THE ADA COMPILER VALIDATION CAPABILITY (ACVC) VERSION 2.0.1 TEST OBJECTIVES DOCUMENT

6 March 1996

Prepared for:

Ada 9X Project : ACVC Program

Prepared by:

Science Applications International Corporation
10770 Wateridge Circle
San Diego, CA 92121

B354001
Check that the expression of a modular_type_definition must be static and that the expected type of the expression can be of any integer type. Check that the modulus must be positive. Check that moduli that are powers of two are allowed up to and including, but not exceeding, System.Max_Binary_Modulus. Check that non-power-of-two moduli are allowed as long as they do not exceed System.Max_Nonbinary_Modulus. Check that the value of a potentially static expression of a modular type that appears in a nonstatic context must be within the base range of its expected type. Check that the predefined logical operators and membership tests are available.

B360001
Check that, within the definition of a nonlimited composite type or a limited composite type that becomes nonlimited later in its immediate scope, if a component definition contains the reserved word aliased and the type of the component is discriminated, the nominal subtype of the component may not be unconstrained.

B390001
Check that: Class wide objects are required to be initialized (whether created by object declaration or an allocator). Aggregates of a class wide type are required to be qualified with a specific type when their expected type is class-wide. Tagged private and tagged limited private require the full type to be a tagged record type. The attribute 'Class is not defined for untagged types. The Class attribute is defined for untagged private types whose full type is tagged, but only in the private part of the package in which the type is declared.

B391001
Check that: A discriminant on a tagged type is not allowed to have a default. Private record extension is not allowed to be declared immediately within a subprogram declarative region. Record extension of a nonlimited type does not allow limited components. A record extension may not be declared in a nested package where it is not accessible from the declaration of its parent type. Record extension does not allow repeating identifiers used in the parent declaration.

B391002
Check that a type extension may not be declared in a generic body if the parent type is declared outside that body.

B392001
Check that a default_expression for a controlling formal parameter of a dispatching operation may not be statically tagged. Check that a controlling formal parameter that is an access parameter may not have a default_expression.

B392002
Check that a subprogram may not be a dispatching operation for two distinct tagged types (in a package).

B392003
Check that: A dispatching operation which overrides an inherited subprogram is required to be subtype conformant with the inherited subprogram. The declaration of dispatching operations does not allow the use of subtypes which do not statically match the first subtype of the tagged type (in a package).

B392004
Check that: A dynamically tagged value is not allowed in an object or expression for which the expected type is a specific tagged value (unless it is a controlling operand on a dispatching operation). An access-to-classwide type is not allowed in an expression for which the expected type is an anonymous access to specific type (unless it is a controlling operand on a dispatching operation). A call on dispatching operation may not have both dynamically tagged and statically tagged controlling operands.

B392005
Check that a subprogram may not be a dispatching operation for two different tagged types (in a child unit package).

B392006
Check that a default_expression for a controlling formal parameter of a dispatching operation must

be tag indeterminate. Specifically, check that it may not be dynamically tagged.

B392007
Check that a dispatching operation declared in a child package which overrides an inherited subprogram declared in parent is required to be subtype conformant with the inherited subprogram.

B392008
Check that a subprogram call through a dereference of an access-to- subprogram value is not considered a call on a dispatching operation; therefore, the actual parameter in such a subprogram call may not be dynamically tagged. Check for the case where the access-to-subprogram type is a generic formal type.

B392009
Check that a subprogram call through a dereference of an access-to- subprogram value is not considered a call on a dispatching operation; therefore, the actual parameter in such a subprogram call may not be dynamically tagged. Check that a designated profile of an access-to-subprogram type which contains parameters of a tagged type does not introduce a primitive operation of the tagged type.

B393001
Check that: Objects and aggregates may not be defined or allocated of an abstract type. The type of a component may not be abstract. A function defined with an abstract result type must be declared abstract. If an abstract subprogram is defined as a primitive subprogram of a a tagged type, then the tagged type must be abstract. The full type of a non-abstract private extension may not be abstract. The full type of an abstract private extension may be non-abstract.

B393002
Check that incorrect orderings of reserved words in a tagged type declaration are flagged as illegal.

B393003
Check that: Bodies are not allowed for abstract subprograms. An abstract subprogram defined using a combination of concrete and abstract types remains abstract upon derivation from the concrete type. The target of an assignment operation may not be abstract.

B393004
Check that the actual subprogram corresponding to a generic formal subprogram must not be abstract.

B393006
Check that, if a non-abstract type is derived from an abstract formal private type within the generic declaration, an instantiation is rejected if the derived type inherits abstract primitive subprograms from the actual (parent) type.

B3A0001
Check that objects defined to be of a general access type may not designate an object or component which is not defined to be aliased. Check that a renaming of an aliased view is also defined to be aliased. Check that an array slice may not be aliased. Check that the general access modifiers "all" and "constant" are allowed. Check that an object designated by an access-to-constant type object cannot be updated through a value of that type. Check that an object designated by a value of an access-to-variable type can be both read and updated.

B3A0003
Check that a designated object cannot be updated through a value of an access-to-constant type. Check for the cases where the access-to- constant type is a generic formal type, or a non-formal type declared within a formal package.

B3A2002
Check that: 'Access is not defined for non-aliased objects. For X'Access of a general access type A, if A is an access-to-constant type, X can be either a constant or a variable. For X'Access of a general access type A, if A is an access-to-variable type, X must denote the view of a variable. Check for cases where X is a: (a) Formal in parameter of a tagged type. (b) Generic formal in parameter of a tagged type. (c) Formal in parameter of a composite type with aliased components. (d) Function

return value of a composite type with aliased components.

B3A2003
Check that, for X'Access of a general access type A, the accessibility level of the view denoted by X must not be statically deeper than that of the access type A. Check for cases where X is: (a) a view denoted by an object declaration. (b) a view denoted by a component definition. (c) a formal parameter of a tagged type.

B3A2004
Check that, for X'Access of a general access type A, the accessibility level of the view denoted by X must not be statically deeper than that of the access type A. Check for cases where X is: (a) a renaming of an aliased view. (b) a dereference of an access-to-object value. (c) a view conversion of an aliased view.

B3A2006
Check that, for P'Access of an access-to-subprogram type S, the accessibility level of the subprogram denoted by P must not be statically deeper than that of S.

B3A2007
Check that, for X'Access of a general access type A, the accessibility level of the view denoted by X must not be statically deeper than that of A. Check for cases where X'Access occurs in the visible part of an instance and X is declared in the instance itself. Check for cases where X is: (a) a view defined by an object declaration. (b) a renaming of an aliased view. (c) a view conversion of an aliased view.

B3A2009
Check that, for P'Access of an access-to-subprogram type S, if the subprogram denoted by P is declared within a generic body, S must also be declared within the generic body.

B3A2010
Check that, for P'Access of an access-to-subprogram type S, the accessibility level of the subprogram denoted by P must not be statically deeper than that of S. Check for cases where P'Access occurs in the visible and private part of an instance.

B460001
Check that if the target type of a type conversion is a general access type, the accessibility level of the operand type must not be statically deeper than that of the target type. Check for cases where the operand is: (a) a stand-alone access object. (b) a formal parameter. (c) an access discriminant.

B641001
Check that the actual parameter corresponding to a formal parameter of mode in out or out must denote a variable; in particular, that it may not be a dereference of an access-to-constant value. Check for the cases where the value is of a generic formal access-to-constant type, or of a non-formal access-to-constant type declared within a formal package.

B730001
Check that: Full type of a tagged private type must be a tagged type. This means that the full type must either be declared using a tagged record definition, or else derived from some other tagged type, in which case it must include a record_extension_part. Full type of a nonlimited tagged private type must be a nonlimited tagged type. Full type of a limited tagged private type must be a limited tagged type. A tagged record type must be a limited type if one of its record components is limited. A record extension must be extended from a limited parent type if one of its record components is limited.

B730002
Check that a private extension is limited if its ancestor type is limited. Check that if a partial view is nonlimited, the full view must be nonlimited. Check that if a partial view of a tagged type is limited, the full view must be limited, but that if a partial view of an untagged type is limited, the full view may be either limited or nonlimited. Check that the full view of a private extension must be derived, either directly or indirectly, from the ancestor type. Check that the ancestor type of a private extension must be a specific type.

B730003

Check that if the partial view of a private type is tagged, the full view must be tagged. Check that if the partial view of a private type is untagged, the full view may be tagged or untagged, but that if the partial view is untagged and the full view is tagged, no derivatives of the partial view are allowed within the immediate scope of the partial view. Check that derivatives of the full view are allowed.

B730004
Check that if a public child is "with"ed by a client, the client does not have visibility into the private part of the child's parent. Check that the full view of a private type defined in a parent and extended in a child is not visible outside the child.

B731A01
Check that the inherited primitive subprograms of a derived type definition are implicitly declared at the earliest place within the immediate scope of the type declaration (but after the type declaration) where the corresponding declaration from the parent is visible. Check that, within its scope, the full view determines which components are visible. Check for the cases where the parent is a partial view (tagged private type) declared in a package, and the derived type is declared in: the visible part of a public child unit a package nested within the visible part of a public child unit

B731A02
Check that the inherited primitive subprograms of a derived type are implicitly declared at the earliest place within the immediate scope of the type declaration (but after the type declaration) where the corresponding declaration from the parent is visible. Check that, within its scope, the full view determines which components are visible. Check for the cases where the parent is a partial view (tagged private type) declared in a package, and the derived type is declared in: the visible part of a private child unit a package nested within the visible part of a private child unit a non-child package, and is further derived from in a child unit a package nested within the visible part of a public child unit

B740001
Check that a deferred constant may be declared of any type and that, if it is completed by a full constant declaration, its completion must occur immediately within the private part of the same package. Check that the deferred and full constants must have the same type.

B840001
Check that the name in a use type clause must denote a subtype. Check that only the primitive operators of the type determined by the subtype mark in a use type clause are use-visible (in particular, that the primitive operators of no other type declared in the same package are use-visible). Check that the scope of a use type clause in the private part of a library unit does not include the visible part of any public descendant of that library unit.

B940001
Check that a protected_element_declaration within the private part of a protected type must be a component_declaration (if it is not a protected_operation_declaration). Specifically: an anonymous array is not allowed

B940002
Check that a protected_element_declaration within the private part of a protected type must be a component_declaration (if it is not a protected_operation_declaration). Specifically: a constant component is not allowed a type declaration is not allowed

B940003
Check that protected declarations (in a normal procedure) require completion by a protected body and vice versa.

B940004
Check that protected declarations (in a package) require completion by a protected body and vice versa.

B940005
Check the visibility of local subprograms and the private parts of protected objects

B940006

Check that component declarations are only allowed in the private part of protected objects

B940007
Check that component declarations are not allowed in the body of protected objects

B951001
Check that the body of a protected function cannot have an internal call to a protected procedure.

B952001
Check that the name that denotes the formal parameter of an entry body is not allowed within the entry barrier

B952002
Check that the body of a protected entry must have an entry barrier. Check that if an entry identifier appears at the end of an entry body it repeats the defining identifier of the entry or the entry family

B952003
Check that, in the body of a protected entry, the entry_index_specification must be enclosed in parentheses.

B954001
Check for error if requeue is not type conformant with the call or if requeue has parameters. Check requeues with/without abort.

B960001
Check that an argument to the delay_until_statement must have type Calendar.Time. In particular check that the delay_expressions of Duration, Float and Integer are flagged as errors

BA11001
Check that in the visible part of a public child, the private declarations of the parent package are not visible.

BA11002
Check that the private declarations of the parent are not visible for a formal parameter list or result type of a public child.

BA11003
Check that a child library unit may not have anything other than a library package or generic library package as its parent unit. Check that nested units cannot have child units. Check that child of a generic package may not be anything other than a generic unit or a renaming of some other child of the same generic unit. Check that a child of an instance of a generic package must be an instance or a renaming of a library unit.

BA11004
Check that a child library subprogram is not primitive subprogram (i.e, is not inherited by types derived from a type declared in the parent).

BA11005
Check that a parent body cannot declare a homograph of the child when a child unit is included in the context clause of the parent body.

BA11007
Check that a child library subprogram may not override a user-defined primitive subprogram.

BA11008
Check that an instance of a child of a generic package that is not part of a formal package declaration and that is a child of an instance of the generic package is not allowed outside the declarative region of the generic package itself. Check that an instance of a generic does not inherit children from the generic. Check that a child of an instance of a generic package must be an instance.

BA11009

Check that if the generic being renamed is itself a child of a generic package P, the renaming must occur in a place that is within the declarative region of P, which includes the body, the children (and descendant ...), and the subunits of P.

BA11010
Check that a library unit renaming declaration may not be used to rename a physically nested package, a physically nested subprogram, or a subunit.

BA12001
Check that the with-clause of a public child of some library unit cannot include a private child of the same ancestor.

BA12002
Check that the with-clause of a public second level descendant of some library unit cannot include a private descendant of the same ancestor.

BA12003
Check that the with-clause of the public descendant of a private descendant of a library unit cannot include any private descendants of its (immediate) parent.

BA12004
Check that a with-clause of a library unit may not include the private child or any descendant of a private child of some other library unit.

BA12005
Check that the with-clause in the body of a (public or private) descendant of a library unit cannot include a private child of a different library unit.

BA12007
Check that the rename of a child unit (i.e. a library unit with an expanded name) does not make declarations within ancestors of the child visible. Check that a parent unit name (in the defining declaration of a child unit) does not designate a renaming declaration.

BA12008
Check that a child unit may not be "with"ed using only its simple name. Check that a child unit may not be "with"ed using any abbreviated version of its full expanded name (e.g., grandparent.child rather than grandparent.parent.child)

BA13B01
Check that a separate subprogram declared in a private child unit of a public parent does not have visibility into the private part of the package on which its parent depends or the private part of its parent's public sibling.

BA13B02
Check that a separate subprogram declared in a public child unit of a private parent does not have visibility into the private part of the package on which its parent depends or the private part of its parent's public sibling.

BB10001
Check that separate exception handlers for Constraint_Error and Numeric_Error are not allowed within a handled sequence of statements.

BC30001
Check that, in the visible part of an instance, legality rules are enforced at compile time of the generic instantiation, and not enforced in other parts of the instance. Specifically, check that, for a tagged actual type passed to a non-tagged formal private type, a tagged type may not be derived from the actual in the visible part of an instance, but may be derived in the private part or body. Check that a non-tagged type derived from a tagged parent type in the private part of an instance is not treated as tagged outside the instance.

BC40001
Check that the type of a generic formal object of mode in must not be limited.

BC50001
Check that, for a generic formal derived type, the actual must be in the class rooted at the ancestor subtype. Check for scalar, array, and access types.

BC50002
Check that, for a generic formal derived type, the actual must be in the class rooted at the ancestor subtype. Check for record and tagged types.

BC51002
Check that if a generic formal derived subtype is definite, the actual subtype must not be indefinite. Check in cases where the formal subtype appears in contexts where an indefinite subtype would be legal.

BC51003
Check that, for a generic formal derived type with no discriminant part, if the ancestor subtype is constrained, the actual subtype must be constrained and must be statically compatible with the ancestor. Check for the case where both constraints are static and the actual subtype is defined by a subtype declaration.

BC51004
Check that, for a generic formal derived type with no discriminant part, if the ancestor subtype is constrained, the actual subtype must be constrained and must be statically compatible with the ancestor. Check for the case where both constraints are static and the actual subtype is defined by a derived type declaration.

BC51005
Check that, for a generic formal derived type with no discriminant part, if the ancestor subtype is an unconstrained access or record subtype, the actual subtype must be unconstrained.

BC51006
Check that, for a generic formal derived type with no discriminant part, if the ancestor subtype is an unconstrained array or tagged subtype, the actual subtype must be unconstrained.

BC51007
Check that, for a generic formal derived type with no discriminant part, if the ancestor subtype is an unconstrained discriminated subtype, the actual type must have the same number of discriminants, and each discriminant of the actual must correspond to a discriminant of the ancestor.

BC51011
Check that, for a formal private type with a known discriminant part, the subtype of each discriminant of the actual type must statically match the subtype of the corresponding discriminant of the formal type.

BC51012
Check that, if the reserved word "abstract" does not appear in the declaration of a formal derived type, the actual type must not be an abstract type. Check that, if the ancestor type is abstract, and the formal derived type is not, neither the ancestor type nor its abstract descendants may be passed as actuals. Check that, if the formal derived type is abstract, then the following entities that are of the formal type are illegal: a component, an object created by an object declaration or an allocator, a generic formal object of mode in, the the result type of a non-abstract function.

BC51013
Check that, if the reserved word "abstract" does not appear in the declaration of a formal private type, the actual type must not be an abstract type. Check that, if the formal private type is abstract, then the following entities that are of the formal type are illegal: a component, an object created by an object declaration or an allocator, a generic formal object of mode in, the result type of a non-abstract function.

BC51015
Check that if the actual type corresponding to a non-tagged formal private type is tagged, an instance is illegal if a (non-tagged) derived type is declared in the visible part. Check that an instance is legal if the derived type is declared in the private part or in the body.

BC51016
Check that, if the reserved word "abstract" appears in the declaration of a formal private type, the reserved word "tagged" must also appear. Check that, if the reserved word "abstract" appears in the declaration of a formal derived type, the reserved words "with private" must also appear. Check that a tagged type derived from a non-tagged generic formal private or derived type is illegal.

BC51017
Check that alternative orderings of reserved words in a formal private type declaration are illegal.

BC51018
Check that alternative orderings of reserved words in a formal (tagged) derived type declaration are illegal.

BC51019
Check that a generic formal derived tagged type is a private extension. Specifically, check that, for a generic formal derived type whose ancestor type has a primitive subprogram which is a function with a controlling result, the function must be overridden for non-abstract record extensions of the formal derived type. Check that the function need not be overridden for record extensions, nor for private extensions, although for non-abstract private extensions it must be overridden for the corresponding full type.

BC51020
Check that, for an abstract generic formal derived type whose ancestor type has an abstract primitive subprogram, non-abstract record and private extensions of the formal derived type must override the subprogram. Check that abstract record and private extensions need not override the subprogram. Check that, for a non-abstract generic formal derived type whose ancestor type has an abstract primitive subprogram, record and private extensions of the formal derived type need not override the subprogram.

BC51B01
Check that if a generic formal private subtype is definite, the actual subtype must not be indefinite, even if the formal subtype appears only in contexts where an indefinite subtype would be legal.

BC51B02
Check that the ancestor of a formal derived type may not be class- wide. Check that a formal derived type may not have a known discriminant part. Check that if a generic formal private or derived subtype is indefinite, it must not appear in a context which requires a definite subtype.

BC51C01
Check that the actual type passed to an abstract generic formal derived type may be either abstract or non-abstract, as may record and private extensions of the formal type. Check that, for a non-abstract type derived from an abstract formal derived type, all abstract primitive subprograms inherited from the actual type must be overridden in the instance.

BC51C02
Check that the actual type passed to an abstract generic formal private type may be either abstract or non-abstract, as may record and private extensions of the formal type. Check that, for a non-abstract type derived from an abstract formal private type, all abstract primitive subprograms inherited from the actual type must be overridden in the instance.

BC54001
Check that if a generic formal access type contains the general access modifier "constant," the actual must be an access-to-constant type. Check that if a generic formal access type contains the general access modifier "all," the actual must be a general access-to-variable type. Check that if a generic formal access type contains no general access modifiers and is not a formal access-to-subprogram type, the actual must be a general or pool-specific access-to-variable type. Check that if a generic formal access type is a formal access-to-subprogram type, the actual must be an access-to-subprogram type.

BC54002
Check that, for a formal access-to-subprogram subtype, the designated profiles of the formal and actual must be mode-conformant. Check that if the calling convention of the formal is not protected, the calling convention of the actual must not be protected.

BC54003
Check that, for a formal access-to-subprogram subtype, the corresponding parameter and result types of the designated profiles of the formal and actual must be the same. Specifically, check for the case where the parameters in the profile of the formal type are themselves formal types.

BC54A01
Check that, for a formal access-to-subprogram subtype whose profile contains access parameters, the designated subtypes of the corresponding access parameters in the formal and actual profiles must statically match. Check cases where the designated subtype is an elementary subtype.

BC54A02
Check that, for a formal access-to-subprogram subtype whose profile contains access parameters, the designated subtypes of the corresponding access parameters in the formal and actual profiles must statically match. Check cases where the designated subtype is a composite subtype.

BC54A03
Check that, for a formal access-to-subprogram subtype whose profile contains access parameters, the designated subtypes of the corresponding access parameters in the formal and actual profiles must statically match. Check cases where the designated subtype is a generic formal subtype.

BC54A04
Check that, for a formal access-to-object type, the designated subtypes of the formal and actual must statically match. Check for the case where the access-to-object type is a general access-to-constant type.

BC54A05
Check that, for a formal access-to-object type, the designated subtypes of the formal and actual must statically match. Check for the case where the access-to-object type is a general access-to-variable type.

BC54A06
Check that, for a formal access-to-object type, the designated subtypes of the formal and actual must statically match. Check for the case where the access-to-object type is a pool-specific access-to-variable type.

BC70001
Check that the actual corresponding to a generic formal package must be an instance of the template for the formal package. Check for the case where the formal package is declared in a library- level generic package.

BC70002
Check that the actual corresponding to a generic formal package must be an instance of the template for the formal package. Check for the case where the formal package is declared in a library- level generic subprogram.

BC70003
Check that the template in a formal package declaration must be a generic package. Check for the case where the formal package is declared in a library-level generic package.

BC70004
Check that the template in a formal package declaration must be a generic package. Check for the case where the formal package is declared in a library-level generic subprogram.

BC70005
Check that if a formal package actual part is not (<>), the generic formal part of the template is not part of the visible part of the formal package. Check for the case where the formal package is declared in a library-level generic package.

BC70006
Check that if a formal package actual part is not (<>), the generic formal part of the template is not part of the visible part of the formal package. Check for the case where the formal package is declared in a library-level generic subprogram.

BC70007
Check that an actual instance of a generic formal package is rejected if its actuals do not match the corresponding actuals in the formal package actual part. Specifically, check that the following cases are illegal: For a formal object of mode IN: The actuals are both static expressions but do not have the same value. The actuals are not both static expressions and do not statically denote the same constant. The actuals are not both the literal null.

BC70008
Check that the actual corresponding to a generic formal package must be an instance of the template for the formal package. Check for the case where the formal package is declared in a library- level generic subprogram. Check for the case where the actuals have been renamed. Check that a generic renaming declaration which renames the template may be used in instantiations of the template.

BC70009
Check that an actual instance of a generic formal package is rejected if its actuals do not match the corresponding actuals in the formal package actual part. Specifically, check that, for formal subprograms and packages, the actuals must statically denote the same entity.

BC70010
Check that an actual instance of a generic formal package is rejected if its actuals do not match the corresponding actuals in the formal package actual part. Specifically, check that, for formal subtypes, the actuals must denote statically matching subtypes.

BDD2001
Check that Stream_IO attributes 'Input, 'Output, 'Class'Input, and 'Class'Output cannot be used with limited types, including composite types containing limited components.

BXA8001
Check that Append_File mode has not been added to package Direct_IO. APPLICABILITY CRITERIA: Applicable to all implementations that support Direct_IO operations.

BXAC001
Check that a stream is limited and may not be the target of an assignment.

BXAC002
Check that the Set_Position procedure and Position function are not predefined in Stream_IO. Check that the type File_Offset is not predefined in Stream_IO. Check that the Set_Index procedure and Index function are predefined in Stream_IO. Check that the type Positive_Count is predefined in Stream_IO. Check that the appropriate parameter types are required for the Stream_IO procedure Set_Index.

BXAC003
Check that an attempt to use the 'Write or 'Read type attribute values to write or read a Stream_IO file is rejected when a stream file object is provided as the parameter, rather than an stream access value. Check that the correct type 'Write or 'Read attribute value is required when writing or reading data to/from a stream. Check that an attempt to use the 'Write or 'Read type attribute values as attributes of an object rather than a type are rejected by the compiler.

BXAC004
Check that an attribute reference for the Stream_IO attributes 'Write and 'Read is illegal if the type is limited, including task types and composite types containing limited components.

BXAC005
Check that Text_IO.File_Type objects cannot be used in conjunction with stream-oriented attributes 'Write and 'Read. Check that Streams.Stream_IO.File_Type objects cannot be used in Text_IO file data transfer operations. Check that stream access objects cannot be used as file object parameters of Text_IO.Put and Text_IO.Get procedures. Check that Put and Get are not defined as type attributes for use with stream files. Check that the package Stream_Support, which was originally defined in the 9X Mapping Specification and Ada 9X ILS, but which has been changed to package Streams in AARM;3.0, is not included in the compilation system predefined library. (Note: This portion of the objective can be deleted in the future.)

BXC3001
Check that pragmas Interrupt_Handler and Attach_Handler are recognized. Check that the handler is a parameterless protected procedure; check that the pragmas are allowed only immediately in a protected definition. Check that a protected declaration for Interrupt_Handler is library level. Check that a protected type declaration for Attach_Handler is library level and that any object declaration of that type is library level. Check that Attach_Handler will accept an expression only of type Interrupts.Interrupt_ID.

BXC5001
Check that pragma Discard_Names may only be declared immediately within a declarative part, immediately within a package specification or as a configuration pragma. Check that its parameter, if present, may denote only a non-derived enumeration subtype, tagged subtype or an exception.

BXC6001
Check that the name referenced in pragmas Atomic and Volatile may only be an object, a non-inherited component or a full type. Check that the name referenced in Atomic_Components or Volatile_Components must be an array type or an object of an anonymous array type.

BXE2A01
Check that a Declared Pure library unit can depend only on other Declared Pure library units. Specifically, it can not depend on a Shared Passive Unit.

BXE2A02
Check that a Declared Pure library unit can depend only on other Declared Pure library units. Specifically it can not depend on a Remote Types unit.

BXE2A03
Check that a Declared Pure library unit can depend only on other Declared Pure library units. Specifically it cannot depend on an Normal unrestricted unit.

BXE2A04
Check that a Shared Passive library unit can depend only on other Shared Passive or Declared Pure library units. Specifically that it can not depend on a Remote Types library unit.

BXE2A05
Check that a Shared Passive library unit can depend only on other Shared Passive library units or Declared Pure library units. Specifically that it can not depend on a Normal unrestricted unit.

BXE2A06
Check that a Remote Types library unit can depend only on other Remote Types library units, Declared Pure library units or Shared Passive library units. Specifically that it cannot depend on a Normal unrestricted unit

BXF1001
Check that values of 2 and 10 are allowable values for Machine_Radix of a decimal first subtype. Check that values other than 2 and 10 are not allowed for Machine_Radix of a decimal first subtype. Check that the expression used to define Machine_Radix must be static. Check that the package Ada.Decimal is available. Check that 10**(-Max_Scale) is allowed as a decimal type's delta. Check that 10**(-Min_Scale) is allowed as a decimal type's delta. Check that Min_Delta and Max_Delta are allowed for delta in decimal fixed point definitions. Check that Max_Decimal_Digits is allowed for digits in a decimal fixed point definition. Check that a value N larger than Max_Scale is not allowed in the expression 10**(-N) as a decimal type's delta. Check that a value N smaller than Min_Scale is not allowed in the expression 10**(-N) as a decimal type's delta. Check that neither a value smaller than Min_Delta nor a value larger than Max_Delta are allowed for delta in decimal fixed point definitions. Check that a value larger than Max_Decimal_Digits is not allowed for digits in a decimal fixed point definition.

C340001
Check that user-defined equality operators are inherited by a derived type except when the derived type is a nonlimited record extension. In the latter case, ensure that the primitive equality operation of the record extension compares any extended components according to the predefined equality operators of the component types. Also check that the parent portion of the extended type is compared using the user-defined equality operation of the parent type.

C340A01
Check that a tagged type declared in a package specification may be passed as a generic formal (tagged) private type to a generic package declaration. Check that the formal type may be extended with a record extension in the generic package. Check that, in the instance, the record extension inherits the user-defined primitive subprograms of the tagged actual.

C340A02
Check that a record extension (declared in a package specification) of a tagged type (declared in a different package specification) may be passed as a generic formal (tagged) private type to a generic package declaration. Check that the formal type may be further extended with a record extension in the generic package. Check that, in the instance, the record extension inherits the user-defined primitive subprograms of the tagged actual, including those inherited by the actual from its parent.

C341A01
Check that formal parameters of a class-wide type can be passed values of any specific type within the class.

C341A02
Check that class-wide objects can be reassigned with objects from the same specific type used to initialize them.

C341A03
Check that an object of one class-wide type can initialize a class-wide object of a different type when the operation is embedded in a generic unit.

C341A04
Check that class-wide objects can be initialized using allocation.

C3900010
Check that a record extension can be declared in the same package as its parent, and that this parent may be a tagged record or a record extension. Check that each derivative inherits all user-defined primitive subprograms of its parent (including those that its parent inherited), and that it may declare its own primitive subprograms. Check that predefined equality operators are defined for the root tagged type. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900011
Check that a record extension can be declared in the same package as its parent, and that this parent may be a tagged record or a record extension. Check that each derivative inherits all user-defined primitive subprograms of its parent (including those that its parent inherited), and that it may declare its own primitive subprograms. Check that predefined equality operators are defined for the root tagged type. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390002
Check that a tagged base type may be declared, and derived from in simple, private and extended forms. (Overlaps with C390B04) Check that the package Ada.Tags is present and correctly implemented. Check for the correct operation of Expanded_Name, External_Tag and Internal_Tag within that package. Check that the exception Tag_Error is correctly raised on calling Internal_Tag with bad input.

C390003
Check that for a subtype S of a tagged type T, S'Class denotes a class-wide subtype. Check that T'Tag denotes the tag of the type T, and that, for a class-wide tagged type X, X'Tag denotes the tag of X. Check that the tags of stand alone objects, record and array components, aggregates, and formal parameters identify their type. Check that the tag of a value of a formal parameter is that of the

actual parameter, even if the actual is passed by a view conversion.

C390004
Check that the tags of allocated objects correctly identify the type of the allocated object. Check that the tag corresponds correctly to the value resulting from both normal and view conversion. Check that the tags of accessed values designating aliased objects correctly identify the type of the object. Check that the tag of a function result correctly evaluates. Check this for class-wide functions. The tag of a class-wide function result should be the tag appropriate to the actual value returned, not the tag of the ancestor type.

C3900050
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900051
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900052
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900053
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900060
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a record extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900061
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a record extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900062
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a record extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C3900063
Check that a private tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a record extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A010
Check that a nonprivate tagged type declared in a package specification may be extended with a record extension in a different package specification, and that this record extension may in turn be extended by a record extension. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A011
Check that a nonprivate tagged type declared in a package specification may be extended with a record extension in a different package specification, and that this record extension may in turn be extended by a record extension. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A020
Check that a nonprivate tagged type declared in a package specification may be extended with a record extension in a different package specification, and that this record extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A021
Check that a nonprivate tagged type declared in a package specification may be extended with a record extension in a different package specification, and that this record extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A022
Check that a nonprivate tagged type declared in a package specification may be extended with a record extension in a different package specification, and that this record extension may in turn be extended by a private extension in a third package. Check that each derivative inherits the user-

defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A030
Check that a nonprivate tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a private extension. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C390A031
Check that a nonprivate tagged type declared in a package specification may be extended with a private extension in a different package specification, and that this private extension may in turn be extended by a private extension. Check that each derivative inherits the user-defined primitive subprograms of its parent (including those that its parent inherited), that it may override these inherited primitive subprograms, and that it may also declare its own primitive subprograms. Check that predefined equality operators are defined for the tagged type and its derivatives. Check that type conversion is defined from a type extension to its parent, and that this parent itself may be a type extension.

C391001
Check that structures nesting discriminated records as components in record extension are correctly supported. Check for this using limited private structures. Check that record extensions inherit all the visible components of their ancestor types. Check that discriminants are correctly inherited.

C391002
Check that structures nesting discriminated records as components in record extension are correctly supported. Check that record extensions inherit all the visible components of their ancestor types. Check that discriminants are correctly inherited.

C3920020
Check that the use of a class-wide formal parameter allows for the proper dispatching of objects to the appropriate implementation of a primitive operation. Check this in the case where the root tagged type is defined in a generic package, and the type derived from it is defined in that same generic package.

C3920021
Check that the use of a class-wide formal parameter allows for the proper dispatching of objects to the appropriate implementation of a primitive operation. Check this in the case where the root tagged type is defined in a generic package, and the type derived from it is defined in that same generic package.

C392003
Check that the use of a class-wide formal parameter allows for the proper dispatching of objects to the appropriate implementation of a primitive operation. Check this where the root tagged type is defined in a package, and the extended type is defined in a nested package.

C392004
Check that subprograms inherited from tagged derivations, which are subsequently redefined for the derived type, are available to the package defining the new class via view conversion. Check that operations performed on objects using view conversion do not affect the extended fields. Check that visible operations not masked by the deriving package remain available to the client, and do not affect the extended fields.

C392005
Check that, for an implicitly declared dispatching operation that is overridden, the body executed is the body for the overriding subprogram, even if the overriding occurs in a private part. Check for the

case where the overriding operations are declared in a public child unit of the package declaring the parent type, and the descendant type is a private extension. Check for both dispatching and nondispatching calls.

C392008
Check that the use of a class-wide formal parameter allows for the proper dispatching of objects to the appropriate implementation of a primitive operation. Check this for the case where the root tagged type is defined in a package and the extended type is defined in a dependent package.

C392A01
Check that the use of a class-wide formal parameter allows for the proper dispatching of objects to the appropriate implementation of a primitive operation. Check this for the root tagged type defined in a package, and the extended type is defined in that same package.

C392C05
Check that for a call to a dispatching subprogram the subprogram body which is executed is determined by the controlling tag for the case where the call has statically tagged controlling operands of the type T. Check this for various operands of tagged types: objects (declared or allocated), formal parameters, view conversions, function calls (both primitive and non-primitive).

C392C07
Check that for a call to a dispatching subprogram the subprogram body which is executed is determined by the controlling tag for the case where the call has dynamic tagged controlling operands of the type T. Check for calls to these same subprograms where the operands are of specific statically tagged types: objects (declared or allocated), formal parameters, view conversions, and function calls (both primitive and non-primitive).

C392D01
Check that, for an implicitly declared dispatching operation that is overridden, the body executed is the body for the overriding subprogram, even if the overriding occurs in a private part. Check that, for an implicitly declared dispatching operation that is NOT overridden, the body executed is the body of the corresponding subprogram of the parent type. Check for the case where the overriding (and non-overriding) operations are declared for a private extension (and its full type) in a public child unit of the package declaring the ancestor type, and the ancestor type is a tagged private type whose full view is itself a derived type.

C392D02
Check that a primitive procedure declared in a private part is not overridden by a procedure explicitly declared at a place where the primitive procedure in question is not visible. Check for the case where the non-overriding operation is declared in a separate (non-child) package from that declaring the parent type, and the descendant type is a record extension.

C392D03
Check that, for an inherited dispatching operation that is overridden, the body executed is the body of the overriding subprogram, even if the overriding occurs in a private part. Check for the case where the overriding operation is declared in a separate (non-child) package from that declaring the parent type, and the descendant type is a record extension. Check for both dispatching and nondispatching calls.

C393001
Check that an abstract type can be declared, and in turn concrete types can be derived from it. Check that the definition of actual subprograms associated with the derived types dispatch correctly.

C393007
Check that an extended type can be derived from an abstract type, where the abstract type is defined in a package, and the type derived from it is defined in a distinct library package.

C393008
Check that an extended type can be derived from an abstract type.

C393009
Check that an extended type can be derived from an abstract type.

C393011
Check that an abstract extended type can be derived from an abstract type, and that a a non-abstract type may then be derived from the second abstract type.

C393012
Check that a non-abstract subprogram of an abstract type can be called with a controlling operand that is a type conversion to the abstract type. Check that converting to the class-wide type of an abstract type inside an operation of that type causes a "redispatch" of the called operation.

C393A02
Check that a dispatching call to an abstract subprogram invokes the correct subprogram body of a descendant type according to the controlling tag. Check that a subprogram can be declared with formal parameters and result that are of an abstract type's associated class-wide type and that such subprograms can be called. 3.4.1(4)

C393A03
Check that a non-abstract primitive subprogram of an abstract type can be called as a dispatching operation and that the body of this subprogram can make a dispatching call to an abstract operation of the corresponding abstract type.

C393A05
Check that for a nonabstract private extension, any inherited abstract subprograms can be overridden in the private part of the immediately enclosing package and that calls can be made to private dispatching operations.

C393A06
Check that a type that inherits abstract operations but overrides each of these operations is not required to be abstract, and that objects of the type and its class-wide type may be declared and passed in calls to the overriding subprograms.

C393B12
Check that an extended type can be derived in the specification of a generic package when the parent is an abstract type in a library package.

C393B13
Check that an extended type can be derived from an abstract type when that derivation is declared in a child package.

C393B14
Check that an extended type can be derived in a private child package from an abstract type defined in a library package.

C3A0001
Check that access to subprogram type can be used to select and invoke functions with appropriate arguments dynamically.

C3A0002
Check that access to subprogram type can be used to select and invoke procedures with appropriate arguments dynamically.

C3A0003
Check that a function in a generic instance can be called using an access-to-subprogram value.

C3A0004
Check that access to subprogram may be stored within array objects, and that the access to subprogram can subsequently be called.

C3A0005
Check that access to subprogram may be stored within record objects, and that the access to subprogram can subsequently be called.

C3A0006

Check that access to subprogram may be stored within data structures, and that the access to subprogram can subsequently be called.

C3A0007
Check that a call to a subprogram via an access-to-subprogram value stored in a data structure will correctly dispatch according to the tag of the class-wide parameter passed via that call.

C3A0008
Check that subprogram references may be passed as parameters using access-to-subprogram types. Check that the passed subprograms may be invoked from within the called subprogram.

C3A0009
Check that subprogram references may be passed as parameters using access-to-subprogram types. Check that the passed subprograms may be invoked from within the called subprogram.

C3A0010
Check that an access-to-subprogram type in a generic instance may be used to declare access-to-subprogram objects which invoke subprograms in the instance.

C3A0011
Check that an access-to-subprogram object whose type is declared in a parent package, may be used to invoke subprograms in a child package. Check that such access objects may be stored in a data structure and that subprograms may be called by walking the data structure.

C3A00120
See file C3A00122.AM

C3A00121
See file C3A00122.AM

C3A00122
Check that an access-to-subprogram object can be used to invoke a subprogram when the subprogram body had been declared and implemented as a subunit.

C3A0013
Check that a general access type object may reference allocated pool objects as well as aliased objects. (3,4) Check that formal parameters of tagged types are implicitly defined as aliased; check that the 'Access of these formal parameters designates the correct object with the correct tag. (5) Check that the current instance of a limited type is defined as aliased. (5)

C3A0014
Check that if the view defined by an object declaration is aliased, and the type of the object has discriminants, then the object is constrained by its initial value even if its nominal subtype is unconstrained. Check that the attribute A'Constrained returns True if A is a formal out or in out parameter, or dereference thereof, and A denotes an aliased view of an object.

C3A2001
Check that an access type may be defined to designate the class-wide type of an abstract type. Check that the access type may then be used subsequently with types derived from the abstract type. Check that dispatching operations dispatch correctly, when called using values designated by objects of the access type.

C3A2002
Check that, for X'Access of a general access type A, Program_Error is raised if the accessibility level of X is deeper than that of A. Check for the case where X denotes a view that is a dereference of an access parameter, or a rename thereof. Check for cases where the actual corresponding to X is: (a) An allocator. (b) An expression of a named access type. (c) Obj'Access.

C3A2003
Check that, for X'Access of a general access type A, Program_Error is raised if the accessibility level of X is deeper than that of A. Check for the case where X denotes a view that is a dereference of an access parameter, or a rename thereof. Check for the case where X is an access parameter and the corresponding actual is another access parameter.

C431001
Check that a record aggregate can be given for a nonprivate, nonlimited record extension and that the tag of the aggregate values are initialized to the tag of the record extension.

C432001
Check that extension aggregates may be used to specify values for types that are record extensions. Check that the type of the ancestor expression may be any nonlimited type that is a record extension, including private types and private extensions. Check that the type for the aggregate is derived from the type of the ancestor expression.

C432002
Check that if an extension aggregate specifies a value for a record extension and the ancestor expression has discriminants that are inherited by the record extension, then a check is made that each discriminant has the value specified. Check that if an extension aggregate specifies a value for a record extension and the ancestor expression has discriminants that are not inherited by the record extension, then a check is made that each such discriminant has the value specified for the corresponding discriminant. Check that the corresponding discriminant value may be specified in the record component association list or in the derived type definition for an ancestor. Check the case of ancestors that are several generations removed. Check the case where the value of the discriminant(s) in question is supplied several generations removed. Check the case of multiple discriminants. Check that Constraint_Error is raised if the check fails.

C432003
Check that if the type of the ancestor part of an extension aggregate has discriminants that are not inherited by the type of the aggregate, and the ancestor part is a subtype mark that denotes a constrained subtype, Constraint_Error is raised if: 1) any discriminant of the ancestor has a different value than that specified for a corresponding discriminant in the derived type definition for some ancestor of the type of the aggregate, or 2) the value for the discriminant in the record association list is not the value of the corresponding discriminant. Check that the components of the value of the aggregate not given by the record component association list are initialized by default as for an object of the ancestor type.

C432004
Check that the type of an extension aggregate may be derived from the type of the ancestor part through multiple record extensions. Check for ancestor parts that are subtype marks. Check that the type of the ancestor part may be abstract.

C452001
For a type extension, check that predefined equality is defined in terms of the primitive equals operator of the parent type and any tagged components of the extension part. For other composite types, check that the primitive equality operator of any matching tagged components is used to determine equality of the enclosing type. For private types, check that predefined equality is defined in terms of the user-defined (primitive) operator of the full type if the full type is tagged. The partial view of the type may be tagged or untagged. Check that predefined equality for a private type whose full view is untagged is defined in terms of the predefined equality operator of its full type.

C460001
Check that if the target type of a type conversion is a general access type, Program_Error is raised if the accessibility level of the operand type is deeper than that of the target type. Check for the case where the operand is an access parameter. Check for cases where the actual corresponding to the access parameter is: (a) An allocator. (b) An expression of a named access type. (c) Obj'Access.

C460002
Check that if the target type of a type conversion is a general access type, Program_Error is raised if the accessibility level of the operand type is deeper than that of the target type. Check for the case where the operand is an access parameter, and the actual corresponding to the access parameter is another access parameter.

C640001
Check that the prefix of a subprogram call with an actual parameter part may be an implicit dereference of an access-to-subprogram value. Check that, for an access-to-subprogram type whose designated profile contains parameters of a tagged generic formal type, an access-to- subprogram

value may designate dispatching and non-dispatching operations, and that dereferences of such a value call the appropriate subprogram.

C730001
Check that the full view of a private extension may be derived indirectly from the ancestor type (i.e., the parent type of the full type may be any descendant of the ancestor type). Check that, for a primitive subprogram of the private extension that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions come from the corresponding primitive subprogram of the ancestor type, while the body comes from that of the parent type. Check both dispatching and non-dispatching cases.

C730002
Check that the full view of a private extension may be derived indirectly from the ancestor type (i.e., the parent type of the full type may be any descendant of the ancestor type). Check that, for a primitive subprogram of the private extension that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions come from the corresponding primitive subprogram of the ancestor type, while the body comes from that of the parent type. Check for a case where the parent type is derived from the ancestor type through a series of types produced by generic instantiations. Examine both the static and dynamic binding cases.

C730003
Check that the characteristics of a type derived from a private extension (outside the scope of the full view) are those defined by the partial view of the private extension. In particular, check that a component of the derived type may be explicitly declared with the same name as a component declared for the full view of the private extension. Check that a component defined in the private extension of a type may be updated through a view conversion of a type derived from the type.

C730A01
Check that a tagged type declared in a package specification may be passed as a generic formal (tagged) private type to a generic package declaration. Check that the formal type may be extended with a private extension in the generic package. Check that, in the instance, the private extension inherits the user-defined primitive subprograms of the tagged actual.

C730A02
Check that a private extension (declared in a package specification) of a tagged type (declared in a different package specification) may be passed as a generic formal (tagged) private type to a generic package declaration. Check that the formal type may be further extended with a private extension in the generic package. Check that the (visible) components inherited by the "generic" extension are visible outside the generic package. Check that, in the instance, the private extension inherits the user-defined primitive subprograms of the tagged actual, including those inherited by the actual from its parent.

C760001
Check that Initialize is called for objects and components of a controlled type when the objects and components are not assigned explicit initial values. Check this for "simple" controlled objects, controlled record components and arrays with controlled components. Check that if an explicit initial value is assigned to an object or component of a controlled type then Initialize is not called.

C760002
Check that assignment to an object of a (non-limited) controlled type causes the Adjust operation of the type to be called. Check that Adjust is called after copying the value of the source expression to the target object. Check that Adjust is called for all controlled components when the containing object is assigned. (Test this for the cases where the type of the containing object is controlled and noncontrolled; test this for initialization as well as assignment statements.) Check that for an object of a controlled type with controlled components, Adjust for each of the components is called before the containing object is adjusted. Check that an Adjust procedure for a Limited_Controlled type is not called by the implementation.

C760007
Check that Adjust is called for the execution of a return statement for a function returning a result of a (non-limited) controlled type. Check that Adjust is called when evaluating an aggregate component association for a controlled component. Check that Adjust is called for the assignment of the ancestor expression of an extension aggregate when the type of the aggregate is controlled.

C761001
Check that controlled objects declared immediately within a library package are finalized following the completion of the environment task (and prior to termination of the program).

C761002
Check that objects of a controlled type that are created by an allocator are finalized at the appropriate time. In particular, check that such objects are not finalized due to completion of the master in which they were allocated if the corresponding access type is declared outside of that master. Check that Unchecked_Deallocation of a controlled object causes finalization of that object.

C761003
Check that an object of a controlled type is finalized when the enclosing master is complete. Check this for controlled types where the derived type has a discriminant. Check this for subprograms of abstract types derived from the types in Ada.Finalization. Check that finalization of controlled objects is performed in the correct order. In particular, check that if multiple objects of controlled types are declared immediately within the same declarative part then type are finalized in the reverse order of their creation.

C761004
Check that an object of a controlled type is finalized with the enclosing master is complete. Check that finalization occurs in the case where the master is left by a transfer of control. Specifically check for types where the derived types do not have discriminants. Check that finalization of controlled objects is performed in the correct order. In particular, check that if multiple objects of controlled types are declared immediately within the same declarative part then they are finalized in the reverse order of their creation.

C761005
Check that deriving abstract types from the types in Ada.Finalization does not negatively impact the implicit operations. Check that an object of a controlled type is finalized when the enclosing master is complete. Check that finalization occurs in the case where the master is left by a transfer of control. Check this for controlled types where the derived type has a discriminant. Check this for cases where the type is defined as private, and the full type is derived from the types in Ada.Finalization. Check that finalization of controlled objects is performed in the correct order. In particular, check that if multiple objects of controlled types are declared immediately within the same declarative part then type are finalized in the reverse order of their creation.

C840001
Check that, for the type determined by the subtype mark of a use type clause, the declaration of each primitive operator is use-visible within the scope of the clause, even if explicit operators with the same names as the type's operators are declared for the subtype. Check that a call to such an operator executes the body of the type's operation.

C854001
Check that a subprogram declaration can be completed by a subprogram renaming declaration. In particular, check that such a renaming-as-body can be given in a package body to complete a subprogram declared in the package specification. Check that calls to the subprogram invoke the body of the renamed subprogram. Check that a renaming allows a copy of an inherited or predefined subprogram before overriding it later. Check that renaming a dispatching operation calls the correct body in case of overriding.

C910001
Check that tasks may have discriminants. Specifically, check where the subtype of the discriminant is a discrete subtype and where it is an access subtype. Check the case where the default values of the discriminants are used.

C940001
Check that a protected object provides coodinated access to shared data. Check that it can be used to sequence a number of tasks. Use the protected object to control a single token for which three tasks compete. Check that only one task is running at a time and that all tasks get a chance to run sometime.

C940002
Check that a protected object provides coordinated access to shared data. Check that it can implement a semaphore-like construct using a parameterless procedure which allows a specific maximum number of tasks to run and excludes all others

C940005
Check that the body of a protected function can have internal calls to other protected functions and that the body of a protected procedure can have internal calls to protected procedures and to protected functions.

C940006
Check that the body of a protected function can have external calls to other protected functions and that the body of a protected procedure can have external calls to protected procedures and to protected functions.

C940007
Check that the body of a protected function declared as an object of a given type can have internal calls to other protected functions and that a protected procedure in such an object can have internal calls to protected procedures and to protected functions.

C940010
Check that if an exception is raised during the execution of an entry body it is propagated back to the caller

C940011
Check that, in the body of a protected object created by the execution of an allocator, external calls to other protected objects via the access type are correctly performed

C940012
Check that a protected object can have discriminants

C940013
Check that items queued on a protected entry are handled FIFO and that the 'count attribute of that entry reflects the length of the queue.

C940A03
Check that a protected object provides coordinated access to shared data. Check that it can implement a semaphore-like construct controlling access to shared data through procedure parameters to allow a specific maximum number of tasks to run and exclude all others.

C951001
Check that two procedures in a protected object will not be executed concurrently.

C951002
Check that an entry and a procedure within the same protected object will not be executed simultaneously.

C954001
Check that a requeue statement within an entry_body with parameters may requeue the entry call to a protected entry with a subtype- conformant parameter profile. Check that, if the call is queued on the new entry's queue, the original caller remains blocked after the requeue, but the entry_body containing the requeue is completed.

C954010
Check that a requeue within an accept statement does not block. This test uses: Requeue to an entry in a different task Parameterless call Requeue with abort

C954011
Check that a requeue is placed on the correct entry; that the original caller waits for the completion of the requeued rendezvous; that the original caller continues after the rendezvous. Specifically, this test checks requeue to an entry in a different task, requeue where the entry has parameters, and requeue with abort.

C954012
Check a requeue within an accept body to another entry in the same task Specifically, check a call with parameters and a requeue with abort.

C954013
Check that a requeue is cancelled and that the requeuing task is unaffected when the calling task is aborted. Specifically, check requeue to an entry in a different task, requeue where the entry has parameters, and requeue with abort.

C954014
Check that a requeue is not canceled and that the requeueing task is unaffected when a calling task is aborted. Check that the abort is deferred until the entry call is complete. Specifically, check requeue to an entry in a different task, requeue where the entry call has parameters, and requeue without the abort option.

C954015
Check that requeued calls to task entries may, in turn, be requeued. Check that the intermediate requeues are not blocked and that the original caller remains blocked until the last requeue is complete. This test uses: Call with parameters Requeue with abort

C954016
Check that when a task that is called by a requeue is aborted, the original caller receives Tasking_Error and the requeuing task is unaffected.

C954017
Check that when an exception is raised in the rendezvous of a task that was called by a requeue the exception is propagated to the original caller and that the requeuing task is unaffected.

C954018
Check that if a task is aborted while a requeued call is queued on one of its entries the original caller receives Tasking_Error and the requeuing task is unaffected. This test uses: Requeue to an entry in a different task Parameterless call Requeue with abort

C954019
Check that when a requeue is to the same entry the items go to the right queue and that they are placed back on the end of the queue.

C954020
Check that a call to a protected entry can be requeued to a task entry. Check that the requeue is placed on the correct entry; that the original caller waits for the completion of the requeue and continues after the requeued rendezvous. Check that the requeue does not block. Specifically, check a requeue with abort from a protected entry to an entry in a task.

C954021
Check that a requeue within a protected entry to an entry in a different protected object is queued correctly.

C954022
In an entry body requeue the call to the same entry. Check that the items go to the right queue and that they are placed back on the end of the queue

C954023
Check that a requeue within a protected entry to a family of entries in a different protected object is queued correctly Call with parameters Requeue with abort

C954024
Check that a call to a protected entry can be requeued to a task entry. Check that the requeue is placed on the correct entry; that the original caller waits for the completion of the requeue and continues after the requeued rendezvous. Check that the requeue does not block. Specifically, check a requeue without abort from a protected entry to an entry in a task.

C954A01
Check that if a task requeued without abort on a protected entry queue is aborted, the abort is

deferred until the entry call completes, after which the task becomes completed.

C954A02
Check that if a task requeued with abort on a protected entry queue is aborted, the protected entry call is canceled and the aborted task becomes completed.

C954A03
Check that a requeue statement in an accept_statement with parameters may requeue the entry call to a protected entry with no parameters. Check that, if the call is queued on the new entry's queue, the original caller remains blocked after the requeue, but the accept_statement containing the requeue is completed. Note that this test uses a requeue "with abort," although it does not check that such a requeued caller can be aborted; that feature is tested elsewhere.

C960001
Confirm that a simple Delay Until statement is performed. Check that the delay does not complete before the requested time and that it does complete thereafter

C960002
Check that the simple "delay until" when the request time is "now" and also some time already in the past is obeyed and returns immediately

C960004
With the triggering statement being a delay and with the Asynchronous Select statement being in a tasking situation complete the abortable part before the delay expires. Check that the delay is cancelled and that the optional statements in the triggering part are not executed.

C974001
Check that the abortable part of an asynchronous select statement is aborted if it does not complete before the triggering statement completes, where the triggering statement is a delay_relative statement and check that the sequence of statements of the triggering alternative is executed after the abortable part is left.

C974002
Check that the sequence of statements of the triggering alternative of an asynchronous select statement is executed if the triggering statement is a delay_until statement, and the specified time has already passed. Check that the abortable part is not executed after the sequence of statements of the triggering alternative is left. Check that the sequence of statements of the triggering alternative of an asynchronous select statement is not executed if the abortable part completes before the triggering statement, and the triggering statement is a delay_until statement.

C974003
Check that the abortable part of an asynchronous select statement is aborted if it does not complete before the triggering statement completes, where the triggering statement is a task entry call, and the entry call is queued. Check that the sequence of statements of the triggering alternative is executed after the abortable part is left.

C974004
Check that the abortable part of an asynchronous select statement is aborted if it does not complete before the triggering statement completes, where the triggering statement is a task entry call, the entry call is queued, and the entry call completes by propagating an exception and that the sequence of statements of the triggering alternative is not executed after the abortable part is left and that the exception propagated by the entry call is re-raised immediately following the asynchronous select.

C974005
Check that Tasking_Error is raised at the point of an entry call which is the triggering statement of an asynchronous select, if the entry call is queued, but the task containing the entry completes before it can be accepted or canceled. Check that the abortable part is aborted if it does not complete before the triggering statement completes. Check that the sequence of statements of the triggering alternative is not executed.

C974006
Check that the sequence of statements of the triggering alternative of an asynchronous select

statement is executed if the triggering statement is a protected entry call, and the entry is accepted immediately. Check that the corresponding entry body is executed before the sequence of statements of the triggering alternative. Check that the abortable part is not executed.

C974007
Check that the sequence of statements of the triggering alternative of an asynchronous select statement is not executed if the triggering statement is a protected entry call, and the entry is not accepted before the abortable part completes. Check that execution continues immediately following the asynchronous select.

C974008
Check that the abortable part of an asynchronous select statement is not started if the triggering statement is a task entry call, and the entry call is not queued. Check that the sequence of statements of the triggering alternative is executed after the abortable part is left.

C974009
Check that the abortable part of an asynchronous select statement is not started if the triggering statement is a task entry call, the entry call is not queued and the entry call completes by propagating an exception. Check that the exception is properly propagated to the asynchronous select statement and thus the sequence of statements of the triggering alternative is not executed after the abortable part is left. Check that the exception propagated by the entry call is re-raised immediately following the asynchronous select.

C974010
Check that the abortable part of an asynchronous select statement is not started if the triggering statement is a task entry call to a task that has already terminated. Check that Tasking_Error is properly propagated to the asynchronous select statement and thus the sequence of statements of the triggering alternative is not executed after the abortable part is left. Check that Tasking_Error is re-raised immediately following the asynchronous select.

C974011
Check that the sequence of statements of the triggering alternative of an asynchronous select statement is not executed if the triggering statement is a task entry call and the entry is not accepted before the abortable part completes. Check that the call queued on the entry is cancelled

C974012
Check that the abortable part of an asynchronous select statement is aborted if it does not complete before the triggering statement completes, where the triggering statement is a call on a protected entry which is queued.

C974013
Check that the abortable part of an asynchronous select statement is aborted if it does not complete before the triggering statement completes, where the triggering statement is a delay_until statement. Check that the sequence of statements of the triggering alternative is executed after the abortable part is left.

C974014
Check that if the triggering alternative of an asynchronous select statement is a delay and the abortable part completes before the delay expires then the delay is cancelled and the optional statements in the triggering part are not performed. In particular, check the case of the ATC in non-tasking code.

CA11001
Check that a child unit can be used to provide an alternate view and operations on a private type in its parent package. Check that a child unit can be a package. Check that a WITH of a child unit includes an implicit WITH of its ancestor unit.

CA11002
Check that a public child can utilize its parent unit's visible definitions.

CA11003
Check that a public grandchild can utilize its ancestor unit's visible definitions.

CA110040
Check that the private part of a child library unit package can utilize its parent unit's visible definitions.

CA110041
Check that the private part of a child library unit package can utilize its parent unit's visible definitions.

CA110042
Check that the private part of a child library unit package can utilize its parent unit's visible definitions.

CA110050
Check that entities and operations declared in a package can be used in the private part of a child of a child of the package.

CA110051
Check that entities and operations declared in a package can be used in the private part of a child of a child of the package.

CA11006
Check that the private part of a child library unit can utilize its parent unit's private definition.

CA11007
Check that the private part of a grandchild library unit can utilize its grandparent unit's private definition.

CA11008
Check that a private child package can use entities declared in the visible part of its parent unit.

CA11009
Check that a private child package can use entities declared in the visible part of the parent unit of its parent unit.

CA11010
Check that a private child package can use entities declared in the private part of its parent unit.

CA11011
Check that a private child package can use entities declared in the private part of the parent unit of its parent unit.

CA11012
Check that a child package of a library level instantiation of a generic can be the instantiation of a child package of the generic. Check that the child instance can use its parent's declarations and operations, including a formal type of the parent.

CA11013
Check that a child function of a library level instantiation of a generic can be the instantiation of a child function of the generic. Check that the child instance can use its parent's declarations and operations, including a formal subprogram of the parent.

CA11014
Check that an instantiation of a child package of a generic package can use its parent's declarations and operations, including a formal package of the parent.

CA11015
Check that a generic child of a non-generic package can use its parent's declarations and operations. Check that the instantiation of the generic child can correctly use the operations.

CA11016
Check that a child of a non-generic package can be a private generic package. Check that the private child instance can use its parent's declarations and operations. Check that the body of a public child package can instantiate its sibling private generic package.

CA11017
Check that body of the parent package may depend on one of its own public children.

CA11018
Check that body of the parent package may depend on one of its own public generic children.

CA11019
Check that body of the parent package may depend on one of its own private generic children.

CA11020
Check that body of the generic parent package can depend on one of its own public generic children.

CA11021
Check that body of the generic parent package can depend on one of its own private generic children.

CA11022
Check that body of a child unit can instantiate its generic sibling.

CA11A01
Check that type extended in a public child inherits primitive operations from its ancestor.

CA11A02
Check that a type extended in a client of a public child inherits primitive operations from parent.

CA11B01
Check that a type derived in a public child inherits primitive operations from parent.

CA11B02
Check that a type derived in a client of a public child inherits primitive operations from parent.

CA11C01
Check that when primitive operations declared in a child package override operations declared in ancestor packages, a client of the child package inherits the operations correctly.

CA11C02
Check that primitive operations declared in a child package override operations declared in ancestor packages, and that operations on class-wide types defined in the ancestor packages dispatch as appropriate to these overriding implementations.

CA11C03
Check that when a child unit is "withed", visibility is obtained to all ancestor units named in the expanded name of the "withed" child unit. Check that when the parent unit is "used", the simple name of a "withed" child unit is made directly visible.

CA11D010
Check that a child unit can raise an exception that is declared in parent.

CA11D011
Check that a child unit can raise an exception that is declared in parent.

CA11D012
Check that a child unit can raise an exception that is declared in parent.

CA11D013
Check that a child unit can raise an exception that is declared in parent.

CA11D02
Check that an exception declared in a package can be raised by a child of a child package. Check that it can be renamed in the child of the child package and raised with the correct effect.

CA11D03
Check that an exception declared in a package can be raised by a client of a child of the package. Check that it can be renamed in the client of the child of the package and raised with the correct effect.

CA13001
Check that a separate protected unit declared in a non-generic child unit of a private parent have the same visibility into its parent, its siblings, and packages on which its parent depends as is available at the point of their declaration.

CA13002
Check that two library child units and/or subunits may have the same simple names if they have distinct expanded names.

CA13003
Check that separate subunits which share an ancestor may have the same name if they have different fully qualified names. Check the case of separate subunits of separate subunits. This test is a change in semantics from Ada 83 to Ada 9X.

CA13A01
Check that subunits declared in non-generic child units of a public parent have the same visibility into its parent, its siblings (public and private), and packages on which its parent depends as is available at the point of their declaration.

CA13A02
Check that subunits declared in generic child units of a public parent have the same visibility into its parent, its siblings (public and private), and packages on which its parent depends as is available at the point of their declaration.

CB20001
Check that exceptions can be handled in accept bodies, and that a task object that has an exception handled in an accept body is still viable for future use.

CB20003
Check that exceptions can be raised, reraised, and handled in an accessed subprogram.

CB20004
Check that exceptions propagate correctly from objects of protected types. Check propagation from protected entry bodies.

CB20005
Check that exceptions are raised and properly handled locally in protected operations.

CB20006
Check that exceptions are raised and properly handled (including propagation by reraise) in protected operations.

CB20007
Check that exceptions are raised and can be directly propagated to the calling unit by protected operations.

CB20A02
Check that the name and pertinent information about a user defined exception are available to an enclosing program unit even when the enclosing unit has no visibility into the scope where the exception is declared and raised.

CB40A01
Check that a user defined exception is correctly propagated out of a public child package.

CB40A020
Check that a user defined exception is correctly propagated from a private child subprogram to its parent and then to a client of the parent.

CB40A021
Check that a user defined exception is correctly propagated from a private child subprogram to its parent and then to a client of the parent.

CB40A030
Check that a predefined exception is correctly propagated from a private child package through a visible child package to a client.

CB40A031
Check that a predefined exception is correctly propagated from a private child package through a visible child package to a client.

CB40A04
Check that a predefined exception is correctly propagated out of a public child function to a client.

CB41001
Check that the 'Identity attribute returns the unique identity of an exception. Check that the Raise_Exception procedure can raise an exception that is specified through the use of the 'Identity attribute, and that Reraise_Occurrence can re-raise an exception occurrence using an exception choice parameter.

CB41002
Check that the message string input parameter in a call to the Raise_Exception procedure is associated with the raised exception occurrence, and that the message string can be obtained using the Exception_Message function with the associated Exception_Occurrence object. Check that Function Exception_Information is available to provide implementation-defined information about the exception occurrence.

CB41003
Check that an exception occurrence can be saved into an object of type Exception_Occurrence using the procedure Save_Occurrence. Check that a saved exception occurrence can be used to reraise another occurrence of the same exception using the procedure Reraise_Occurrence. Check that the function Save_Occurrence will allocate a new object of type Exception_Occurrence_Access, and saves the source exception to the new object which is returned as the function result.

CB41004
Check that Raise_Exception and Reraise_Occurrence have no effect in the case of Null_Id or Null_Occurrence. Check that Exception_Message, Exception_Identity, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence input parameter. Check that calling the Save_Occurrence subprograms with the Null_Occurrence input parameter saves the Null_Occurrence to the appropriate target object, and does not raise Constraint_Error. Check that Null_Id is the default initial value of type Exception_Id.

CC30001
Check that if a non-overriding primitive subprogram is declared for a type derived from a formal derived tagged type, the copy of that subprogram in an instance can override a subprogram inherited from the actual type.

CC50001
Check that, in an instance, each implicit declaration of a predefined operator of a formal tagged private type declares a view of the corresponding predefined operator of the actual type (even if the operator has been overridden for the actual type). Check that the body executed is determined by the type and tag of the operands.

CC50A01
Check that a formal parameter of a library-level generic unit may be a formal tagged private type. Check that a nonlimited tagged type may be passed as an actual. Check that if the formal type is indefinite, both indefinite and definite types may be passed as actuals.

CC50A02
Check that a nonlimited tagged type may be passed as an actual to a formal (non-tagged) private type. Check that if the formal type has an unknown discriminant part, a class-wide type may also be passed as an actual.

CC51001
Check that a formal parameter of a generic package may be a formal derived type. Check that the formal derived type may have an unknown discriminant part. Check that the ancestor type in a formal derived type definition may be a tagged type, and that the actual parameter may be a descendant of the ancestor type. Check that the formal derived type belongs to the derivation class rooted at the ancestor type; specifically, that components of the ancestor type may be referenced within the generic. Check that if a formal derived subtype is indefinite then the actual may be either definite or indefinite.

CC51002
Check that, for formal derived tagged types, the formal parameter names and default expressions for a primitive subprogram in an instance are determined by the primitive subprogram of the ancestor type, but that the primitive subprogram body executed is that of the actual type.

CC51003
Check that if the ancestor type of a formal derived type is a composite type that is not an array type, the formal type inherits components, including discriminants, from the ancestor type. Check for the case where the ancestor type is a record type, and the formal derived type is declared in a generic subprogram.

CC51004
Check that if the ancestor type of a formal derived type is a composite type that is not an array type, the formal type inherits components, including discriminants, from the ancestor type. Check for the case where the ancestor type is a tagged type, and the formal derived type is declared in a generic subprogram.

CC51006
Check that, in an instance, each implicit declaration of a primitive subprogram of a formal (nontagged) derived type declares a view of the corresponding primitive subprogram of the ancestor type, even if the subprogram has been overridden for the actual type. Check that for a formal derived type with no discriminant part, if the ancestor subtype is an unconstrained scalar subtype then the actual may be either constrained or unconstrained.

CC51007
Check that a generic formal derived tagged type is a private extension. Specifically, check that, for a generic formal derived type whose ancestor type has abstract primitive subprograms, neither the formal derived type nor its descendants need be abstract. Check that objects and components of the formal derived type and its nonabstract descendants may be declared and allocated, as may nonabstract functions returning these types, and that aggregates of nonabstract descendants of the formal derived type are legal. Check that calls to the abstract primitive subprograms of the ancestor dispatch to the bodies corresponding to the tag of the actual parameters.

CC51A01
Check that, in an instance, each implicit declaration of a user-defined subprogram of a formal derived record type declares a view of the corresponding primitive subprogram of the ancestor, even if the primitive subprogram has been overridden for the actual type.

CC51B03
Check that the attribute S'Definite, where S is an indefinite formal private or derived type, returns true if the actual corresponding to S is definite, and returns false otherwise.

CC51D01
Check that, in an instance, each implicit declaration of a user-defined subprogram of a formal private extension declares a view of the corresponding primitive subprogram of the ancestor, and that if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type. Check subprograms declared within a generic formal package. Check for the case where the actual type passed to the formal private extension is a specific tagged type. Check for several types in the same class.

CC51D02
Check that, in an instance, each implicit declaration of a user-defined subprogram of a formal private extension declares a view of the corresponding primitive subprogram of the ancestor, and

that if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type. Check subprograms declared within a generic formal package. Check for the case where the actual type passed to the formal private extension is a class-wide type. Check for several types in the same class.

CC54001
Check that a general access-to-constant type may be passed as an actual to a generic formal access-to-constant type.

CC54002
Check that a general access-to-variable type may be passed as an actual to a generic formal general access-to-variable type. Check that designated objects may be read and updated through the access value.

CC54003
Check that a general access-to-subprogram type may be passed as an actual to a generic formal access-to-subprogram type. Check that designated subprograms may be called by dereferencing the access values.

CC54004
Check that the designated type of a generic formal pool-specific access type may be class-wide. Check that calls to primitive subprograms in the instance dispatch to the appropriate bodies when the controlling operand is a dereference of an object of the access- to-class-wide type.

CC70001
Check that the template for a generic formal package may be a child package, and that a child instance which is an instance of the template may be passed as an actual to the formal package. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification.

CC70002
Check that a formal package actual part may specify actual parameters for a generic formal package. Check that these actual parameters may be formal types, formal objects, and formal subprograms. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification, and that if the formal package actual part is (<>), it also includes the generic formal part of the template for the formal package.

CC70003
Check that the actual passed to a formal package may be a formal access-to-subprogram type. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification.

CC70A01
Check that the visible part of a generic formal package includes the first list of basic declarative items of the package specification. Check for a generic package which declares a formal package with (<>) as its actual part.

CC70A02
Check that the visible part of a generic formal package includes the first list of basic declarative items of the package specification. Check for a generic subprogram which declares a formal package with (<>) as its actual part.

CC70B01
Check that a formal package actual part may specify actual parameters for a generic formal package. Check that a use clause in the generic formal part provides direct visibility of declarations within the generic formal package. Check that the scope of such a use clause extends to the generic subprogram body. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification. Check the case where the formal package is declared in a generic subprogram.

CC70B02
Check that a formal package actual part may specify actual parameters for a generic formal package. Check that such an actual parameter may be a formal parameter of a previously declared formal

package (with a (<>) actual part). Check that a use clause in the generic formal part provides direct visibility of declarations within the generic formal package, including formal parameters (if the formal package has a (<>) actual part). Check that the scope of such a use clause extends to the generic subprogram body. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification. Check the case where the formal package is declared in a generic package.

CC70C01
Check that a generic formal package is an instance. Specifically, check that a generic formal package may be passed as an actual parameter in an instantiation of a generic package. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification.

CC70C02
Check that a generic formal package is an instance. Specifically, check that a generic formal package may be passed as an actual parameter to another generic formal package. Check that the visible part of the generic formal package includes the first list of basic declarative items of the package specification.

CXA3001
Check that the character classification functions defined in package Ada.Characters.Handling produce correct results when provided constant arguments from package Ada.Characters.Latin_1.

CXA3002
Check that the conversion functions for Characters and Strings defined in package Ada.Characters.Handling provide correct results when given character/string input parameters.

CXA3003
Check that the functions defined in package Ada.Characters.Handling for use in classifying and converting characters between the ISO 646 and type Character sets produce the correct results with both Character and String input values.

CXA3004
Check that the functions defined in package Ada.Characters.Handling for classification of and conversion between Wide_Character and Character values produce correct results when given the appropriate Character and String inputs.

CXA4001
Check that the types, operations, and other entities defined within the package Ada.Strings.Maps are available and/or produce correct results.

CXA4002
Check that the subprograms defined in package Ada.Strings.Fixed are available, and that they produce correct results. Specifically, check the subprograms Index, "*" (string constructor function), Count, Trim, and Replace_Slice.

CXA4003
Check that the subprograms defined in package Ada.Strings.Fixed are available, and that they produce correct results. Specifically, check the subprograms Index, Index_Non_Blank, Head, Tail, Translate, Find_Token, Move, Overwrite, and Replace_Slice.

CXA4004
Check that the subprograms defined in package Ada.Strings.Fixed are available, and that they produce correct results. Specifically, check the subprograms Count, Find_Token, Index, Index_Non_Blank, and Move.

CXA4005
Check that the subprograms defined in package Ada.Strings.Fixed are available, and that they produce correct results. Specifically, check the subprograms Delete, Head, Insert, Overwrite, Replace_Slice, Tail, Trim, and "*".

CXA4006
Check that the subprograms defined in package Ada.Strings.Bounded are available, and that they

produce correct results. Specifically, check the subprograms Length, Slice, "&", To_Bounded_String, Append, Index, To_String, Replace_Slice, Trim, Overwrite, Delete, Insert, and Translate.

CXA4007
Check that the subprograms defined in package Ada.Strings.Bounded are available, and that they produce correct results. Specifically, check the subprograms Append, Count, Element, Find_Token, Head, Index_Non_Blank, Replace_Element, Replicate, Tail, To_Bounded_String, "&", ">", "<", ">=", "<=", and "*".

CXA4008
Check that the subprograms defined in package Ada.Strings.Bounded are available, and that they produce correct results, especially under conditions where truncation of the result is required. Specifically, check the subprograms Append, Count with non-Identity maps, Index with non-Identity maps, Index with Set parameters, Insert (function and procedure), Replace_Slice (function and procedure), To_Bounded_String, and Translate.

CXA4009
Check that the subprograms defined in package Ada.Strings.Bounded are available, and that they produce correct results, especially under conditions where truncation of the result is required. Specifically, check the subprograms Overwrite (function and procedure), Delete, Function Trim (blanks), Trim (Set characters, function and procedure), Head, Tail, and Replicate (characters and strings).

CXA4010
Check that the subprograms defined in package Ada.Strings.Unbounded are available, and that they produce correct results. Specifically, check the subprograms To_String, To_Unbounded_String, Insert, "&", "*", Length, Slice, Replace_Slice, Overwrite, Index, Index_Non_Blank, Head, Tail, and "=", "<=", ">=".

CXA4011
Check that the subprograms defined in package Ada.Strings.Unbounded are available, and that they produce correct results. Specifically, check the subprograms To_Unbounded_String, "&", ">", "<", Element, Replace_Element, Count, Find_Token, Translate, Trim, Delete, and "*".

CXA4012
Check that the types, operations, and other entities defined within the package Ada.Strings.Wide_Maps are available and produce correct results.

CXA4013
Check that the subprograms defined in package Ada.Strings.Wide_Fixed are available, and that they produce correct results. Specifically, check the subprograms Index, "*" (Wide_String constructor function), Count, Trim, and Replace_Slice.

CXA4014
Check that the subprograms defined in package Ada.Strings.Wide_Fixed are available, and that they produce correct results. Specifically, check the subprograms Find_Token, Head, Index, Index_Non_Blank, Move, Overwrite, and Replace_Slice, Tail, and Translate. Use the access-to-subprogram mapping version of Translate (function and procedure).

CXA4015
Check that the subprograms defined in package Ada.Strings.Wide_Fixed are available, and that they produce correct results. Specifically, check the subprograms Count, Find_Token, Index, Index_Non_Blank, and Move.

CXA4016
Check that the subprograms defined in package Ada.Strings.Wide_Fixed are available, and that they produce correct results. Specifically, check the subprograms Delete, Head, Insert, Overwrite, Replace_Slice, Tail, Trim, and "*".

CXA4017
Check that the subprograms defined in package Ada.Strings.Wide_Bounded are available, and that they produce correct results. Specifically, check the subprograms Append, Delete, Index, Insert , Length, Overwrite, Replace_Slice, Slice, "&", To_Bounded_Wide_String, To_Wide_String,

Translate, and Trim.

CXA4018
Check that the subprograms defined in package Ada.Strings.Wide_Bounded are available, and that they produce correct results. Specifically, check the subprograms Append, Count, Element, Find_Token, Head, Index_Non_Blank, Replace_Element, Replicate, Tail, To_Bounded_Wide_String, "&", ">", "<", ">=", "<=", and "*".

CXA4019
Check that the subprograms defined in package Ada.Strings.Wide_Bounded are available, and that they produce correct results, especially under conditions where truncation of the result is required. Specifically, check the subprograms Append, Count with non-Identity maps, Index with non-Identity maps, Index with Set parameters, Insert (function and procedure), Replace_Slice (function and procedure), To_Bounded_Wide_String, and Translate (function and procedure).

CXA4020
Check that the subprograms defined in package Ada.Strings.Wide_Bounded are available, and that they produce correct results, especially under conditions where truncation of the result is required. Specifically, check the subprograms Overwrite (function and procedure), Delete, Function Trim (blanks), Trim (Set wide characters, function and procedure), Head, Tail, and Replicate (wide characters and wide strings).

CXA4021
Check that the subprograms defined in package Ada.Strings.Wide_Unbounded are available, and that they produce correct results. Specifically, check the subprograms Head, Index, Index_Non_Blank, Insert, Length, Overwrite, Replace_Slice, Slice, Tail, To_Wide_String, To_Unbounded_Wide_String, "*", "&", and "=", "<=", ">=".

CXA4022
Check that the subprograms defined in package Ada.Strings.Wide_Unbounded are available, and that they produce correct results. Specifically, check the subprograms Count, Element, Index, Replace_Element, To_Unbounded_Wide_String, and "&", ">", "<".

CXA4023
Check that the subprograms defined in package Ada.Strings.Wide_Unbounded are available, and that they produce correct results. Specifically, check the subprograms Delete, Find_Token, Translate, Trim, and "*".

CXA8001
Check that all elements to be transferred to a sequential file of mode Append_File will be placed following the last element currently in the file. Check that it is possible to append data to a file that has been previously appended to. Check that the predefined procedure Write will place an element after the last element in the file in mode Append_File.

CXA8002
Check that resetting a file using mode Append_File allows for the writing of elements to the file starting after the last element in the file. Check that the result of function Name can be used on a subsequent reopen of the file. Check that a mode change occurs on reset of a file to/from mode Append_File.

CXA8003
Check that Append_File mode has not been added to package Direct_IO.

CXA9001
Check that the operations defined in the generic package Ada.Storage_IO provide the ability to store and retrieve objects which may include implicit levels of indirection in their implementation, from an in-memory buffer.

CXA9002
Check that the operations defined in the generic package Ada.Storage_IO provide the ability to store and retrieve objects of tagged types from in-memory buffers.

CXAA001

Check that the Line_Length and Page_Length maximums for a Text_IO file of mode Append_File are initially zero (unbounded) after a Create, Open, or Reset, and that these values can be modified using the procedures Set_Line_Length and Set_Page_Length. Check that setting the Line_Length and Page_Length attributes to zero results in an unbounded Text_IO file. Check that setting the line length when in Append_Mode doesn't change the length of lines previously written to the Text_IO file.

CXAA002
Check that the procedures New_Page, Set_Line, Set_Col, and New_Line subprograms perform properly on a text file created with mode Append_File. Check that the attributes Page, Line, and Column are all set to 1 following the creation of a text file with mode Append_File. Check that the functions Page, Line, and Col perform properly on a text file created with mode Append_File. Check that the procedures Put and Put_Line perform properly on text files created with mode Append_File. Check that the procedure Set_Line sets the current line number to the value specified by the parameter "To" for text files created with mode Append_File. Check that the procedure Set_Col sets the current column number to the value specified by the parameter "To" for text files created with mode Append_File.

CXAA003
Check that the procedures New_Page, Set_Line, Set_Col, and New_Line subprograms perform properly on a text file reset (from Out_File) with mode Append_File. Check that the attributes Page, Line, and Column are all set to 1 following the reset of a text file with mode Append_File. Check that the functions Page, Line, and Col perform properly on a text file reset with mode Append_File. Check that the procedures Put and Put_Line perform properly on text files reset with mode Append_File. Check that the procedure Set_Line sets the current line number to the value specified by the parameter "To" for text files reset with mode Append_File. Check that Set_Line has no effect if the specified line equals the current line. Check that the procedure Set_Col sets the current column number to the value specified by the parameter "To" for text files reset with mode Append_File.

CXAA004
Check that the procedures New_Page, Set_Line, Set_Col, and New_Line perform properly on a text file opened with mode Append_File. Check that the attributes Page, Line, and Column are all set to 1 following the opening of a text file with mode Append_File. Check that the functions Page, Line, and Col perform properly on a text file opened with mode Append_File. Check that the procedures Put and Put_Line perform properly on text files opened with mode Append_File. Check that the procedure Set_Line sets the current line number to the value specified by the parameter "To" for text files opened with mode Append_File. Check that the procedure Set_Col sets the current column number to the value specified by the parameter "To" for text files reset with mode Append_File.

CXAA005
Check that the procedure Put, when called with string parameters, does not update the line number of a text file of mode Append_File, when the line length is unbounded (i.e., only the column number is updated). Check that a call to the procedure Put with a null string argument has no measurable effect on a text file of mode Append_File.

CXAA006
Check that for a bounded line length text file of mode Append_File, when the number of characters to be output exceeds the number of columns remaining on the current line, a call to Put will output characters of the string sufficient to fill the remaining columns of the line (up to line length), then output a line terminator, reset the column number, increment the line number, then output the balance of the item. Check that the procedure Put does not raise Layout_Error when the number of characters to be output exceeds the line length of a bounded text file of mode Append_File.

CXAA007
Check that the capabilities of Text_IO.Integer_IO perform correctly on files of Append_File mode, for instantiations with integer and user-defined subtypes. Check that the formatting parameters available in the package can be used and modified successfully in the storage and retrieval of data.

CXAA008
Check that the capabilities provided in instantiations of the Ada.Text_IO.Fixed_IO package operate correctly when the mode of the file is Append_File. Check that Fixed_IO procedures Put and Get properly transfer fixed point data to/from data files that are in Append_File mode. Check that the formatting parameters available in the package can be used and modified successfully in the

appending and retrieval of data.

CXAA009
Check that the capabilities provided in instantiations of the Ada.Text_IO.Float_IO package operate correctly when the mode of the file is Append_File. Check that Float_IO procedures Put and Get properly transfer floating point data to/from data files that are in Append_File mode. Check that the formatting parameters available in the package can be used and modified successfully in the appending and retrieval of data.

CXAA010
Check that the operations defined in package Ada.Text_IO.Decimal_IO are available, and that they function correctly when used for the input/output of Decimal types.

CXAA011
Check that the operations of Text_IO.Enumeration_IO perform correctly on files of Append_File mode, for instantiations using enumeration types. Check that Enumeration_IO procedures Put and Get properly transfer enumeration data to/from data files. Check that the formatting parameters available in the package can be used and modified successfully in the storage and retrieval of data.

CXAA012
Check that the exception Mode_Error is raised when an attempt is made to read from (perform a Get_Line) or use the predefined End_Of_File function on a text file with mode Append_File.

CXAA013
Check that the exception Mode_Error is raised when an attempt is made to skip a line or page using the predefined Skip_Line and Skip_Page procedures on a text file with mode Append_File.

CXAA014
Check that the exception Mode_Error is raised when an attempt is made to check for the end of a line or page using the predefined functions End_Of_Line or End_Of_Page on a text file with mode Append_File.

CXAA015
Check that the exception Status_Error is raised when an attempt is made to create or open a file in Append_File mode when the file is already open. Check that the exception Name_Error is raised by procedure Open when attempting to open a file in Append_File mode when the name supplied as the filename does not correspond to an existing external file.

CXAB001
Check that the operations defined in package Wide_Text_IO allow for the input/output of Wide_Character and Wide_String data.

CXAC001
Check that the attribute T'Write will, for any specific non-limited type T, write an item of the subtype to the stream. Check that the attribute T'Read will, for a specific non-limited type T, read a value of the subtype from the stream.

CXAC002
Check that the subprograms defined in package Ada.Streams.Stream_IO are accessible, and that they provide the appropriate functionality.

CXAC003
Check that the correct exceptions are raised when improperly manipulating stream file objects.

CXACA01
Check that the default attributes 'Write and 'Read work properly when used with objects of a variety of types, including records with default discriminants, records without default discriminants, but which have the discriminant described in a representation clause for the type, and arrays.

CXACA02
Check that user defined subprograms can override the default attributes 'Read and 'Write using attribute definition clauses. Use objects of record types.

CXACB01
Check that the default attributes 'Input and 'Output work properly when used with objects of a variety of types, including two-dimensional arrays and records without default discriminants.

CXACB02
Check that user defined subprograms can override the default attributes 'Input and 'Output using attribute definition clauses, when used with objects of discriminated record and multi-dimensional array types.

CXACC01
Check that the use of 'Class'Output and 'Class'Input allow stream manipulation of objects of non-limited class-wide types.

CXB3001
Check that the specifications of the package Interfaces.C are available for use.

CXB3002
Check that the specifications of the package Interfaces.C.Strings are available for use.

CXB3003
Check that the specifications of the package Interfaces.C.Pointers are available for use.

CXB4001
Check that the specifications of the package Interfaces.COBOL are available for use

CXB5001
Check that the specification of the package Interfaces.FORTRAN are available for use.

CXC3001
Check that an interrupt handler can be attached to an interrupt using Attach_Handler and Exchange_Handler. Check that Is_Reserved and Is_Attached return the correct status of an interrupt. Check that Current_Handler returns an access value to the handler currently attached to an interrupt.

CXC6001
Check that atomic and volatile elementary types that are not by-copy types, as well as types with subcomponents that are atomic or volatile are by-reference types.

CXC6002
Check that atomic and volatile composite types that are not by-copy types, as well as types with subcomponents that are atomic or volatile are by-reference types. Check that parameters are passed by copy when an actual parameter is defined as atomic or volatile, and the formal parameter is not.

CXC7001
In the package Ada.Task_Identification, check that Current_Task returns the Task_ID of the calling task; Abort_Task aborts the task corresponding to the Task_ID parameter; Is_Terminated and Is_Callable return the corresponding attribute values for the task corresponding to the Task_ID parameter. Check that an object of type Task_ID is default initialized to Null_Task_ID. Check that the attribute T'Identity returns a Task_ID that identifies task T and the C'Caller returns a Task_ID that identifies the caller of entry E.

CXD1001
Check that the range of System.Priority is at least 30 values; that System.Interrupt_Priority has at least one value and is higher than System.Priority and the System.Default_Priority is at the center of the range of System.Priority. Check the behavior of Ada.Dynamic_Priorities.Set_Priority and Get_Priority; specifically that Set_Priority will set a value that can later be confirmed with Get_Priority. Check that, in the absence of Pragma Priority, the main subprogam has a base priority of Default_Priority.

CXD1002
Check that the base priority of the main subprogram can be set by means of pragma priority. Check that a task's base priority is the priority of the parent at the time the task is created when

the priority of the parent has been set by means of pragma priority Check that a task's base priority is the priority of the parent at the time the task is created when the priority of the grandfather has been set by means of pragma priority

CXD1003
Check that during rendezvous, the task accepting the entry call inherits the active priority of the caller. Specifically, check when the caller has a higher priority than the receiver.

CXD1004
Check that during activation, a task being activated inherits the active priority of its activator (in this case the activator's base priority). Check that, if this priority is higher than the base priority of the activated task, this base priority remains unchanged.

CXD1005
Check that, during activation, a task being activated inherits the active priority of its activator. Specifically, check when the active priority of the activator is higher than the activator's Base Priority. Check that if the priority of the activated task is higher than its base priority, the base priority remains unchanged.

CXD2001
Check that when Task_Dispatching_Policy is FIFO_Within_Priorities and the setting of the base priority of a task takes effect, the task is added to the tail of the ready queue of its active priority.

CXD2003
Check that when Task_Dispatching_Policy is FIFO_Within_Priorities and and a task's priority is lowered due to the loss of inherited priority it is added to the head of the ready queue for its priority

CXD2004
Check that when Task_Dispatching_Policy is FIFO_Within_Priorities and the active priority of a running task is lowered due to loss of its inherited priority and there is a ready task of the same priority that is not running, the running task continues to run.

CXD3001
Check that Program_Error is raised if a task calls a protected operation who's ceiling is lower than the task's active priority. Check this for Function, Procedure and Entry. Check that the exception is not raised if the ceiling is equal to or higher than the priority of the calling task.

CXD3002
Check that when Locking_Policy is Ceiling_Locking and no pragma Priority, Interrupt_Priority, Interrupt_Handler or Attach_Handler is specified in a protected definition the Ceiling Priority of the protected object is System.Priority'Last

CXD4001
Check that when Priority Queuing is in effect and the base priority of a task is set (changed), the priorities of any queued calls from that task are updated and that the ordering is modified accordingly.

CXD4002
Check that if no Queuing_Policy is specified, the policy for the partition is FIFO_Queuing and that the priorities of the calling tasks have no effect.

CXD4003
Check that if Queuing_Policy FIFO_Queuing is specified for a partition the task entry queues are handled in FIFO order and that the priorities of the calling tasks have no effect.

CXD4004
Check that changes to the active priority of the caller do not affect the priority of a call after it is first queued when the queuing policy is priority queuing.

CXD4005
Check that when Priority Queuing is in effect and the base priority of a task is set (changed), the priorities of any queued calls from that task to entries in a Protected Object are updated and that the

ordering is modified accordingly.

CXD4006
Check that if Queuing_Policy is Priority_Queuing, the calls to an entry are queued in an order consistent with the priority of the calls and that if an entry is removed and then reinserted it is added behind any other calls with equal priority in that queue.

CXD5001
Check that for Get_Priority, Tasking_Error is raised if the specified task has terminated. Check that for Get & Set Priority, Program_Error is raised if the task has a null Task_Identification.

CXD8001
Check the basic functions in the Package Ada.Real_Time.

CXDA001
Check that, in Ada.Synchronous_Task_Control, Set_True and Set_False alter the state of a Suspension_Object appropriately. Check that Current_State returns the expected state. Check that the initial value of a Suspension_Object is set to false.

CXDA002
Check that, in Ada.Synchronous_Task_Control, Suspend_Until_True does suspend the task until the Suspension_Object is Set_True. Check that a call on Suspend_Until_True will raise Program_Error if another task is waiting on the same Suspension_Object.

CXDB001
Check that, in Ada.Asynchronous_Task_Control, the Hold operation reduces the priority of the target task to such a state that it does not run and that Continue raises it such that it will run again. Check that Is_Held returns true if, and only if, the target task is in the Held state. Check that Tasking_Error is raised if any of these operations is applied to a task that is terminated.

CXDB002
Check that the effect of calling Get_Priority and Set_Priority on a Held task is the same as on any other task.

CXDB003
Check that if a task becomes Held while waiting in a selective accept and an entry call is issued to one of the open entries, the corresponding accept body executes. Check that once the rendezvous completes the task does not execute until another Continue.

CXDB004
Check that if a calling task is Held while waiting for a rendezvous to complete the active priority of the receiver is unaffected.

CXE1001
Check that the attribute D'Partition_ID is available where D denotes a library level declaration. Check that this attribute identifies the partition in which D is elaborated.

CXE5001
Check that the specifications of the package System.RPC are available for use.

CXF1001
Check that values of 2 and 10 are allowable values for Machine_Radix of a decimal first subtype. Check that the value of Decimal.Max_Decimal_Digits is at least 18; the value of Decimal.Max_Scale is at least 18; the value of Decimal.Min_Scale is at most 0.

CXF2001
Check that the Divide procedure provides the following results: Quotient = Dividend divided by Divisor and Remainder = Dividend - (Divisor * Quotient) Check that the Remainder is calculated exactly.

CXF3A01
Check that the function Ada.Text_IO.Editing.Valid returns False if a) Pic_String is not a well-formed Picture string, or b) the length of Pic_String exceeds Max_Picture_Length, or c)

Blank_When_Zero is True and Pic_String contains '*'; Check that Valid otherwise returns True.

CXF3A02
Check that the function Ada.Text_IO.Editing.To_Picture raises Picture_Error if the picture string provided as input parameter does not conform to the composition constraints defined for picture strings. Check that when Pic_String is applied to To_Picture, the result is equivalent to the actual string parameter of To_Picture; Check that when Blank_When_Zero is applied to To_Picture, the result is the same value as the Blank_When_Zero parameter of To_Picture.

CXF3A03
Check that function Length in the generic package Decimal_Output returns the number of characters in the edited output string produced by function Image, for a particular decimal type, currency string, and radix mark. Check that function Valid in the generic package Decimal_Output returns correct results based on the particular decimal value, and the Picture and Currency string parameters.

CXG1001
Check that the subprograms defined in the package Ada.Numerics.Generic_Complex_Types provide correct results. Specifically, check the functions Re, Im (both versions), procedures Set_Re, Set_Im (both versions), functions Compose_From_Cartesian (all versions), Compose_From_Polar, Modulus, Argument, and "abs".

CXG1002
Check that the subprograms defined in the package Ada.Numerics.Generic_Complex_Types provide the prescribed results. Specifically, check the various versions of functions "+" and "-".

CXG1003
Check that the subprograms defined in the package Text_IO.Complex_IO provide correct results.

CXG1004
Check that the specified exceptions are raised by the subprograms defined in package Ada.Numerics.Generic_Complex_Elementary_Functions given the prescribed input parameter values.

CXG1005
Check that the subprograms defined in the package Ada.Numerics.Generic_Complex_Elementary_Functions provide correct results.

LXD7001
Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Task_Hierarchy) is included.

LXD7003
Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Abort_Statements) is included. Specifically a task with an abort_statement is not allowed.

LXD7004
Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Terminate_Alternatives) is included.

LXD7005
Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Task_Allocators) is included.

LXD7006
Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Task_Allocators) is included. Specifically that there are no allocators for types containing task subcomponents

LXD7007
Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Dynamic_Priorities) is included. Specifically when there is a semantic dependency on Ada.Dynamic_Priorities in a package making up the partition