (see AI05-0020-1).
							6	Check that the universal access equality is used if either operand is designated a type D such that D has a primitive equality with one or more access parameters designating D but that returns a non-Boolean type.	C-Test.
(9.2/2)			Subpart	Tested with 9.1/2.					
(9.3/3)			Subpart	Tested with 9.1/2.					
(9.4/2)			Subpart	Tested with 9.1/2.					
(9.5/2)		Legality			***B452002***	All		Check that a universal access equality operator is illegal if one of the operands is access-to-object and the other is access-to-subprogram.	
					C3A0025 (access-to-object, one use in Use_it); ***C3A0026*** (access-to-subprogram, one use in Use_It).	Part	4	Check that one of the operands of a universal access equality operator can be the literal null.	C-Test. This is the motivating usage, so it is important to test. Would like a more through test, but the existing tests do provide an existence check.
(9.6/2)		Legality			***B452002***	All		Check that a universal access equality operator is illegal when both operands have access-to-object types and the designated subtypes of the operands are elementary and do not statically match.	
					B452002	All		Check that a universal access equality operator is illegal when both operands have access-to-object types and the designated subtypes of the operands are array types and do not statically match.	

						B452002	Part	6	Check that a universal access equality operator is illegal when both operands have access-to-object types and the designated types of the operands are record types and one does not cover the other.	B-Test. Probably should try more class-wide cases, especially involving interfaces or abstract root types.
						B452002	All		Check that a universal access equality operator is illegal when both operands have access-to-object types and the designated subtypes of the operands are different kinds of types.	
(9.7/2)		Legality				***B452002***	All		Check that a universal access equality operator is illegal when both operands have access-to-subprogram types and the designated profiles of the operand types are not subtype conformant.	
(9.8/2)	1	Legality	Widely Used	Any normal definition of an equality operator.						
			Negative	Note: We don't need a similar rule for tagged types since all primitives are banned after freezing for tagged types.				9	Check that it is illegal to declare a type-conformant equality operator for an untagged record type after the type is frozen.	B-Test.
	2			Note: The original second sentence was deleted by AI12-0101-1 (from the Ada 2012 corrigendum).				7	Check that an instantiation is illegal if it declares a type-conformant equality operator for a formal private type after the type is frozen, and the actual type is an untagged record type.	B-Test.
(10)		Dynamic				C45201A (equal, enum), C45201B (ordering, enum), C45210A (ordering, enum), C45220A (equal, Boolean), C45220B (ordering, Boolean)			Check that the relational operators for enumeration types work as expected.	
						CC1221A (equal, generic integer)		8	Check that the relational operators for signed integer types work as expected.	C-Test. Include generic cases.
						C450001		4	Check that the relational operators for modular types work as expected.	C-Test. Still need generic formal modular type test.
						CC1220A		7	Check that the relational operators for generic discrete types work as expected.	C-Test.
(11)		Dynamic				CC1222A (equal, generic float)		8	Check that the relational operators for float types work as expected.	C-Test. Include generic cases.
						C45251A (normal type), CC1223A (equal, generic fixed)		4	Check that the relational operators for ordinary fixed point types work as expected.	C-Test. Still need ordering operators for generic formal fixed types.
								6	Check that the relational operators for decimal fixed point types work as expected.	C-Test. Include generic cases.
(12)		Dynamic				C45281A (pool-specific)		5	Check that equality of access-to-object types works as expected.	C-Test. Still to try: general access, anonymous access, access-to-constant; don't forget null values.
(13)		Dynamic						7	Check that equality of access-to-subprogram types work as expected.	C-Test. Use all kinds of access types (named and anonymous); don't forget null values.
(14/3)		Dynamic				C340001			Check that primitive, rather than predefined, equality of the parent type is used when implementing predefined equality of a type extension.	
				Untagged case added by AI05-0123-1.		C452001			Check that primitive, rather than predefined, equality is used for extension components of a record type when implementing predefined equality of a type extension.	Note: This Ada 95 test was extensively modified to check Ada 2012 rules.

				C452001 (array)	6	Check that predefined, rather than primitive, equality is used for extension components of a non-record type when implementing predefined equality of a type extension.	C-Test. Still need to check discrete, float, fixed, and access types, but user-defined "=" is not very common for any of these.
						Check that the primitive equality of the parent type and the appropriate equality of all of the extension components participate in the result of predefined equality of a type extension.	
				C452001			
(15/5)	Dynamic			C452001		Check that for an untagged private type, if the full type is a tagged record type, the primitive equality of the full type is used to implement the predefined equality of the partial view.	
						Check that for an untagged private type, if the full type is an untagged record type, the primitive equality of the full type is used to implement the predefined equality of the partial view.	
			Case added by AI05-0123-1.	C452001			
			Case added by AI12-0328-1 (although implictly always there).		5	Check that for an untagged private type, if the full type is a tagged record extension, the primitive equality of the full type is used to implement the predefined equality of the partial view.	C-Test. Try type extensions; could base on C452001. It seems not very likely that this would be wrong.
					8	Check that for an untagged private type, if the full type is an access type, the predefined equality of the full type is used to implement the predefined equality of the partial view.	C-Test. Try several different access types (general, pool-specific, etc.), and have a (different) primitive equality for some of them.
				C452001		Check that for an untagged private type, if the full type is an array type, the predefined equality of the full type is used to implement the predefined equality of the partial view.	
					8	Check that for an untagged private type, if the full type is a scalar type, the predefined equality of the full type is used to implement the predefined equality of the partial view.	C-Test. Try several different scalar types (float, integer, enumeration, etc.), and have a (different) primitive equality for some of them.
(16)	Dynamic	Portion	Lead-in for following bullets; also defines "matching components"				
(17)	Dynamic	Portion	Tested whether record equality is tested; doesn't need a separate test (as the order of the components has to be the same for all values of a type anyway).				
(18)	Dynamic			C45264A (single case)	6	Check that the bounds of one-dimensional array objects do not participate in predefined equality for the array type.	C-Test. Should be more than one test case!
(19)	Dynamic		If the bounds of the objects differ, by definition there are unmatched components and the result is False.		8	Check that the bounds of multidimensional array objects do participate in predefined equality for the array type unless the objects are null.	C-Test. Check that objects of different subtypes with different bounds by the same lengths compare unequal.
(20)	Dynamic	Subpart	This rule is used when "matching components" is used to define array and record conversions. We'll test it there (in 4.6).				
(21)	Dynamic	Portion	Lead-in for following bullets				
(22)	Dynamic				7	Check that predefined equality compares two one-dimensional null arrays to be equal, regardless of their bounds.	C-Test. Note: C45264A's objective says it tests this, but it doesn't.
				C45264A		Check that predefined equality compares two multi-dimensional null arrays to be equal, regardless of their bounds.	
					7	Check that predefined equality compares two null record objects to be equal.	C-Test.

(23)		Dynamic		We checked multi-dimensional arrays earlier. Is there any other way for there to be unmatched components? I don't think so.	C45264B, C45264C		Check that predefined equality for one-dimensional array types compares array objects of different lengths to be unequal.	
(24/3)		Dynamic			C452001 (untagged – type Star_Data)	8	Check that primitive, rather than predefined, equality is used for components of a record type when implementing predefined equality of an enclosing record type.	C-Test. Need a tagged record case (only have extensions in the existing test).
					C452001 (array – type Mission)	8	Check that predefined, rather than primitive, equality is used for components of a non-record type when implementing predefined equality of an enclosing record type.	C-Test. Still need to check discrete, float, fixed, and access types.
					C452001 (both tagged and untagged).		Check that primitive, rather than predefined, equality is used when implementing predefined equality for an array type if the component type is a record type.	
						9	Check that predefined, rather than primitive, equality is used when implementing predefined equality for an array type if the component type is a non-record type.	C-Test. Check discrete, float, access, and array types.
					C45264B (one-dim array), C45271A (immutable untagged record), C45272A (mutable untagged record), C45273A (untagged record)			
						7	For composite types, check that all components participate in predefined equality.	C-Test. Still to check array types and tagged record types.
(24.1/5)		Dynamic		Altered by AI12-0413-1 to support instance case.		8	Check that if a predefined equality includes a call on an abstract primitive equality for some (untagged record) component, Program_Error is raised if that equality would need to be called to determine the result.	C-Test. Try in an array type, record type, and tagged type extension.
						8	Check that if a subprogram in a generic body calls equality for a generic formal private type, and the actual type is an untagged record type with an abstract equality, Program_Error is raised by the equality call.	C-Test.
						5	Check that if a subprogram in a generic body calls equality for a generic formal derived type with a normal equality, and the actual type is an untagged record type with an abstract equality, Program_Error is raised by the equality call.	C-Test.
(24.2/1)		Dynamic	Not Testable	The order of evaluation is arbitrary, moreover, we can't even tell whether any user-defined “=” will be called if the result is False. C-Tests for predefined equality need to be aware of this rule and avoid testing cases that require a specific order.				
(25)		Dynamic		If equality is overridden for a type, 6.6 causes a matching inequality to be created. That is tested in that clause (not here).	C45273A (untagged record), C392013 (tagged record)	4	Check that predefined inequality gives the complementary result to predefined equality when no overriding equality is defined.	C-Test. Still to try: discrete, float, fixed, access, and array types.
(26/3)	1	Definitions		“lexographic ordering”	C45262A (array of integer), C45262B (string), C45262C (array of enum)		Check that a null array is less than any non-null array for the ordering operators of discrete array types.	
	2				C45262A (array of integer), C45262B (string), C45262C (array of enum)		Check that a non-null array value V is less than another array W if the first component of V is less than the first component of W.	

				C45262A (array of integer), C45262B (string), C45262C (array of enum)			Check that a non-null array value V is less than another array W if the first component of V equals the first component of W, and the tail of V is less than the tail of W. Especially check cases where V is shorter than W.	
(26.1/3)	Definitions		“individual membership test”	C45262D		3	Check that user-defined operators are not used to implement lexographic ordering of discrete arrays.	Note: A better test should be created using a separate integer type with overridden operators; this would correspond more directly to actual usage. We would **replace** the existing test in that case; the existing test does not have primitive operators.
(27/4)	Dynamic	Not Testable	The order of evaluation is arbitrary, can't test “arbitrary”. AI12-0039-1 introduced “tested_simple_expression”.					
(27.1/4)	Dynamic		AI12-0039-1 introduced “tested_simple_expression”.	***C452004***	All		Check that for a membership test with multiple choices, the tested expression is evaluated first, and then the choices are evaluated in order.	
(28/3)	Dynamic	Portion	Lead-in for following bullets	***C452004***	All		Check that for a membership test with multiple choices, choices after the first individual membership test that evaluates True are not evaluated.	
(28.1/5)	Dynamic		AI12-0328-1 clarified limited rules.	***C452004*** (private record)	Part	4	Check that an individual membership test that is an expression of an untagged record yields True if the primitive equals for the type yields True, and False otherwise.	C-Test. Could try a non-private type, but that is a rarer case.
						8	Check that an individual membership test that is an expression of a tagged record yields True if the primitive equals for the type yields True, and False otherwise.	C-Test.
				C452005 (limited array), ***C452006*** (limited private, completed by an access type)	Part	5	Check that an individual membership test that is an expression of a limited type yields True if the primitive equals for the type yields True, and False otherwise.	C-Test. Still to try limited record types and protected types, all with defined equality operators.
				C452005 (nonlimited array)	All		Check that an individual membership test that is an expression of an array type yields True if the predefined equals for the type yields True, and False otherwise.	
				C452002 (enumerations)	Part	8	Check that an individual membership test that is an expression of a scalar type yields True if the predefined equals for the type yields True, and False otherwise.	C-Test. Still must try several different scalar types (float, integer, etc.), and have a (different) primitive equality for some of them.
				C452006 (private, pool-specific, with equality)	Part	4	Check that an individual membership test that is an expression of an access type yields True if the predefined equals for the type yields True, and False otherwise.	C-Test. Try several different access types (general, pool-specific, etc.), and have a (different) primitive equality for some of them.
(28.2/4)	Dynamic		C450001 (Modular)	C45202B (enumeration), C45211A (character enum), C45220E (Boolean), C45231A (Integer), ***C452002*** (enumerations)	Part	8	Check that an individual membership test that is a range yields True if the tested expression belongs to the range, and False otherwise.	C-Test. Still must try various scalar types: Character, signed integer, float, ordinary fixed, decimal fixed.
(29/4)	Dynamic		AI12-0039-1 introduced “tested_simple_expression”.	C45202B (enumeration), C45211A (char enum), C45220E (Boolean), C45231A (Integer), C45253A (ordinary fixed)		8	Check that an individual membership test that is a scalar subtype yields True if the tested expression belongs to the range of the subtype and the subtype does not have any predicates, and False otherwise.	C-Test. Still need to try various scalar types: signed integer, modular, float, ordinary fixed, decimal fixed.

				C324001 (enumeration), ***C324004*** (enumeration, integer)	7 Check that an individual membership test that is a scalar subtype yields True if the tested expression belongs to the range of the subtype and the predicates of the subtype are satisfied, and False otherwise.	C-Test. Still try various scalar types: Character, modular, float, ordinary fixed, decimal fixed.
(30.4)	Dynamic	AI12-0039-1 introduced "tested_simple_expression".	C45265A		Check that an individual membership test that is a constrained array subtype with no predicates yields True if the tested expression satisfies the constraint of the subtype, and False otherwise.	
				C45274C (only True cases)	8 Check that an individual membership test that is a constrained discriminated record subtype with no predicates yields True if the tested expression satisfies the constraint of the subtype, and False otherwise.	C-Test.
					8 Check that an individual membership test that is an array subtype with predicates yields True if the tested expression satisfies the predicates of the subtype, and False otherwise.	C-Test.
				C324001 (private), ***C324003*** (private), ***C324004*** (private), ***C324005*** (private)	6 Check that an individual membership test that is a record subtype with predicates yields True if the tested expression satisfies the predicates of the subtype, and False otherwise.	C-Test. Still should try tagged record, limited record, and possibly untagged record that's not private.
				C45282A (access-to-array), C45282B (access-to-record, private)	5 Check that an individual membership test that is a constrained access subtype with no predicates yields True if the tested expression satisfies the constraint of the subtype, and False otherwise.	C-Test. Still to try: access-to-discriminated-task, access-to-discriminated-protected
					7 Check that an individual membership test that is an access subtype with predicates yields True if the tested expression satisfies the predicates of the subtype, and False otherwise.	C-Test.
					7 Check that an individual membership test that is a constrained discriminated task subtype with no predicates yields True if the tested expression satisfies the constraint of the subtype, and False otherwise.	C-Test.
					7 Check that an individual membership test that is a task subtype with predicates yields True if the tested expression satisfies the predicates of the subtype, and False otherwise.	C-Test.
					7 Check that an individual membership test that is a constrained discriminated protected subtype with no predicates yields True if the tested expression satisfies the constraint of the subtype, and False otherwise.	C-Test.
					7 Check that an individual membership test that is a protected subtype with predicates yields True if the tested expression satisfies the predicates of the subtype, and False otherwise.	C-Test.
		"untagged composite subtype" corresponds to excluding tagged types and access types (along with scalar types, not covered by this paragraph).	C45265A (array), C45274A (record, private), C45274B (record, private with discrims), C45291A (task, limited private, array, private)		4 Check that an individual membership test that is an unconstrained untagged composite subtype without predicates yields True.	C-Test. Still to try: protected.
			C45282A (access-to-scalar, access-to-private), C45282B (access-to-task)		Check that an individual membership test that is an unconstrained pool-specific access subtype without predicates or exclusions yields True.	
(30.1/4)	Dynamic	AI12-0039-1 introduced "tested_simple_expression".			10 Check that if the tested expression has a class-wide type, and the individual membership test is a subtype, the test yields True if the expression's tag is covered by the subtype, and False otherwise.	C-Test. Ought to try interfaces (the motivating case), as well as more usual hierarchies of tagged types. Could borrow the hierarchy from B452002 or possibly one of the C3A tests.

(30.2/4)	Dynamic		AI12-0039-1 introduced “tested_simple_expression”.	***C452003*** (general access, anonymous tested expr)	Part	6	Check that if the tested type is an access type, and the individual membership test is a subtype that excludes null, the test yields True if the expression is not null, and False otherwise.	C-Test. Still ought to try on pool-specific access types, and on regular access objects.
(30.3/4)	Dynamic		AI12-0039-1 introduced “tested_simple_expression”.	***C452003*** (a-2-var to a-2-cnst)	Part	7	Check that if the tested type is a general access-to-object type, and the individual membership test is a subtype, the test yields True if the expression is convertible to the type of the subtype, and False otherwise.	C-Test. Use library-level accessibility for this test so it doesn't get involved. Test values of various anonymous access types as the tested expression, since that is the interesting case. Still to try: access-to-untagged record/array with misatching constraints.
				C452003 (access parameters, SAOAATs)	Part	4	Check that if the tested type is a general access-to-object type, and the individual membership test is a subtype, the test yields True if the accessibility of the expression is no deeper than that of the tested type, and False otherwise.	C-Test. Test values of various anonymous access types as the tested expression, since that is the interesting case. Still need to try anonymous access components and discriminants.
(31/3)	Dynamic	Subpart	This last part corresponds to the dynamic check made by subtype conversions. The point of this rule is that if the membership test returns True, it should be possible to safely convert (without exceptions!) the tested expression to the tested type. This is tested as part of the preceding objectives.	***C452003*** (one level)	Part	5	Check that if the tested type is a general access-to-object type designating a tagged type, and the individual membership test is a subtype, the test yields True if the tag of the object designated by the expression is covered by the tag of the type designated by the subtype, and False otherwise.	C-Test. Still need to try a more complex tagged type hierarchy, and access-to-specific types on both sides.
(32)	Dynamic		Many types should be tested here. Some don't seem usefully testable: Container Reference_Type and Constant_Reference_Type are not intended to exist for long, so a usage-oriented test couldn't exist and thus we don't test those.	C45274A, C45282A, C45282B, **C324001**, ***C452002***			Check that a membership test using not in gives the complementary result to that using in.	Note: Most membership tests also try “not in”, we just listed a few.
(32.1/1)	Impl-Req		The objectives cover all other nonlimited untagged language-defined private types.	***C452A01***	All		Check that “=” operator of type System.Addess is used in the equality for a record type containing a component of Address.	
				C452A02	All		Check that “=” operator of type Ada.String.Maps.Character_Set is used in the equality for a record type containing a component of Character_Set.	
				C452A02	All		Check that “=” operator of type Ada.String.Wide_Maps.Wide_Character_Set is used in the equality for a record type containing a component of Wide_Character_Set.	
				C452A02	All		Check that “=” operator of type Ada.String.Wide_Wide_Maps.Wide_Wide_Character_Set is used in the equality for a record type containing a component of Wide_Wide_Character_Set.	
				C452A02	All		Check that “=” operator of type Bounded_String (from an instance Ada.String.Bounded) is used in the equality for a record type containing a component of Bounded_String.	
				C452A02	All		Check that “=” operator of type Ada.Strings.Unbounded.Unbounded_String is used in the equality for a record type containing a component of Unbounded_String.	
				C452A03	All		Check that “=” operator of type Ada.Task_Identification.Task_Id is used in the equality for a record type containing a component of Task_Id.	

			Check that “=” operator of type Cursor (from an instance of an Ada.Containers) is used in the equality for a record type containing a component of Cursor.	C-Test. We need one test per container type, but only one each for regular, indefinite, and bounded containers is high priority. Use foundation F452A00, and C452A01 as a pattern.
8				
			Check that “=” operator of types Reference and Constant_Reference (from an instance of an Ada.Containers) is used in the equality for a record type containing a component of that type.	C-Test. This is very unlikely to occur in practice, so a test is rather pathological. If we do make a test, we need one test per container type, but only one each for regular, indefinite, and bounded containers is high priority. Use foundation F452A00, and C452A01 as a pattern.
2				
	C452A02	All	Check that “=” operator of type Wide_Bounded_String (from an instance Ada.Strings.Wide_Bounded) is used in the equality for a record type containing a component of Wide_Bounded_String.	
	C452A02	All	Check that “=” operator of type Ada.Strings.Wide_Unbounded.Wide_Unbounded_String is used in the equality for a record type containing a component of Wide_Unbounded_String.	
	C452A02	All	Check that “=” operator of type Wide_Wide_Bounded_String (from an instance Ada.Strings.Wide_Wide_Bounded) is used in the equality for a record type containing a component of Wide_Wide_Bounded_String.	
	C452A02	All	Check that “=” operator of type Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Unbounded_String is used in the equality for a record type containing a component of Wide_Wide_Unbounded_String.	
			Check that “=” operator of type State (from an instance Ada.Numerics.Discrete_Random) is used in the equality for a record type containing a component of State.	C-Test. Use foundation F452A00, and C452A01 as a pattern.
6				
			Check that “=” operator of type Ada.Numerics.Float_Random.State is used in the equality for a record type containing a component of State.	C-Test. Use foundation F452A00, and C452A01 as a pattern.
6				
			Check that “=” operator of type Ada.Real_Time.Time_Span used in the equality for a record type containing a component of Time_Span.	C-Test. Note: This is an Annex D test. Use foundation F452A00, and C452A01 as a pattern.
6				
			Check that “=” operator of type Ada.Real_Time.Time is used in the equality for a record type containing a component of Time.	C-Test. Note: This is an Annex D test. Use foundation F452A00, and C452A01 as a pattern.
6				
			Check that “=” operator of type Imaginary (from an instance Ada.Numerics.Generic_Complex_Types) is used in the equality for a record type containing a component of Imaginary.	C-Test. Note: This is an Annex G test. Use foundation F452A00, and C452A01 as a pattern.
5				
	C452A01	All	Check that “=” operator of type Ada.Calendar.Time is used in the equality for a record type containing a component of Time.	
	C452A01	All	Check that “=” operator of type Ada.Exceptions.Exception_Id is used in the equality for a record type containing a component of Exception_Id.	
			Check that “=” operator of type Interfaces.C.Strings.chars_ptr is used in the equality for a record type containing a component of Chars_Ptr.	C-Test. Note: This is an Annex B test. Use foundation F452A00, and C452A01 as a pattern.
5				

(33/2)	Deleted	
(34)	NonNormative	A note.
(35)	NonNormative	Start of examples.
(36)	NonNormative	
(37/5)	NonNormative	
(38/3)	NonNormative	
(39/3)	NonNormative	

4	Check that “=” operator of types Interfaces.COBOL.Display_Format, Binary_Format, Packed_Format is used in the equality for a record type containing a component of that type.	C-Test. Note: This is an Annex B test. Use foundation F452A00, and C452A01 as a pattern.
5	Check that “=” operator of type Ada.Execution_Time.CPU_Time is used in the equality for a record type containing a component of CPU_Time.	C-Test. Note: This is an Annex D test. Use foundation F452A00, and C452A01 as a pattern.
4	Check that “=” operator of type Ada.Text_IO.Editing.Picture is used in the equality for a record type containing a component of Picture.	C-Test. Note: This is an Annex F test. Use foundation F452A00, and C452A01 as a pattern.

Paragraphs:		Objectives with tests:	Objectives to test:	Total objectives:	Objectives with submitted tests:
13	324	283	224	427	1

Must be tested	Objectives with Priority 10	1
	Objectives with Priority 9	2
Important to test	Objectives with Priority 8	20
	Objectives with Priority 7	30
Valuable to test	Objectives with Priority 6	31
	Objectives with Priority 5	34
Ought to be tested	Objectives with Priority 4	41
	Objectives with Priority 3	32
Worth testing	Objectives with Priority 2	30
Not worth testing	Objectives with Priority 1	3
	Total:	224

Objectives covered by new tests since ACATS 2.6	133
Completely:	112