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

Clauses 7.3.2-7.6.1

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).
7.3.2	(1/4)		StaticSem	Portion	Modified by Corrigendum AI12-0041-1.						
	(2/3)		StaticSem			***B732C01, C732A01*** (actually ***F732A00***)	All		Check that Type_Invariant can be specified for a private_type_declaration.		
						B732C01, C732001, C732002	All		Check that Type_Invariant can be specified for a private_extension_declaration.		
						B732C01, C732C01 (actually ***F732C00***)	All		Check that aspect Type_Invariant can be specified on the full_type_declaration that completes a private_type_declaration.		
						B732C01	Part	7	Check that aspect Type_Invariant can be specified on the full_type_declaration that completes a private_extension_declaration.	C-Test. Probably can be part of another test. The B-Test does the declaration but doesn't run it.	
				Negative		***B732C01***	All		Check that aspect Type_Invariant cannot be specified on an interface type.		
				Negative		***B732C01***	All		Check that aspect Type_Invariant cannot be specified on a record type that doesn't have a partial view.		
				Negative		***B732C01***	All		Check that aspect Type_Invariant cannot be specified on array types or elementary types.		
				Negative		***B732C01***	All		Check that aspect Type_Invariant cannot be specified on a subtype.		
				Negative		***B732C01***	All		Check that aspect Type_Invariant can only be specified on type declarations.		
	(3/4)	1	StaticSem	Definition	"Invariant expression"						
		2				***B732C02, C732002***	All		Check that aspect Type_Invariant'Class can be specified on a (tagged) private_type_declaration.		
						B732C02	Part	7	Check that aspect Type_Invariant'Class can be specified on a private_type_extension.	C-Test. Probably can be part of another test. The B-Test does the declaration but doesn't run it.	
					Added by AI12-0041-1.	***B732C02***	Part	7	Check that aspect Type_Invariant'Class can be specified on an interface declaration.	C-Test. Note that this can be anywhere, not just in a package specification. B-Test doesn't execute, of course.	
				Negative		***B732C02***	All		Check that aspect Type_Invariant'Class cannot be specified on the completion of a private type or private extension.		
				Negative	Note: Untagged illegal cases are covered by (6/3), line 1, below.	***B732C02***	All		Check that aspect Type_Invariant'Class cannot be specified on a tagged record type.		
						B732C02	All		Check that aspect Type_Invariant'Class cannot be specified on a subtype.		
						B732C02	All		Check that aspect Type_Invariant'Class can only be specified on types.		
		3		Definition	Added by AI12-0150-1, "class-wide type invariant".						
	(4/3)		NameRes			***B732001***	Part	1	Check that the expression for an aspect Type_Invariant can be of any boolean type.	C-Test. But this is highly unlikely in practice, and we have an existence test in the B-Test.	

							4	Check that the expression for an aspect Type_Invariant'Class can be of any boolean type.	C-Test. Try some non-Boolean boolean types. (But a B-Test OK line is probably good enough.)
							4	Check that the expression for an aspect Type_Invariant can be resolved with the knowledge that it is of any boolean type.	C-Test; try cases with overloaded function calls defined for Boolean and some other type. Not very important as it's normal resolution.
							4	Check that the expression for an aspect Type_Invariant'Class can be resolved with the knowledge that it is of any boolean type.	C-Test; try cases with overloaded function calls defined for Boolean and some other type. Not very important as it's normal resolution.
			Negative		***B732001***	All		Check that the expression for an aspect Type_Invariant cannot have a non-boolean type.	Could try more cases, but hardly worth it.
							5	Check that the expression for an aspect Type_Invariant'Class cannot have a non-boolean type.	B-Test. Not very important, because it's pretty obvious – but easy to check.
(5/4)	1	Redundant	Widely used	Given elsewhere, but any realistic invariant expression will test.					
	2	NameRes		Replaced by AI12-0150-1. The non-overloaded case is "Widely-Used"; any type invariant expression will test it.			4	Check that the type of the current instance in a invariant expression for aspect Type_Invariant for type T resolves to T.	C-Test. Check that overloaded calls can be resolved with the knowledge that the type is T. Not very important, this is just normal resolution.
	3	NameRes		New by AI12-0150-1. Non-overloaded cases are "Widely-Used"; any class-wide type invariant expression will test it.			4	Check that the type of the current instance in a invariant expression for aspect Type_Invariant'Class for type T effectively resolves to T for primitive operations.	C-Test. Check that overloaded calls can be resolved with the knowledge that the type is T. Not very important, this is just normal resolution.
			Negative				7	Check that the type of the current instance in a class-wide type invariant expression for type T does not resolve to type T or T'Class for objects or non-primitive operations.	B-Test. Try the Baird cases described in AI12-0113-1. Try class-wide objects declared with the type. Try non-primitive operations. (Anything else not inherited by descendants??)
	4	Redundant		Sentence 5 was removed by AI12-0159-1.					
(6/3)	1	Redundant		Given elsewhere, but we'll still test it here so we're sure that it is properly tested. (13.1.1 is the general definition of aspect specifications, it's unlikely that all of the possibilities will be checked there.)	***B732C02***	All		Check that aspect Type_Invariant'Class cannot be specified on an untagged private type.	
					B732C02	All		Check that aspect Type_Invariant'Class cannot be specified on an untagged type.	
	2	Legality			***B732001***	All		Check that aspect Type_Invariant cannot be specified on an abstract type.	
			Negative				5	Check that aspect Type_Invariant'Class can be specified on an abstract type.	C-Test.
(6.1/4)				Rule added by AI12-0042-1.			6	Check that an inherited private operation for a type with a class-wide invariant requires overriding or is abstract.	B-Test. Get examples from AI12-0042-1. (Might be able to use one of existing foundations here.)
(7/3)		Redundant	Definition	This is also widely used; any Invariant will check.					

(8/3)	Redundant	Definition	We have an objective here, as this may not come up in other tests (even through it is the intended use).	*C732002* (private extension)	Part	8	Check that if aspect Type_Invariant'Class is specified for type T, it also is checked for a type NT extended from T, even if that type is not a private type.	C-Test. Check for all kinds of extensions. Possibly define a foundation for this sort of test (the invariant would be a primitive boolean function which could be overridden).
(9/4)	Dynamic	Portion	Introductory text, tested below. Revised by AI12-0150-1.					
		Negative				8	Check that no type invariant checks are performed if the type is abstract.	C-Test. Use routines that are abstract in the type invariant, as well as concrete routines that are overridden for descendants. Ensure that the overridden routines are not called for any inherited or overridden routines for a type descended from the abstract root type. (Could also try an abstract type in the middle of a hierarchy.)
(10/4)	Dynamic		Modified by Corrigendum AI12-0133-1.	*C732A01* (specific invariant, whole stand-alone object), *C732A02* (specific invariant, components/aggregates), *C732B01* (specific invariant, whole stand-alone object)	Part	7	Check that an invariant check is applied to a default-initialized object of a type T to which invariant expressions apply, no matter where it is declared, unless the partial view of T has unknown discriminants.	C-Test. Be sure to try such an object within the defining package. Must check cases where the invariant check fails for an enabled expression, of course. Still need class-wide invariants both for full objects and for default-initialized components (both in object decls and in aggregates).
				C732C01 (specific invariant, whole stand-alone object)	Part	7	Check that an invariant check is never applied to a default-initialized object of a type T to which invariant expressions apply whose partial view has unknown discriminants.	C-Test. Only can use within the defining package. Must check cases where the invariant check fails for an enabled expression, of course. Still need class-wide invariants, and uses of default-initialized components (both in object decls and in aggregates).
(10.1/4)			Added by AI12-0049-1			8	Check that an invariant check is applied to a deferred constant with a part of a type T to which invariant expressions apply.	C-Test. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts.
(11/3)	Dynamic					8	Check that an invariant check is applied to the result of a type conversion to T, where T is a type to which invariant expressions apply.	C-Test. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. No parts here.
(12/3)	Dynamic	Portion	Long lead-in for following bullets.					
(13/3)	Dynamic			*C732001* (specific invariant, direct conversion)	Part	5	Check that when assigning to a view conversion to an ancestor of a type T to which invariant expressions apply, an invariant check is made on the T part of the object.	C-Test. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants, as well as type hierarchies where the conversion crosses an invariant (as well as direct conversions from T).
(14/3)	Dynamic			*C732001* (specific invariant, direct conversion)	Part	5	Check that when returning from a call to which a view conversion to an ancestor of a type T to which invariant expressions apply and which was passed as an in out or out parameter, an invariant check is made on the T part of the object.	C-Test. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants, as well as type hierarchies where the conversion crosses an invariant (as well as direct conversions from T).

						6	Check that appropriate specific invariant checks are performed upon return from a call to which a class-wide view conversion is passed as an in out or out parameter.	C-Test. This is testing AARM note 14.c/3, where the required checks depend on the run-time tag of the actual object. If possible, try to invent a case where checking too much would fail, but checking the correct amount would succeed. This requires a hierarchy of at least 3 levels.
						4	Check that appropriate class-wide invariant checks are performed upon return from a call to which a class-wide view conversion is passed as an in out or out parameter.	C-Test. This is testing AARM note 14.c/3, where the required checks depend on the run-time tag of the actual object. If possible, try to invent a case where checking too much would fail, but checking the correct amount would succeed. This requires a hierarchy of at least 3 levels with multiple invariants.
(15/4)	Dynamic		A presentation change in AI12-0146-1 (does not change objective).			7	Check that after a successful call to a Read or Input stream-oriented attribute, an invariant check is performed on the object initialized by the attribute.	C-Test. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. No parts here.
(16/3)	Dynamic	Portion	Lead-in for following rules.					
(17/4)	Dynamic	Portion	This and the following were redone by AI12-0042-1. Tested with the following rules.					
		Negative				7	Check that successful return from a call on a subprogram declared outside of the immediate scope of a type T that has applicable invariant expressions does not check the invariant of T.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants.
						5	Check that successful return from a call on a subprogram declared by a generic instance where the generic unit is outside of the immediate scope of a type T that has applicable invariant expressions does not check the invariant of T.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts.
(18/4)	Deleted		Removed by AI12-0042-1's wording reorganization.					
(19/4)	Dynamic	Portion	Lead-in for following bullets.					
(19.1/4)	Dynamic			***C732A01*** (specific invariant, whole object), ***C732A02*** (specific invariant, part of array), ***C732B01*** (specific invariant, whole object)	Part	8	Check that a successful return from a call on a function declared in the immediate scope of T and visible outside of that scope and that returns an object with a part of T in the return object, includes an invariant check for T.	C-Test. Check cases where the invariant check fails for an enabled expression. Still need class-wide invariants on the whole object, and class-wide invariants on parts.
(19.2/4)	Dynamic			***C732A01*** (specific invariant, whole object, in **out** of procedure), ***C732A02*** (specific invariant, part of array, **in out** of procedure), ***C732002*** (class-wide invariant, whole object, **in out** of procedure)	Part	7	Check that a successful return from a call on a subprogram declared in the immediate scope of T and visible outside of that scope and that has in out or out parameters with a part of T, includes an invariant check for T on those parameters.	C-Test. Check cases where the invariant check fails for an enabled expression. Still need class-wide invariants on parts, and at least one test of in out parameters on functions and entries.

(19.3/4)	Dynamic			***C732B02*** (specific invariant, whole designated object)	Part	7	Check that a successful return from a call on a subprogram declared in the immediate scope of T and visible outside of that scope and that has an access-to-object parameter with a designated type with a part of T, includes an invariant check for T on those parameters.	C-Test. Check both named and anonymous access type parameters. Check cases where the invariant check fails for an enabled expression. Still need a specific invariant on a part, and class-wide invariants on both whole object and on parts.
			Rule added by AI12-0149-1 (clearly missing).	***C732B02*** (specific invariant, whole designated object)	Part	7	Check that a successful return from a call on a function declared in the immediate scope of T and visible outside of that scope and that returns an access-to-object result with a designated type with a part of T, includes an invariant check for T on that result.	C-Test. Check both named and anonymous access type parameters. Check cases where the invariant check fails for an enabled expression. Still need a specific invariant on a part, and class-wide invariants on both whole object and on parts.
(19.4/4)	Dynamic					7	Check that a successful return from a call on a procedure declared in the immediate scope of T and visible outside of that scope and that has **in** parameters with a part of T, includes an invariant check for T on those parameters.	C-Test. Check cases where the invariant check fails for an enabled expression. The parameter necessarily has some sort of indirection involved for this to fail, and that indirection is modified. Try both specific and class-wide invariants. Don't forget parts.
						6	Check that a successful return from a call on a function declared in the immediate scope of T and visible outside of that scope and that has in parameters with a part of T, does not include an invariant check for T on those parameters.	C-Test. Critically important to avoid infinite recursion in invariant expressions. Check cases where the invariant check fails for an enabled expression. As above, the parameter necessarily must have some sort of indirection *and* a modification; not very likely in a function. Try both specific and class-wide invariants. Don't forget parts.
				C732A01 (specific invariant, whole object), ***C732002*** (class-wide invariant, whole object)	Part	4	Check that including in an invariant a function declared in the immediate scope of T and visible outside of that scope and that has in parameters with a part of T, does not cause infinite recursion.	C-Test. Check both for specific and class-wide invariants, and for parts, and for access types. Not that likely to be wrong, and occurs in many tests.
(19.5/4)	Dynamic	Portion	Another lead-in.					
(19.6/4)	Dynamic		We have separate objectives here to ensure that everything is covered.	***C732A01*** (specific invariant, whole object), ***C732A02*** (specific invariant, part of array), ***C732B01*** (specific invariant, whole object), ***C732002*** (class-wide invariant, whole object, in out parameters only)	Part	5	Check that invariant checks for T are performed for subprograms when the subprograms are declared within the immediate scope of T and are visible outside of the immediate scope of T, and T is a private type.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Still need class-wide invariants on the whole object (returns), and class-wide invariants on parts.
						7	Check that invariant checks for T are performed for subprograms when the subprograms are declared within the immediate scope of T and override inherited operations that are visible outside of the immediate scope of T, and T is a private type.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts.

		7	Check that invariant checks for T are performed for subprograms when the subprograms are declared within the immediate scope of T and are visible outside of the immediate scope of T, and T is a private extension.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts.
		7	Check that invariant checks for T are performed for subprograms when the subprograms are declared within the immediate scope of T and override inherited operations that are visible outside of the immediate scope of T, and T is a private extension.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts.
(19.7/4)		7	Check that invariant checks for T are performed for subprograms when the subprograms are declared within the immediate scope of T and are visible outside of the immediate scope of T, and T is a record extension.	C-Test. In this case, the invariant has to be a class-wide invariant of an ancestor. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try cases where the invariant is on a grandparent.
		7	Check that invariant checks for T are performed for subprograms when the subprograms are declared within the immediate scope of T and override inherited operations that are visible outside of the immediate scope of T, and T is a record extension.	C-Test. In this case, the invariant has to be a class-wide invariant of an ancestor. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try cases where the invariant is on a grandparent.
	Negative	6	Check that invariant checks for T are not performed for subprograms when the subprograms are not visible outside of the immediate scope of T, and T is a private type.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts. Need a case where the invariant is broken, then fixed before returning to the client test program.
	Negative	6	Check that invariant checks for T are not performed for subprograms when the subprograms are not visible outside of the immediate scope of T, and T is a private extension.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts. Need a case where the invariant is broken, then fixed before returning to the client test program.
		6	Check that invariant checks for T are not performed for subprograms when the subprograms are not visible outside of the immediate scope of T, and T is a record type.	C-Test. Check at least in out parameters and return objects. Check cases where the invariant check fails for an enabled expression. Try both specific and class-wide invariants. Don't forget parts. Need a case where the invariant is broken, then fixed before returning to the client test program.

							1	Check that invariant checks for T are not performed for subprograms even when the subprograms are declared within the immediate scope of T and override inherited operations that are visible outside of the immediate scope of T, if T is not a record extension, private extension, or private type.	C-Test, but I think this is untestable as there isn't any way to inherit from a tagged type (necessary for a class-wide invariant) that doesn't involve some sort of extension. I left this because there might be some Bairdian way to do this using generic private types.
(20/3)		Dynamic	Portion	Almost goes without saying, but we said it.					
(20.1/4)		Dynamic		Rule added by AI12-0042-1.			7	Check that invariant checks for T are performed for view conversions to class-wide types from a specific descendant of T when the conversions occur within the immediate scope of T.	C-Test. Includes T itself. See the example in Randy Brukardt's mail in the appendix of AI12-0042-1. Use enabled invariants and try both specific and class-wide invariants.
(21/4)		Dynamic		AI12-0080-1 and AI12-0159-1 corrected typos here, no semantic change.			6	Check that invariant checks for T are not performed when the Assertion_Policy is Ignore for Type_Invariant at the point of the aspect specification for Type_Invariant.	C-Test. Try both global and specific assertion policies.
							6	Check that invariant checks for T are performed when the Assertion_Policy is Check for Type_Invariant at the point of the aspect specification for Type_Invariant, even if the policy if Ignore at the point of the call.	C-Test. Try both global and specific assertion policies.
							6	Check that invariant checks for T are not performed when the Assertion_Policy is Ignore for Type_Invariant'Class at the point of the aspect specification for Type_Invariant'Class.	C-Test. Try both global and specific assertion policies.
							6	Check that invariant checks for T are performed when the Assertion_Policy is Check for Type_Invariant'Class at the point of the aspect specification for Type_Invariant'Class, even if the policy if Ignore at the point of the call.	C-Test. Try both global and specific assertion policies.
							6	Check that invariant checks for T whose parent is P are performed when the Assertion_Policy is Check for Type_Invariant'Class at the point of the aspect specification for Type_Invariant'Class for P, even if the policy if Ignore at the declaration of T.	C-Test. Try both global and specific assertion policies.
(22/3)	1	Dynamic	Portion	Can't test this separately because of the arbitrary order rules.					
	2			Almost widely-used (every test for a failing invariant will depend on this, but we ought to have at least one test that has this specifically as one of the objectives.	***C732A01***	All		Check that Assertion_Error is raised if any enabled invariant expression yields False when evaluated.	
	3		Not Testable	“Arbitrary order” is not testable.					
	4						7	Check that invariant checks on a call are performed before any copy-back of parameters.	C-Test. Check that the by-copy parameters are not modified after an invariant check fails, either for the parameters or for the function result.
	5		Not Testable	“Arbitrary order” is not testable.					

	(22.1/4)				Rule added by AI12-0150-1, reworded by AI12-0159-1.			8	Check that a class-wide invariant check always calls the routines for type T, even when the tag of the object identifies some other descendant type.	C-Test. Specifically, we're trying to check that the routines do not dispatch. Be sure to test cases where the invariant is defined on an ancestor of T. Important because it could represent a change.
	(23/3)							6	Check that the specific invariants evaluated for a dispatching call are those of the subprogram actually invoked.	C-Test.
								5	Check that the specific invariants evaluated for a call through an access-to-subprogram are those of the actual subprogram.	C-Test. This is probably only interesting in the case in the AARM note (as the types have to be the same as subtype conformance is required), so it mainly is the presence or absence of the check.
								4	Check that the class-wide invariants evaluated for a dispatching call are those of the subprogram actually invoked.	C-Test. This is only interesting when the dispatching call is for a root type, but the class-wide invariant is added later in the derivation tree. (If it was on the root, all of the routines would have the same invariant.)
								4	Check that the class-wide invariants evaluated for a call through an access-to-subprogram are those of the actual subprogram.	C-Test. This isn't likely to be interesting, other than in the case from the AARM note.
	(24/3)		NonNormative		This is a note. But the case discussed in the note seems like it should be tested directly.			7	Check that for a derived type NT, specific invariants are checked for both T and NT for an inherited primitive subprogram, while only the specific invariants of NT are checked for an overridden primitive subprogram.	C-Test.
7.4	(1)		Redundant							
	(2)	1	Redundant							
		2	Definitions		Deferred constant					
		3	Legality	Subpart	Any legal test of deferred constants will test this.					
				Negative	Modified by Ada 2012, AI05-0229-1 to talk about aspects rather than pragmas.	B740001			Check that a deferred constant declaration requires a completion of a full constant declaration unless aspect Import is true for the deferred constant.	Note: We check the case where aspect Import is True for 7.4(8/3).
	(3)	1	Legality	Subpart	Any legal test of deferred constants will test this.					
				Negative		**B740003**	All		Check that a deferred constant declaration completed with a full constant declaration can only be given in the visible part of a package specification.	
		2	Legality	Portion	Lead-in for the bullets below.					
	(4)		Legality	Subpart	Any legal test of deferred constants will test this.					
				Negative		**B740003**	All		Check that the full constant declaration that completes a deferred constant declaration can only occur in the private part of the same package.	
	(5/2)		Legality	Subpart	The same type isn't very interesting, and other tests will cover that.					
								6	Check that a deferred constant declaration can include an anonymous access type.	C-Test.

			Negative		**B740002**	All		Check that the full constant declaration completing a deferred constant declaration is illegal if it has an anonymous access type that does not statically match that of the deferred constant declaration.	
							4	Check that the full constant declaration completing a deferred constant declaration is illegal if it does not have an anonymous access type and the type is not the same as the one used in the deferred constant declaration.	B-Test. Try numeric types with the same range; and structurally similar records. No tests in ACATS 2.6; the coverage document claims that B740001 tests this, but it does not (it only tries complete omission of the completion).
					B740001			Check that the full constant declaration that completes a deferred constant declaration cannot declare an anonymous array type.	This will always be a separate type.
(6/3)	1	Legality	Subpart	Any legal test of deferred constants will test this.					
			Negative	Approved AI05-0062-1 changed this wording.			4	If the deferred constant declaration includes a constrained subtype_indication, the full constant declaration is illegal if its constraint does not statically match that of the deferred constant.	B-Test. No tests in ACATS 2.6; the coverage document claims that B740001 tests this, but it does not.
	2	Redundant		This is really the lack of a rule, but we test it anyway as implementers are likely to require exact matching.	C74307A		2	Check that if the subtype of a deferred constant declaration is unconstrained, the full constant declaration can give any subtype of the type.	C-Test. Try index constraints.
	3	Redundant							
(7/2)		Legality					3	Check that a full constant declaration can give **aliased** even if the deferred constant does not.	C-Test.
			Negative		B740001			Check that the full constant declaration must include **aliased** if the deferred constant declaration includes **aliased.**	
(7.1/2)		Legality		Approved AI05-0062-1 makes this objective valid.			5	Check that a full constant declaration can exclude null even if the deferred constant does not.	C-Test. Tested in B-Test. Note that a private type completed by an access type may not allow a null exclusion on the deferred constant; this should be tested.
					B740002	All		Check that the full constant declaration must exclude null if the deferred constant declaration excludes null.	
(8/3)		Legality		Modified by Ada 2012, AI05-0229-1 to talk about aspects rather than pragmas.			4	Check that a deferred constant declaration for which aspect Import is True can appear anywhere.	C-Test. This is marked as untested in the coverage document for ACATS 2.6. This will need a test like the ones in Annex B for C interfacing.
							3	Check that a deferred constant declaration for which aspect Import is True cannot also be completed with a full constant declaration.	B-Test. Use “Ada” as the convention name to avoid having to use something implementation-defined.
(9/2)		Legality			B74304A (initializing objects), B74304B (generic in parameter), B74304C (generic in parameter)		4	Check that a use of a deferred constant that freezes the constant before the completion is illegal.	B-Test. Pretty much any use other than in a default_expression is illegal. Try in a range constraint, index constraint, and discriminant constraint. (Ada 83 rules made these unlikely, so they were not tested; but they're not as unlikely in Ada 95 or later.) Also try in an object renames.

						B74304B (generic defaults), C74305A (parameters, record components), C74305B (parameters)			Check that the use of a deferred constant in a default_expression is not considered freezing.	
	(10/3)		Dynamic		AI05-0004-1 adds access_definition, which was missing.			2	Check that the elaboration of a deferred constant elaborates the subtype indication, access definition, or array type declaration.	C-Test. Try an array type declaration (access definition elaborations have no effect).
					This is caused by 3.3.1(7) and the lack of a prohibition here; we need to test it here since it is related to completions.	C74302A			Check that multiple declarations can be used for deferred constant declarations, even if the full declarations are given individually.	
						C74302A			Check that multiple declarations can be used for full constant declarations completing deferred constant declarations, even if the deferred declarations are given individually.	
	(11)		NonNormative		A note.					
	(12)		NonNormative		Start of an example...					
	(13)		NonNormative							
	(14)		NonNormative		...end of the example.					
7.5	(1/2)		General							
	(2/2)	1	Legality	Subpart	Any test of limited tagged types with limited components will check this.					
				Negative		B391004, B730001			Check that a non-limited tagged record declaration is illegal if it has any limited components.	
		2	Redundant		Defined in 3.4(5.1/2) and 3.9.4(12/2).					
	(2.1/3)		Legality	Portion	This is the lead-in (and meat) of the following bullets.					
					Other contexts don't have restrictions; check that.			6	Check that in an actual parameter of a subprogram call, an expression of a limited type is not restricted; specifically, object names are allowed.	C-Test. Must check that we don't go too far.
								6	Check that in a default expression for a subprogram parameter, an expression of a limited type is not restricted; specifically, object names are allowed.	C-Test. Must check that we don't go too far.
								5	Check that an object renaming allows renaming limited objects that are function calls that are dereferenced, indexed, sliced, and selected.	C-Test. Must check that we don't go too far.
					Conditional_expressions added by AI05-0147-1, new test cases below.					
	(2.2/2)		Legality	Subpart	Tests of legal limited expressions will cover this.					
				Negative		**B750A01**	All		In the initialization expression of an object declaration, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.	
					Added by AI05-0147-1.	***B750A08***	All		In the initialization expression of an object declaration, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).	

			Added by AI12-0172-1 (not in TC1).			1	In the initialization expression of an object declaration, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute.	B750A08 contains these cases, commented out.
(2.3/2)	Legality	Subpart	Tests of legal limited expressions will cover this.						
		Negative		**B750A02**	All		In the default expression of a component declaration, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.		
			Added by AI05-0147-1.	***B750A09***	All		In the default expression of a component declaration, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).		
			Added by AI12-0172-1 (not in TC1).			1	In the default expression of a component declaration, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute.	B750A09 contains these cases, commented out.
(2.4/2)	Legality	Subpart	Tests of legal limited expressions will cover this.						
		Negative		**B750A03**	All		In the expression of a record component association of an aggregate, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.		
			Added by AI05-0147-1.			7	In the expression of a record component association of an aggregate, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).	B-Test. Try both if and case expressions; also try nested cases. Also try raise expressions (see below).	
			Added by AI12-0172-1 (not in TC1).			1	In the expression of a record component association of an aggregate, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute. Include in test for previous objective.	
(2.5/2)	Legality	Subpart	Tests of legal limited expressions will cover this.						
		Negative				7	In the expression for the ancestor part of an extension aggregate, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.	B-Test. Try object names (including those dereferenced, indexed or selected), functions that are dereferenced, indexed, or selected, type conversions, qualified and parenthesized versions of these. Use foundation F750A00 and pattern on B750A02.	
			Added by AI05-0147-1.			6	In the expression for the ancestor part of an extension aggregate, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).	B-Test. Try both if and case expressions; also try nested cases. Also try raise expressions (see below).	
			Added by AI12-0172-1 (not in TC1).			1	In the expression for the ancestor part of an extension aggregate, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute. Include in test for previous objective.	
(2.6/2)	Legality	Subpart	Tests of legal limited expressions will cover this.						

		Negative				8	In an expression of an array aggregate, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.	B-Test. Try object names (including those dereferenced, indexed or selected), functions that are dereferenced, indexed, or selected, type conversions, qualified and parenthesized versions of these.	
			Added by AI05-0147-1.			7	In the expression of an array aggregate, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).	B-Test. Try both if and case expressions; also try nested cases. Also try raise expressions (see below).	
			Added by AI12-0172-1 (not in TC1).			1	In the expression of an array aggregate, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute. Include in test for previous objective.	
(2.7/2)	Legality	Subpart	Tests of legal limited expressions will cover this.						
		Negative		*B750A04*	All		In the qualified expression of an initialized allocator, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.		
			Added by AI05-0147-1.	*B750A10*	All		In the qualified expression of an initialized allocator, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).		
			Added by AI12-0172-1 (not in TC1).			1	In the qualified expression of an initialized allocator, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute.	B750A10 contains these cases, commented out.
(2.8/2)	Legality	Subpart	Tests of legal limited expressions will cover this.						
		Negative		*B750A05, B750A06*	All		In the expression of a return statement, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.		
			Added by AI05-0147-1.	*B750A11, B750A12*	All		In the expression of a return statement, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).		
			Added by AI12-0172-1 (not in TC1).			1	In the expression of a return statement, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute.	B750A11 and B750A12 contains these cases, commented out.
(2.9/3)	Legality	Subkind	Tests of legal limited expressions will cover this. Rule added by Ada 2012, AI05-0177-1.						
		Negative		*B750A07*	All		In the expression of an expression function, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.		
				B750A13	All		In the expression of an expression function, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).		

			Added by AI12-0172-1 (not in TC1).		1	In the expression of an expression function, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute.	B750A13 contains these cases, commented out.
(2.10/3)	Legality	Subpart	Tests of legal limited expressions will cover this.					
		Negative			7	In the default expression or actual parameter for a generic formal object of mode in, an expression of a limited type cannot be anything other than an aggregate, function call, or a qualified or parenthesized expression whose operand would be allowed.	B-Test. Try object names (including those dereferenced, indexed or selected), functions that are dereferenced, indexed, or selected, type conversions, qualified and parenthesized versions of these. Use foundation F750A00 and pattern on B750A02.	
			Added by AI05-0147-1.		6	In the default expression or actual parameter for a generic formal object of mode in, an expression of a limited type cannot be a conditional expression which has a dependent expression that is not allowed by 7.5(2.1).	B-Test. Try both if and case expressions; also try nested cases. Also try raise expressions (see below.)	
			Added by AI12-0172-1 (not in TC1).		1	In the default expression or actual parameter for a generic formal object of mode in, an expression of a limited type can be a raise expression.	B-Test is sufficient, no need to try to execute. Include in test for previous objective.	
(3/3)	Definitions	Portion	Lead-in for bullets below; defines "limited". Changed to include "view of" by AI-178, no testing difference.					
(4/2)	Definitions				3	Check that a value of a derived type with the word limited cannot be assigned or compared for equality.	B-Test.	
					3	Check that a value of an interface with the words synchronized, task, or protected is limited.	B-Test.	
					3	Check that a value of a record type with the reserved word limited cannot be assigned or compared for equality.	B-Test. Marked as untested in ACATS 2.x.	
				B92001B		Check that a value of a task type cannot be assigned or compared for equality.	B-Test.	
					3	Check that a value of a protected type cannot be assigned or compared for equality.	B-Test.	
(5/3)	Definitions		Added rule from approved AI05-0087-1.		3	Check that a value of a class-wide type whose specific type is limited cannot be assigned or compared for equality.	B-Test.	
(6/2)	Definitions			B74404B		Check that value of a composite type with a limited component cannot be assigned or compared for equality.		
(6.1/3)	Definitions		Added by AI05-0178-1, but already was in 3.10.1(2.1/2).		6	Check that an object that has an incomplete view cannot be assigned.	B-Test. One way to do this is to have a subprogram with two parameters with a tagged incomplete type from a limited with. Then A := B is illegal in the body as the incomplete view is limited, and there is no other reason for an error. AI05-0178-1 has another way.	
(6.2/2)	Definitions		Careful: this was renumbered by AI05-0178-1.		4	Check that a value of a derived type whose parent type is a limited non-interface type cannot be assigned or compared for equality.	B-Test.	
(7/2)	Definitions							
					6	Check that a value of a derived type whose parent type is a limited interface type but that is not otherwise limited can be assigned and compared for equality.	C-Test.	

	(8/2)	Redundant							
	(8.1/3)	Definitions	Lead-in.	Defines "immutably limited"; approved by AI05-0052-1.					
	(8.2/3)	Definitions		Check in 3.7(10/2); from AI05-0052-1.					
	(8.3/3)	Definitions		Check in 3.7(10/2); from AI05-0217-1 (a correction to AI05-0052-1).					
	(8.4/3)	Definitions		Check in 3.7(10/2); from AI05-0052-1.					
	(8.5/3)	Definitions		Check in 3.7(10/2); from AI05-0052-1.					
	(8.6/3)	Definitions		Check in 3.7(10/2); from AI05-0052-1.					
	(8.7/3)	Definitions		Check in 3.7(10/2); from AI05-0052-1.					
	(8.8/3)	Deleted		Careful: this was renumbered by AI05-0052-1 and AI05-0217-1 and then deleted by AI05-0067-1.					
	(9/2)	NonNormative		A note.					
	(10/2)	Deleted							
	(11/2)	Deleted							
	(12/2)	Deleted							
	(13/2)	Deleted							
	(14/2)	Deleted							
	(15/2)	Deleted							
	(16)	NonNormative		A note.					
	(17)	NonNormative		Start of an example...					
	(18)	NonNormative							
	(19)	NonNormative							
	(20)	NonNormative							
	(21)	NonNormative							
	(22)	NonNormative							
	(23/2)	NonNormative		...end of example.					
7.6	(1)	General							
	(2)	General							
	(3)	StaticSem	Portion	A lead-in for the next part.					
	(4/3)	StaticSem	Widely used	Any use of a controlled type.					
				In Ada 2012, AI05-0212-1 makes this Pure.	***C760014***	All		Check that package Ada.Finalization is pure.	
	(5/2)	StaticSem	Widely used	Any use of a non-limited controlled type.					
			Subpart	Preelaborable initialization objectives are tested in 10.2.1.					
	(6/2)	StaticSem					3	Check that Initialize, Adjust, and Finalize are inherited for a type derived from Controlled and that they can be called but do nothing.	C-Test. Low priority because doing nothing is not very interesting, and normal operation is tested widely.

(7/2)		StaticSem	Widely used	Any use of a limited controlled type.				
			Subpart	Preelaborable initialization objectives are tested in 10.2.1.				
(8/2)		StaticSem				3	Check that Initialize and Finalize are inherited for a type derived from Limited_Controlled and that they can be called but do nothing.	C-Test. Low priority because doing nothing is not very interesting, and normal operation is tested widely.
(9/2)	1	Definitions	Widely used	"Controlled type"				
				This objective is here as it doesn't fit anywhere else, and it is new for Ada 2005 (caused by AI95-0344-1 repealing the 3rd sentence of 3.9.1(3)).	***C760015*** (subprograms)	6	Check that a controlled type can be declared at any level.	C-Test. Try in tasks, and in generic units instantiated in tasks and subprograms. Also might try in a block in one of the above.
	2	Dynamic				5	Check that "=" for type Controlled returns True.	C-Test. Check when it is incorporated into an extension.
	3	General						
(9.1/2)		Definitions	Portion	"Needs finalization"; this is the lead-in for the definition.				The easiest way to check proper definition of "needs finalization" is to use the Unchecked_Union rule added by AI05-0026. We do that here.
(9.2/2)		Definitions				6	Check that a component declared in a variant_part of an Unchecked_Union type cannot need finalization by being a controlled type.	B-Test; depends on AI05-0026.
						6	Check that a component declared in a variant_part of an Unchecked_Union type cannot need finalization by being a protected type.	B-Test; depends on AI05-0026.
						6	Check that a component declared in a variant_part of an Unchecked_Union type cannot need finalization by being a task type.	B-Test; depends on AI05-0026.
(9.3/3)		Definitions		AI05-0092-1 rewords this slightly, but the testing remains unchanged.		6	Check that a component declared in a variant_part of an Unchecked_Union type cannot need finalization by having a component that needs finalization.	B-Test; depends on AI05-0026. Check components of controlled types, protected types, task types, and types with components of these.
(9.4/3)		Definitions		Original rule replaced by approved AI05-0013. Can't test this with a component (as we do with the others) as class-wide components are illegal.		1	Check that <something> cannot need finalization by having a class-wide type.	Not sure how to test this.
(9.5/3)		Definitions		New rule from approved AI05-0026.		6	Check that a component declared in a variant_part of an Unchecked_Union type cannot need finalization by being a private type whose full type needs finalization.	B-Test; depends on AI05-0026. Check full types of controlled types, protected types, task types, and types with components of these.
(9.6/2)		Definitions		Careful: Paragraph number changed by AI05-0026.		7	Check that a component declared in a variant_part of an Unchecked_Union type cannot need finalization by being a language-defined type that needs finalization.	B-Test. Try various file I/O types, and containers types, and others. Higher priority because it is more likely to be wrong.
(10/2)	1	Dynamic			C760001 (object decls), C760009 (extension aggs ancestor parts)	6	Check that Initialize is called on controlled components that do not have an initialization expression that occur in an top-level object that is initialized by default.	C-Test(s). Need to test <> in aggregates, and the return object in extended return statements.
	2	Dynamic			C760001		Check that Initialize is called on top-level controlled objects that are initialized by default.	

			Negative		C760001, C760013		Check that Initialize is not called for an object or component whose value is assigned (including by default initial expressions).	
(11/2)		Dynamic			C760009		Check that Initialize is called for a extension aggregate whose ancestor_part is a subtype_mark denoting a controlled subtype, unless than Initialize routine is abstract.	
						5	Check that an extension aggregate can have a subtype_mark denoting a controlled subtype with an abstract Initialize routine.	C-Test. The subtype will necessarily be abstract.
(12/2)	1	Dynamic	Not Testable	An arbitrary order can be anything, so there is nothing to test.				
	2	Dynamic				5	Check that Initialize for an entire object is applied after the initialization of its components.	C-Test. (This may be a side-effect of some other test, but not one of those for 7.6.)
	3	Dynamic			C760012		Check that record components that have per-object access discriminant constraints are initialized after any components that are not so constrained.	
						4	Check that protected type components that have per-object access discriminant constraints are initialized after any components that are not so constrained.	C-Test.
	4	Dynamic			C760012		Check that record components that have per-object access discriminant constraints are initialized in the order of their component declarations.	
						4	Check that protected type components that have per-object access discriminant constraints are initialized in the order of their component declarations.	C-Test.
	5	Dynamic				5	Check that any task activations required for an allocator occur after any needed calls to Initialize.	C-Test.
(13)		Dynamic	Portion	This is just a lead-in.				
(14)		Dynamic	Widely-used	Any assignment will test this.				
(15)		Dynamic	Subpart	This just says that adjustment happens; what that means is given in the following paragraphs.				
(16/3)	1	Dynamic		Slightly modified by AI05-0067-1	C760002 (object decls, assignment), C760007 (simple return statement, aggregates)	6	Check that on any assignment operation, Adjust is called on any controlled parts of the operation.	C-Test: Check for extended return statements. Careful: There are permissions to eliminate these operations.
					C760002 (object decls, assignment)	3	Check that Adjust is called on the assignment to an object after the adjustment of all of its components.	C-Test: Check for aggregates, ancestor parts of extension aggregates, and extended return statements. Careful: There are permissions to eliminate these operations.
	2	Dynamic	Not Testable	No effect is not testable, since we aren't going to try to guess possible incorrect effects.				
(17)		Dynamic				7	Check that any controlled part in the target of an assignment_statement is finalized before the value is assigned to it.	C-Test. This does not appear to be tested in ACATS 2.6.
			Subpart	Adjust is tested as part of 7.6(16).				

							5	Check that if an assignment_statement uses an anonymous object, it is finalized at the end of the statement.	C-Test. This test requires care to avoid tripping over the permissions of 7.6(18-21).
	(17.1/3)	Definitions	Subpart	"built in place"					
	(17.2/3)	Dynamic					8	Check that no separate anonymous object is used for an immutably limited function call initializing an object.	C-Test; use foundation F760A00, try cases like those in C760A01. Make sure to test aggregates in return statements (didn't do that in C760A01).
							7	Check that no separate anonymous object is used for an immutably limited expression function call initializing an object.	C-Test; use foundation F760A00, try cases like those in C760A01. Make sure to test aggregates as the return expression (didn't do that in C760A01).
					C760A01	All		Check that no separate anonymous object is used for a limited aggregate initializing an object.	Test originally was in section 7.5 in ACATS 3.0.
	(17.3/3)	Dynamic			C761010		6	Check that the assignment (other than in an assignment_statement) of an aggregate with a controlled part does not use an anonymous object.	C-Test: Add subtests for controlled subcomponents.
	(17.4/3)	Dynamic	Not Testable	This is unspecified.					
	(17.5/3)	Dynamic	Subpart	Lead-in					
	(17.6/3)	Dynamic	Subpart	Tested as part of 7.6(17.2-3), if testable at all.					
	(17.7/3)	Dynamic	Subpart	Tested as part of 7.6(17.2-3).					
	(17.8/3)	Dynamic	Subpart	Tested as part of 7.6(17.2-3).					
	(17.9/3)	Dynamic	Subpart	Tested as part of 7.6(17.2-3), if testable at all.					
	(17.10/3)	Dynamic	Subpart	Tested as part of 7.6(17.2-3), if testable at all.					
	(17.11/3)	Dynamic	Subpart	Tested as part of 7.6(17.2-3), if testable at all.					
	(17.12/3)	Deleted		Deleted by AI05-0067-1.					
	(18/3)	Impl-Perm	Portion	This is just a lead-in.					
	(19/3)	Impl-Perm	Not Testable	But take care that other tests take this permission into account. Modified by AI05-0067-1.					
	(20/3)	Impl-Perm	Not Testable	But take care that other tests take this permission into account. Modified by AI05-0067-1.					
	(21/3)	Impl-Perm	Not Testable	But take care that other tests take this permission into account. Modified by AI05-0067-1.					
	(22/2)	Impl-Perm	Not Testable	But take care that other tests take this permission into account.					
	(23/2)	Impl-Perm	Portion	Part of the previous rule.					
	(24/2)	Impl-Perm	Portion	Part of the previous rule.					
	(25/2)	Impl-Perm	Portion	Part of the previous rule.					
	(26/2)	Impl-Perm	Portion	Part of the previous rule.					
	(27/2)	Impl-Perm	Portion	Part of the previous rule.					
7.6.1	(1)	General							

(2/2)	1	Definitions	Subpart	Completion. Other rules (like finalization) tests this.				
	2	Definitions	Subpart	Normal completion. Other rules (like finalization) tests this.				
	3	Definitions	Subpart	Abnormal completion. Other rules (like finalization) tests this.				
(3/2)	1	Definitions	Not testable	Left (a construct). This has no effect of its own.				
	2	Definitions		Master. We test this here because it is too complex to get right otherwise.	C760011, C761003	3	Check that a subprogram body is a master: leaving the body causes objects declared in that body to be finalized.	C-Test. Try function bodies.
				Note: Protected bodies can't have objects.		7	Check that a task body is a master: leaving the body causes objects declared in that body to be finalized.	C-Test.
					C760011 (function calls, aggregates).	4	Check that a procedure call is a master: leaving the call causes objects created by that call to be finalized.	C-Test; check anonymous access allocators and possibly task awaiting.
						6	Check that an entry call is a master: leaving the call causes objects created by that call to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting.
					C761002		Check that a block statement is a master: leaving the block causes objects declared in the block to be finalized.	
					C760011 (function calls, aggregates).	4	Check that the expression of an if statement is a master: leaving the expression causes objects created by that expression to be finalized.	C-Test; check anonymous access allocators and possibly task awaiting. Also try "elsif".
						7	Check that the expression of an case statement is a master: leaving the expression causes objects created by that expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. Note: The choices of a case statement need to be static and elementary, thus they aren't interesting.
						7	Check that the expression of an while loop and the range of a for loop are masters: leaving the loop header causes objects created by the header to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting.
						8	Check that an assignment_statement is a master: leaving the statement causes objects created by the expressions to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. Be sure to test both the source expression and the target name.
						8	Check that a return_statement is a master: leaving the statement causes objects created by the expression (other than the return object) to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. Be sure to exclude the return object, and to try both simple and extended returns.
						7	Check that an exit statement is a master: leaving the statement causes objects created by the when expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting.
						7	Check that a raise_statement is a master: leaving the statement causes objects created by the message expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting.
						7	Check that a delay_statement is a master: leaving the statement causes objects created by the expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting.

						6	Check that an abort_statement is a master: leaving the statement causes objects created by the name to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. Mst of these cases only can occur in the expression of an array index or in a function call.
						8	Check that the expression of an object declaration is a master: leaving the declaration causes objects created by that expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. C760011 appears to cover this but does not require finalization soon enough.
						8	Check that the actual parameter expressions given in an generic instantiation are masters: leaving the instance causes objects created by that expressions (but not the values of the expressions) to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting for parameters of function calls.
						7	Check that the expressions and ranges in constraints of a type or subtype declaration are masters: leaving the declaration causes objects by those expressions and ranges to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. Don't forget in constraints in components and discriminants.
						7	Check that an expression or function_call renamed as an object is a master: leaving the renames causes objects created by that expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting.
						6	Check that an expression renamed as a a subprogram is a master: leaving the renames causes objects created by that expression to be finalized.	C-Test, check aggregates, function calls, anonymous access allocators, and possibly task awaiting. The expression must have an access-to-subprogram type.
					C761001		Check that objects declared in library-level packages are finalized when the environment task is completed.	
	3	Definitions	Subpart	Tested as part of the previous sentence.				
(4)	1	Dynamic	Subpart	Tested as part of 9.3.				
	2	Dynamic	Subpart	Tested as part of 7.6.1(3/2).				
	3	Redundant						
	4	Dynamic				9	Check that all masters are finalized innermost-out when an exit statement causes several masters to be left.	C-Test.
					C761002	9	Check that all masters are finalized innermost-out when a goto statement causes several masters to be left.	C-Test. The existing test only gotos out of one master.
					C761002	9	Check that all masters are finalized innermost-out when a return statement causes several masters to be left.	C-Test. The existing test doesn't check the order of finalization.
						7	Check that all masters are finalized innermost-out when a requeue statement causes several masters to be left.	C-Test
						7	Check that all masters are finalized innermost-out when the selection of a terminate alternative causes several masters to be left.	C-Test.
					C761004	5	Check that all masters are finalized innermost-out when exception propagation causes several masters to be left.	C-Test. The existing test is simple: a single recursive function.
(5)		Dynamic	Portion	Just a lead-in for the below.				
(6/3)		Dynamic	Not Testable	No effect is not testable; we aren't going to guess what implementers might do wrong. Wording clarified by AI05-0099-1, no semantic change.				

(7/3)		Dynamic	Widely Used	Any controlled type C-Test will check this. Wording clarified by AI05-0099-1, no semantic change.					
(8/3)		Dynamic	Subpart	Tested in 9.4(20). Wording clarified by AI05-0099-1, no semantic change.					
(9/3)		Dynamic		Wording clarified by AI05-0099-1, no semantic change.	C760012			Check that record components that have per-object access discriminant constraints are finalized in the reverse order of their component declarations, and before any components that are not so constrained.	
(9.1/2)		Dynamic					6	Check that each coextension of an object is finalized after the object that designates it.	C-Test.
(10)		Dynamic			C761002			Check that Unchecked_Deallocation of a controlled object causes finalization of that object.	
(11/3)	1	Dynamic		Wording revised by AI05-0190-1, no semantic change here.	C761003, C761004, C761005			Check that objects created by declarations are finalized in reverse order of their creation.	This is also covered indirectly by many other tests.
	2	Definitions		Defines "existence". Testing that would be rather meta-physical. :-)					
(11.1/3)	1	Definitions		Defines “collection”. Rules split out and changed by AI05-0190-1.					
	2	Dynamic	Subpart	Tested as part of testing finalization of a collection.					
	3	Dynamic	Subpart	Tested as part of testing finalization of a collection.					
	4	Dynamic	Lead-in						
(11.2/3)		Dynamic			C761002		3	Check that objects created by an allocator are finalized at the appropriate point for named access types.	C-Test: Try cases where it's possible to tell that the finalization happens at the first freezing point of the access type.
(11.3/3)		Dynamic					6	Check that objects created by an allocator for an anonymous access parameter are finalized immediately after the associated call returns.	C-Test. Careful: No order is required if there are more than one in a single call.
(11.4/3)		Dynamic					6	Check that objects created by an allocator for an anonymous access return type are finalized with the master of the call.	C-Test.
(11.5/3)		Dynamic					7	Check that objects created by an allocator for an anonymous access type other than an access parameter or return type are finalized when the innermost enclosing declaration is finalized.	C-Test. Careful: No order is required if there are more than one in a declaration.
(12/2)		Dynamic	Subpart	Tested in 7.6(17).					
(13/3)	1	Dynamic		Much of this is tested by the tests for 7.6(3/2). We try some unusual cases here.	**C761013**	All		Check that a function call renamed as an object is not finalized until the unit or block that directly contains the renaming is left.	
					C761013	All		Check that a renaming of a controlled object is not finalized too soon (which an object declared at the place of a renaming would be).	
							6	Check that a object allocated for a derived access type is not finalized until the finalization of the collection for the (ultimate) parent access begins.	C-Test.

2			Added by AI05-0142-4, modified by AI05-0269-1.		8	Check that the anonymous object associated with the actual object of an explicitly aliased parameter is not finalized until the innermost master enclosing the function call is finalized.	C-Test.
3			Added by AI05-0066-1.		8	Check that the anonymous object associated with a function_call or aggregate is finalized as soon as the master of the call or aggregate is.	C-Test. This may be hard to test, as there is no requirement for such an object to be created. Perhaps it can be forced by using it as a non-aliased parameter.
(13.1/3)	Dynamic	Subpart	Most normal cases are tested in the objectives for 7.6.1(3/2). Wording changed by AI05-0066-1 and AI05-0262-1, but no testing impact.	C761012	3	Check that anonymous objects associated with an expression are finalized if a transfer of control or exception occurs before the expression is left.	C-Test, check aggregates (if possible), anonymous access allocators, and possibly task awaiting.
(14/1)	BoundedErr	Negative	These cases that are not a bounded error.	C760010		Check that explicit calls to Adjust and Finalize raise the exception propagated, not Program_Error.	
				C760010		Check that all calls to Initialize raise the exception propagated, not Program_Error	
(15)	BoundedErr			C761006		For a Finalize that propagates an exception and that was called as part of an assignment statement, check that Program_Error is raised at the point of the assignment.	
(16/2)	BoundedErr			C761006		For an Adjust that propagates an exception and that was called as part of an assignment statement, check that Program_Error is raised at the point of the assignment after any other Adjusts due to be performed are called.	
				C761006		For an Adjust that propagates an exception that was called as part of an assignment other than an assignment statement, check that Program_Error is raised at the point of the assignment (other Adjusts may or may not be called).	
(17)	BoundedErr			C761006		For a Finalize that propagates an exception and that was called as part of an Unchecked_Deallocation, check that Program_Error is raised after any other Finalizes due to be performed are called.	
(17.1/3)	Deleted		This paragraph was deleted by AI05-0064; the following rule and objective are sufficient to cover this case. (But having the extra test cases is still worthwhile.)	C761011		For a Finalize that propagates an exception and that was called as part of finalizing an anonymous object, check that Program_Error is raised after any other Finalizes due to be performed are called.	
(17.2/1)	BoundedErr			C761011		For a Finalize that propagates an exception and that was called as part of the finalizations caused by the end of execution of a master, check that Program_Error is raised after any other Finalizes due to be performed are called.	
(18/2)	BoundedErr			C761011		For a Finalize that propagates an exception and that was invoked by the transfer of control of an exit statement, check that Program_Error is raised no later than the point where normal execution would have resumed after any other Finalizes due to be performed are called.	
				C761011		For a Finalize that propagates an exception and that was invoked by the transfer of control of a goto statement, check that Program_Error is raised no later than the point where normal execution would have resumed after any other Finalizes due to be performed are called.	

				C761011		For a Finalize that propagates an exception and that was called invoked by the transfer of control of a return statement, check that Program_Error is raised no later than the point where normal execution would have resumed after any other Finalizes due to be performed are called.	
					5	For a Finalize that propagates an exception and that was invoked by the transfer of control of a requeue statement, check that Program_Error is raised no later than the point where normal execution would have resumed after any other Finalizes due to be performed are called.	C-Test.
(19)	BoundedErr			C761011		For a Finalize that propagates an exception and that was invoked by the transfer of control caused by exception propagation, check that Program_Error is raised after any other Finalizes due to be performed for the master are called.	
(20)	BoundedErr				5	For a Finalize that propagates an exception and that was invoked by an abort, check that any other Finalizes due to be performed are called and the exception ignored.	C-Test.
				C761007		For a Finalize that propagates an exception and that was invoked by the selection of a terminate alternative, check that any other Finalizes due to be performed are called and the exception ignored.	
(20.1/3)	Impl-Perm	Not Testable	A permission. Must take care that other tests don't violate this. Added by AI05-0107-1.				
(20.2/3)	Impl-Perm		This permission is binary; it allows finalization at exactly one of two places, which we can test. Added by AI05-0111-3.		6	Check that objects created by an allocator from a storage pool that supports subpools are finalized either when their associated named access type is finalized or when the storage pool object is finalized.	C-Test.
(21/3)	NonNormative	Not Testable	A Note, editorially changed only.				
(22)	NonNormative	Not Testable	A Note				
(23)	NonNormative	Not Testable	A Note				
(24)	NonNormative	Not Testable	A Note				

	Paragraphs:		Objectives with tests:	Objectives to test:	Total objectives:	Objectives with submitted tests:
5	173		99	153	222	5

Must be tested	Objectives with Priority 10	0
	Objectives with Priority 9	3
Important to test	Objectives with Priority 8	14
	Objectives with Priority 7	38
Valuable to test	Objectives with Priority 6	36
	Objectives with Priority 5	17
Ought to be tested	Objectives with Priority 4	18
	Objectives with Priority 3	13
Worth testing	Objectives with Priority 2	2
Not worth testing	Objectives with Priority 1	12
	Total:	153

Objectives covered by new tests since ACATS 2.6	52
Completely:	37