Note that this document contains the OBJECTIVE section of each Ada 95 test. None
of the legacy tests are included.

B330001

Check that if a subtype indication of a variable object defines an indefinite subtype, then there is an initialization. Check that if the array type definition of a variable object defines an unconstrained array subtype, then there is an initialization. Check that indefinite subtypes may not be used as the subtype indication of a component definition (in either an array or a record definition).

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.

B370001

Check that a discriminant specification for an access discriminant may not appear in the declaration of a type (that is not a task or protected type) if the word limited does not appear in the definition of the type or in that of one of its ancestors. Check for basic cases, including a type that is limited only due to the presence of a limited component. Check for the generic case, where the type is derived from a nonlimited tagged formal private type, or a formal private extension. Check for the instance case, where the type is derived from a limited tagged formal private type, and the corresponding actual is not limited. Check in both the visible and private part of an instance, using record and private extensions.

B370002

Check that for derived types with known discriminant parts the parent subtype must be constrained; if the parent type is not tagged, each discriminant of the new type must be used in the constraint defining the parent subtype; and, if a discriminant is used in the constraint defining the parent subtype, the subtype of that discriminant must be statically compatible with the subtype of the corresponding parent discriminant.

B380001

Check that the name of a non-inherited discriminant can be the same as the name of a newly added component. Check that the name of the current instance of a type used to define the constraint of a component may only be used as a direct_name that is the prefix of an attribute_reference whose result is of an access type, and that the attribute_reference must appear alone. Check that the name of a non-inherited discriminant is not allowed within the discriminant part.

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.

B391003

Check that the parent type of a record extension may not be a class-wide type. Check for the basic case. Check for the generic case, where the parent type is the class-wide type of a formal tagged private type or formal private extension. Check for the instance case, where the parent type is a formal tagged private type or formal private extension, and the corresponding actual type is a class-wide type. Check that this rule is enforced in the visible and private part of an instance.

B391004

Check that if a (non-derived) tagged type has any limited components, the reserved word limited must appear in its definition. Check for basic and generic cases. Check that if the parent type of a record extension is nonlimited, each of the components of the record extension part must be nonlimited. Check for generic declarations and instances.

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. Subprogram bodies in a package body that are homographs of inherited primitive abstract operations are illegal.

B393004

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

B393005

Check that an abstract type derived from a tagged parent may override primitive functions with controlling results as abstract. Check that an abstract type derived from a tagged parent may not override primitive functions with controlling results as not abstract. Check that when a non-abstract or untagged type is derived from a tagged parent with a primitive function returning a controlling result, the function with the controlling result must be overridden. Check that an abstract private type may not have a primitive abstract subprogram if the full view of the type is not 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.

B3A0002

Check that subtype conformance is required for actual values of access to subprogram types. Check that the mode, number and subtype of parameters must statically match. Check that the calling convention of the value must not be Intrinsic. Check that corresponding subtypes of the profiles must statically match. Check that a generic formal subprogram may not be the actual value of an access to subprogram type because it cannot subtype-conform with anything.

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.

B3A0004

For an array object X used as the prefix for the attributes X'Access or X'Unchecked_Access, where the expected type for X'Access or X'Unchecked_Access is the general access type A: Check that the nominal subtype of an aliased view of X must statically match A's designated array subtype.

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.

B3A2005

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 the case where A is an anonymous access type, and X'Access is used to initialize an access discriminant of an object created by an allocator.

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.

B3A2008

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 private 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 view defined by a component definition. (c) a dereference of an access-to-object value.

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.

B3A2011

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 private part of an instance and X is passed as an actual during instantiation. 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.

B3A2012

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 passed as an actual during instantiation. Check for cases where X is: (a) a view defined by an object declaration. (b) a view denoted by a component definition. (c) a dereference of an access-to-object value.

B3A2013

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 current instance of a limited type. (b) a current instance of a limited type in a type conversion.

B3A2014

Check that, for X'Access of a general access type A, if the designated type is tagged, the type of the view denoted by X must be covered by the designated type. Check that if the designated type is not tagged, the type of the view must be the same, and either A's designated subtype must statically match the nominal subtype of the view, or the designated subtype must be discriminated and unconstrained.

B3A2015

Check that, for X'Access of a general access type A, the view denoted by X must not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. Check that, for a renaming of an object, the renamed entity must not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. Check that if the view denoted by X is that of a subcomponent of an aliased composite object, the word aliased must appear in the subcomponent's component definition. Check for objects which are declared aliased, objects created by an allocator, and formal parameters of a tagged type.

B430001

Check that an aggregate may not be of a class-wide type. Check that "null record" may appear as a record component association list in record aggregates and extension aggregates. Check that if no components are needed in a record component association list, "null record" must appear, and that if components are needed, "null record" must not appear.

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.

B460002

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 type conversion occurs in the visible or private part of an instance.

B460004

Check that if the target type of a type conversion is tagged, the operand type must be covered by or descended from the target type, or the operand type must be a class-wide type that covers the target type. Check that if the target designated type of a general access type conversion is tagged, the operand designated type must be convertible to the target designated type.

B480001

Check that if the subtype indication of an uninitialized allocator specifies an access type, no explicit constraint is permitted.

B490001

Check that the following are static expressions: (a) A numeric literal whose expected type is not a static subtype. (b) A string literal of a static string subtype. (c) X'First, X'Last, or X'Length, where X statically denotes a statically constrained array object or array subtype. (d) A type conversion whose subtype mark denotes a static scalar subtype, and whose operand is a static expression. (e) A membership test whose simple expression is a static expression, and whose range is a static range or whose subtype mark denotes a static subtype. (f) A short-circuit control form both of whose relations are static expressions. Check that the Range attribute of a statically constrained array subtype or object gives a static range. Check that a predefined concatenation operator whose result type is a string type is a static function. Check that a static expression is illegal if its evaluation fails a language-defined check other than Overflow_Check, even if it is part of a larger static expression. Check that

B490002

Check that a static string expression that is the result of a catenation is illegal if it has length greater than that permitted by the expected type. Check that it is illegal to assign a null string literal to an object whose lower bound is equal to the lower bound of the base range of the index type.

B610001

Check that access parameters may have default expressions. Check that the expected type of the actual access parameter is the nominal subtype of the formal anonymous access parameter.

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.

B660001

Check that the "=" operator may be overloaded for non-limited private types. Check that explicit overloadings of "/=" may not have a Boolean result. Check that the result of an explicitly declared "=" operator may be other than Boolean. Check that an equality operator may rename a function other than an equality operator.

B660002

Check that the "=" operator may be overloaded for non-limited types. Check that explicit overloadings of "/=" may not have a Boolean result. Check that the result of an explicitly declared "=" operator may be other than Boolean. Check that an equality operator may rename a function other than an equality operator. Check that a declaration of "=" whose result type is not Boolean does not implicitly declare a "/=" operation that gives the complementary result.

B7200010

See B7200016.A.

B7200011

See B7200016.A.

B7200012

See B7200016.A.

B7200013

See B7200016.A.

B7200014

See B7200016.A.

B7200015

See B7200016.A.

B7200016

Check that if a library package declaration or library generic package declaration does not require a body, that a body is not allowed. Check that pragma Elaborate_Body can be used to require a body even if not otherwise required.

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.

B730005

Check that the ancestor type of a private extension may not be a class-wide type. Check for the basic case. Check for the generic case, where the ancestor type is the class-wide type of a formal tagged private type or formal private extension. Check for the instance case, where the ancestor type is a formal (tagged or untagged) private type or formal private extension, and the corresponding actual type is a class-wide type. Verify this rule in the visible and private part of an instance. In the private part, check specifically for the case where the parents of the partial and full views are different.

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.

B810001

Check that a choice_parameter_specification in an exception handler hides outer declarations with the same name. Check that two choice parameters within exception handlers of the same handled_sequence_of_statements can have the same name. Check that a choice_parameter_specification in an exception handler is not visible outside the handler.

B830001

Check that two homographs are not allowed to be declared explicitly immediately within the same declarative region. Check for cases of child package names. Check for cases of dispatching operations declared in the visible part of an instance.

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.

B952004

Check that an entry_declaration in a task declaration cannot contain a specification for an access parameter. Check that an accept_statement is not allowed within an asynchronous_select inner to the enclosing task_body.

B954001

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

B954003

Check that the accessibility level of the target task object of a requeue_statement is not equal to or statically deeper than any enclosing accept_statement of the task unit. Check that for a requeue statement of an entry_body the target object is either a formal parameter of the entry_body or the accessibility level of the target object is not statically deeper that that of the entry_declaration.

B954004

Check that a requeue_statement is only allowed directly within an entry_body or accept_statement.

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.

BA11011

Check that the renamed entity must be a generic unit of the appropriate kind. Check that in a library_unit_renaming_declaration, the (old) name must denote a library_item. Check that a generic renaming of a child of a parent generic package is not allowed outside the declarative region of the parent generic. Check for subsequent renaming declarations of public children.

BA11012

Check that in a library_unit_renaming_declaration, the (old) name must denote a library_item. Check that a generic renaming of a child of a parent generic package is not allowed outside the declarative region of the parent generic. Check that a library unit must be a private descendent of the parent of a private child when the private child has been renamed and the name denoting the renaming has been used in a with clause. Check for subsequent renaming declarations of private children.

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.

BA15001

Check that configuration pragmas must appear before the first

compilation unit of a compilation.

BA21001

Check that each of the following constructs is illegal within a library package declaration to which a pragma Preelaborate applies: (a) A call to a nonstatic function. (b) A primary that is a name of an object, including within the default expression for a default-initialized component, if the name is not a static expression and does not statically denote a discriminant of an enclosing type. (c) A declaration of an object of a descendant of a task type. (d) A declaration of an object of a descendant of a controlled type without an initialization expression. (e) A declaration of an object with a component of a descendant of a private type (outside the scope of the full view) without an initialization expression. (f) An extension aggregate with an ancestor subtype mark denoting a subtype of a controlled type. Check that each of the following constructs is legal within a library package declaration to which a pragma Preelaborate applies: (g) A call to a static function. (h) A primary that is a name of an ob

BA21002

Check that each of the following constructs is illegal within the body of a library package to which a pragma Pure applies: (a) A statement other than a null statement. (b) A primary that is a name of an object, if the name is not a static expression and does not statically denote a discriminant of an enclosing type. (c) A declaration of an object of a descendant of a protected type with entry declarations. (d) A declaration of a variable, with or without an initialization expression, outside of a subprogram, generic subprogram, task unit, or protected unit. (e) A declaration of a named access type outside of a subprogram, generic subprogram, task unit, or protected unit. (f) An extension aggregate with an ancestor subtype mark denoting a subtype of a private extension. (g) A declaration of a (constant) object which causes the evaluation of a default expression that will call a user-defined function. Check that each of the following constructs is legal within the body of a library package to which a prag

BA210030

Check that all compilation units of a preelaborated library unit must depend semantically only on compilation units of other preelaborated library units. Check that all compilation units of a declared-pure library unit must depend semantically only on compilation units of other library units which are declared pure. Check that a preelaborated unit may have a non-preelaborable child unit, but not a non-preelaborable subunit.

BA210031

Check that all compilation units of a preelaborated library unit must depend semantically only on compilation units of other preelaborated library units. Check that all compilation units of a declared-pure library unit must depend semantically only on compilation units of other library units which are declared pure. Check that a preelaborated unit may have a non-preelaborable child unit, but not a

non-preelaborable subunit.

BA210032

Check that all compilation units of a preelaborated library unit must depend semantically only on compilation units of other preelaborated library units. Check that all compilation units of a declared-pure library unit must depend semantically only on compilation units of other library units which are declared pure. Check that a preelaborated unit may have a non-preelaborable child unit, but not a non-preelaborable subunit.

BA210033

Check that all compilation units of a preelaborated library unit must depend semantically only on compilation units of other preelaborated library units. Check that all compilation units of a declared-pure library unit must depend semantically only on compilation units of other library units which are declared pure. Check that a preelaborated unit may have a non-preelaborable child unit, but not a non-preelaborable subunit.

BA210034

Check that all compilation units of a preelaborated library unit must depend semantically only on compilation units of other preelaborated library units. Check that all compilation units of a declared-pure library unit must depend semantically only on compilation units of other library units which are declared pure. Check that a preelaborated unit may have a non-preelaborable child unit, but not a non-preelaborable subunit.

BA210035

Check that all compilation units of a preelaborated library unit must depend semantically only on compilation units of other preelaborated library units. Check that all compilation units of a declared-pure library unit must depend semantically only on compilation units of other library units which are declared pure. Check that a preelaborated unit may have a non-preelaborable child unit, but not a non-preelaborable subunit.

BA21A01

Check that an instantiation of a generic library package declaration to which a pragma Preelaborate applies is illegal if the instantiation occurs within a library package declaration to which a pragma Preelaborate also applies, and the generic library package contains any of the following constructs in its visible or private part (such that the construct is evaluated upon instantiation): (a) A call to a nonstatic function. (b) A call to a formal function, if the corresponding actual is a nonstatic function. (c) A primary that is a name of an object, if the name is not a static expression and does not statically denote a discriminant of an enclosing type. (d) A declaration of an object of a descendant of a protected type with entry declarations. (e) A declaration of an object of a descendant of a controlled type without an initialization expression. (f) A declaration

of an object of a descendant of a private type (outside the scope of the full view) without an initialization expression. (g) A declaration of

BA21A02

Check that the body corresponding to a generic library package declaration to which a pragma Preelaborate applies is illegal if it contains any of the following constructs (if the construct would be elaborated upon instantiation): (a) A statement other than a null statement. (b) A call to a nonstatic function. (c) A call to a formal function. (d) A primary that is a name of an object, including within the default expression for a default-initialized component, if the name is not a static expression and does not statically denote a discriminant of an enclosing type. (e) A declaration of an object of a descendant of a task type. (f) A declaration of an object with a component of a descendant of a controlled type without an initialization expression. (g) A declaration of an object with a component of a descendant of a private type (outside the scope of the full view) without an initialization expression. (h) A declaration of an object of a descendant of a private extension (outside the scope of the full view)

BB10001

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

BB20001

Check that an exception_name of a choice cannot denote an exception declared in a generic formal package.

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.

BC40002

Check that, for a generic formal object of mode in: If the formal object is of tagged type T, the type of the actual must be T. If the formal object is of type T'Class, the type of the actual must be a type in that class. Check that, for a generic formal object of mode in out: If the formal object is of tagged type T, the type of the actual must

be T. If the formal object is of type T'Class, the type of the actual must be T'Class.

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.

BC50003

Check that the actual corresponding to a formal signed integer type may not be a modular type. Check that the actual corresponding to a formal modular type may not be a signed integer type.

BC50004

Check that the actual corresponding to a formal ordinary fixed point type may not be a decimal fixed point type. Check that the actual corresponding to a formal decimal fixed point type may not be a ordinary fixed point type.

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.

BC53001

Check that the index subtypes of an unconstrained formal array subtype and its corresponding actual subtype must statically match. Check that the index ranges of a constrained formal array subtype and its corresponding actual subtype must statically match. Check that the component subtypes of a formal array type and its corresponding actual type must statically match.

BC53002

Check that if a formal array type has aliased components, the corresponding actual type must also have aliased components. Check that if a formal array type does not have aliased components, the corresponding actual type may nevertheless have aliased components.

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.

BDB0A01

Check that Storage_Size may not be specified for a derived

access_to_object type. Check that Storage_Pool may not be specified for a derived access_to_object type. Check that type Root_Storage_Pool is abstract, and requires overriding definitions for procedures Allocate, Deallocate and function Storage_Size. Check that Storage_Size may not be specified for a given access type if Storage_Pool is specified for it.

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.

BDE0001

Check that the explicit declaration of a primitive subprogram of a tagged type must occur before the type is frozen. Check for cases where the tagged type is frozen by: The declaration of a record extension (check also that a private extension does not freeze the parent type, and that freezing is deferred until the full type declaration). The declaration of an object of the type. An expression that is an allocator, the type of which designates the tagged type. Check that the tagged type is not frozen by a nonstatic expression that is part of a default expression.

BDE0002

Check that the explicit declaration of a primitive subprogram of a tagged type must occur before the type is frozen. Check for cases where the component type of a composite type is a tagged type, and the tagged type is frozen by: The declaration of an object of the composite type. An expression that is an allocator, the type of which designates the composite type. An expression that is an aggregate, which contains a composite value of the composite type. Check that the tagged type is not frozen by a nonstatic expression that is part of a default expression.

BDE0003

Check that the explicit declaration of a primitive subprogram of a tagged type must occur before the type is frozen. Check for cases where the tagged type is frozen by the completion of a deferred constant declaration. Check also that the deferred constant declaration itself does not freeze the type. Check that a deferred constant is completed before the constant is frozen.

BDE0004

Check that the explicit declaration of a primitive subprogram of a tagged type must occur before the type is frozen. Check for cases where the tagged type is frozen by the occurrence of a generic instantiation. Check that the tagged type is not frozen by a nonstatic expression that is a default name.

BDE0005

Check that the explicit declaration of a primitive subprogram of a

tagged type must occur before the type is frozen. Check for cases where the primitive subprogram occurs in a package body.

BDE0006

Check that a representation clause for a type must occur before the type is frozen. Check for cases where the type is frozen by: The declaration of an object of the type. The declaration of an object with a component of the type. The declaration of a record extension of the type. An expression that is an allocator, the type of which designates the type.

BDE0007

Check that a representation clause for an object or a type must occur before the object or type is frozen. Check for cases where the object or type is frozen by the occurrence of a generic instantiation. Check that an instance body does not cause freezing of entities declared before it within the same declarative part.

BDE0008

Check that a representation clause for a type must occur before the type is frozen. Check for cases where the type is frozen by a static expression or a nonstatic expression which is not a default expression. Check that a nonstatic expression that is part of a default expression does not cause freezing. Check for cases of subprogram renaming.

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 Attach_Handler will accept an expression only of type Interrupts.Interrupt_ID.

BXC3002

Check that pragmas Interrupt_Handler and Attach_Handler are recognized for protected types. Check that the pragmas are allowed only immediately in a protected definition. Check that a protected declaration for Attach_Handler must be library level. Check that a protected type declaration for Interrupt_Handler must be library level and that any object declaration of that type must be library level.

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.

BXC6002

Check that if an atomic object is used as an actual for a generic formal object of mode in out, the type of the generic formal object

must be atomic. Check that if the prefix of 'Access denotes an atomic object (including a component), the designated type of the resulting access type must be atomic.

BXC6003

Check that the implementation rejects a pragma Atomic when it cannot support indivisible reads or updates of the object. Check that the implementation rejects a pragma Atomic_Components when it cannot support indivisible reads or updates of the components of the array object.

BXC6A01

Check that if a volatile object is passed as a parameter, then the type of the formal parameter must not be a non-volatile by-reference type.

BXC6A02

Check that if a volatile object is used as an actual for a generic formal object of mode in out, the type of the generic formal object must be volatile. Check that if the prefix of 'Access denotes a volatile object (including a component), the designated type of the resulting access type must be volatile.

BXC6A03

Check that if a volatile type is used as an actual for a generic formal derived type, the ancestor of the formal type must not be a non-volatile by-reference type.

BXC6A04

Check that if a pragma Volatile, Volatile_Components, Atomic, or Atomic_Components applies to a stand-alone constant object, then a pragma Import must also apply to it. Check that if a stand-alone constant object is atomic or volatile solely because of its type, a pragma Import need not apply to it.

BXD1001

Check that a Priority pragma is allowed immediately within a task_definition, a protected_definition, and the declarative_part of a subprogram_body. Check that a Priority pragma is not allowed in other places. Check that an Interrupt_Priority pragma is allowed immediately within a task_definition or a protected_definition. Check that an Interrupt_Priority pragma is not allowed in the declarative part of a subprogram_body. Check that only one such pragma is allowed within a given construct.

BXD1002

Check that the pragma priority expression must be static when the pragma is located within the declarative_part of a subprogram_body. Check that the expression in a Priority and Interrupt_Priority pragma is required to be of type Integer. Check that the pragma priority expression need not be static when the pragma is located within a

task_definition or protected_definition.

BXE2007

Check that a declared Shared_Passive library unit may not contain: objects that are not preelaborable, library level task object declarations, protected objects with entries, access types that designate a class-wide type, access types that designate a task type, or access types that designate a protected type with entries. Check that a declared Shared_Passive library unit may contain: objects that are preelaborable, protected objects without entries, protected types with entries, and task types.

BXE2008

Check that a declared Remote_Types library unit may not contain: variable declarations; private types where the full view of the type contains a non-remote access type and no READ and WRITE attributes are supplied; visible access types where the type is neither an access-to-subprogram type nor a general access type that designates a class-wide limited private type. Check that a declared Remote_Types library unit may contain: private types where the full view of the type contains a non-remote access type and READ and WRITE attributes are supplied.

BXE2009

Check that a declared Remote_Call_Interface library unit may not contain: variable declarations, task type declarations, protected type declarations, nested generic declarations, limited types, subprogram declarations to which a pragma inline applies, non-preelaborable constant declarations, a subprogram declaration with a formal parameter of an access type, or a subprogram declaration with a formal parameter of a limited type without READ and WRITE attributes. Check that a Remote_Call_Interface library unit may not depend upon a shared passive or normal package. Check that a declared Remote_Call_Interface library unit may contain: subprogram declaration with a formal parameter of a limited type with READ and WRITE attributes. Check that pragma Asynchronous can only be applied to RCI procedures containing only mode in parameters.

BXE2010

Check that a public child library unit of a remote call interface library unit must itself have a Remote_Call_Interface pragma. Check that a private child library unit of a remote call interface library unit are not subjected to the restrictions of an RCI unit unless the private child unit contains a Remote_Call_Interface pragma. Check the parameterized form of the pragma to see that a library unit name can be specified and if it is specified, it must correspond to the library unit in which it is contained. Check that a public child library unit of a normal package cannot be a remote call interface unit. Check that a public child library unit of a pure package can be a remote call interface unit.

BXE2011

Check that a value of a remote access-to-subprogram type can only be converted to another conformant remote access-to-subprogram type. Check that the prefix of an access attribute_reference that yields a value of a remote access-to-subprogram type shall statically denote a conformant remote subprogram. Check that a value of a remote access-to-class-wide type can only be converted to another remote access-to-class-wide type. Check that the Storage_Pool and Storage_Size attributes are not defined for remote access-to-class-wide types.

BXE2012

Check that a remote access-to-class-wide type must designate a limited private type. Check that the primitive subprograms of the limited private type designated by a remote access-to-class-wide type can only have access parameters for the controlling parameters. Check that non-controlling parameters of limited private types designated by a remote access-to-class-wide type are required to have Read and Write attributes. Check that a value of a remote access-to-class-wide type can be implicitly converted only as part of a dispatching call where the value designates a controlling operand of the call.

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

BXE4001

Check that pragma Asynchronous can only be applied to one of the following three categories of items: Remote procedures where the formal parameters of the procedures are all of mode in; The first subtype of a remote access-to-procedure type where the formal parameters of the designated profile of the type are all of mode in; Remote access-to-class-wide types.

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

BXH4001

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Protected_Types disallows protected types in the compilations.

BXH4002

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Allocators disallows allocators in the compilations.

BXH4003

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Local_Allocators disallows allocators and generic package instantiations in subprograms, generic subprograms, tasks, and entry bodies. Check that allocators and generic instantiations are still allowed at the library package level.

BXH4004

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Unchecked_Deallocation disallows the use of Unchecked_Deallocation; Check that the application of the configuration pragma Restrictions with the specific restriction: Immediate_Reclamation is accepted.

BXH4005

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Exceptions is accepted.

BXH4006

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Floating_Point is accepted.

BXH4007

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Fixed_Point is accepted.

BXH4008

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Unchecked_Conversion does not allow the use of Unchecked_Conversion.

BXH4009

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Access_Subprograms is accepted.

BXH4010

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Unchecked_Access is accepted.

BXH4011

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Dispatch disallows occurrences of T'Class.

BXH4012

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_IO rejects any semantic dependence on Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO or Stream_IO.

BXH4013

Check pragma Restrictions. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Delay is accepted.

C250001

Check that wide character literals are supported. Check that wide character string literals are supported.

C250002

Check that characters in Latin-1 above ASCII.Del can be used in identifiers, character literals and strings.

C330001

Check that a variable object of an indefinite type is properly initialized/constrained by an initial value assignment that is a) an aggregate, b) a function, or c) an object. Check that objects of the above types do not need explicit constraints if they have initial values.

C330002

Check that if a subtype indication of a variable object defines an indefinite subtype, then there is an initialization expression. Check that the object remains so constrained throughout its lifetime. Check for cases of tagged record, arrays and generic formal type.

C332001

Check that the static expression given for a number declaration may be of any numeric type. Check that the type of a named number is universal_integer or universal_real regardless of the type of the static expression that provides its value.

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.

C352001

Check that the predefined Character type comprises 256 positions. Check that the names of the non-graphic characters are usable with the attributes (Wide_)Image and (Wide_)Value, and that these attributes produce the correct result.

C354002

Check that the attributes of modular types yield correct values/results. The attributes checked are: First, Last, Range, Base, Min, Max, Succ, Pred, Image, Width, Value, Pos, and Val

C354003

Check that the Wide_String attributes of modular types yield correct values/results. The attributes checked are: Wide_Image Wide_Value

C360002

Check that modular types may be used as array indices. Check that if aliased appears in the component_definition of an array_type that each component of the array is aliased. Check that references to aliased array objects produce correct results, and that out-of-bounds indexing correctly produces Constraint_Error.

C371001

Check that if a discriminant constraint depends on a discriminant, the evaluation of the expressions in the constraint is deferred until an object of the subtype is created. Check for cases of records with private type component.

C371002

Check that if a discriminant constraint depends on a discriminant, the evaluation of the expressions in the constraint is deferred until an object of the subtype is created. Check for cases of records.

C371003

Check that if a discriminant constraint depends on a discriminant, the evaluation of the expressions in the constraint is deferred until an object of the subtype is created. Check for cases of records where the component containing the constraint is present in the subtype.

C3900010

See C3900011.AM.

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

See C3900053.AM.

C3900051

See C3900053.AM.

C3900052

See C3900053.AM.

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

See C3900063.AM.

C3900061

See C3900063.AM.

C3900062

See C3900063.AM.

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.

C390007

Check that the tag of an object of a tagged type is preserved by type conversion and parameter passing.

C390010

Check that if S is a subtype of a tagged type T, and if S is constrained, then the allowable values of S'Class are only those that, when converted to T, belong to S.

C390011

Check that tagged types declared within generic package declarations generate distinct tags for each instance of the generic.

C390A010

See C390A011.AM.

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

See C390A022.AM.

C390A021

See C390A022.AM.

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

See C390A031.AM.

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.

C392002

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.

C392010

Check that a subprogram dispatches correctly with a controlling access parameter. Check that a subprogram dispatches correctly when it has access parameters that are not controlling. Check with and without default expressions.

C392011

Check that if a function call with a controlling result is itself a controlling operand of an enclosing call on a dispatching operation, then its controlling tag value is determined by the controlling tag value of the enclosing call.

C392012 (This test has been removed.)

Check that if all of the controlling operands of a call on a dispatching operation are tag indeterminate, then if the call has a controlling result and is a controlling operand of an enclosing call, then its controlling tag value is determined by the controlling tag value of the enclosing call.

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.

C393010

Check that an extended type can be derived from an abstract type and that a call on an abstract operation is a dispatching operation. Check that such a call can dispatch to an overriding operation declared in the private part of a package.

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.

C3A1001

Check that the full type completing a type with no discriminant part or an unknown discriminant part may have explicitly declared or inherited discriminants. Check for cases where the types are records and protected types.

C3A1002

Check that the full type completing a type with no discriminant part or an unknown discriminant part may have explicitly declared or inherited discriminants. Check for cases where the types are tagged records and task types.

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.

C3A2A01

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 cases where X'Access occurs in an instance body, and A is passed as an actual during instantiation.

C3A2A02

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 cases where X'Access occurs in an instance body, and A is a type either declared inside the instance, or declared outside the instance but not passed as an actual during instantiation.

C410001

Check that evaluating an access to subprogram variable containing the value null causes the exception Constraint_Error. Check that the default value for objects of access to subprogram types is null.

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.

C450001

Check that operations on modular types perform correctly. Check that loops over the range of a modular type do not over or under run the loop.

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.

C460004

Check that if the operand type of a type conversion is class-wide, Constraint_Error is raised if the tag of the operand does not identify a specific type that is covered by or descended from the target type.

C460005

Check that, for a view conversion of a tagged type that is the left side of an assignment statement, the assignment assigns to the corresponding part of the object denoted by the operand.

C460006

Check that a view conversion to a tagged type is permitted in the prefix of a selected component, an object renaming declaration, and (if the operand is a variable) on the left side of an assignment statement. Check that such a renaming or assignment does not change the tag of the operand. Check that, for a view conversion of a tagged type, each nondiscriminant component of the new view denotes the matching component of the operand object. Check that reading the value of the view yields the result of converting the value of the operand object to the target subtype.

C460007

Check that, in a numeric type conversion, if the target type is an integer type and the operand type is real, the result is rounded to the nearest integer, and away from zero if the result is exactly halfway between two integers. Check for static and non-static type conversions.

C460008

Check that conversion to a modular type raises Constraint_Error when the operand value is outside the base range of the modular type.

C460009

Check that Constraint_Error is raised in cases of null arrays when: 1. an assignment is made to a null array if the length of each dimension of the operand does not match the length of the corresponding dimension of the target subtype. 2. an array actual parameter does not match the length of corresponding dimensions of the formal in out parameter where the actual parameter has the form of a type conversion. 3. an array actual parameter does not match the length of corresponding dimensions of the formal out parameter where the actual parameter has the form of a type conversion.

C460010

Check that, for an array aggregate without an others choice assigned to an object of a constrained array subtype, Constraint_Error is not raised if the length of each dimension of the aggregate equals the length of the corresponding dimension of the target object, even if the bounds of the corresponding index ranges do not match.

C460A01

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 cases where the type conversion occurs in an instance body, and the operand type is passed as an actual during instantiation.

C460A02

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 cases where the type conversion occurs in an instance body, and the operand type is declared inside the instance or is the anonymous access type of an access parameter or access discriminant.

C490001

Check that, for a real static expression that is not part of a larger static expression, and whose expected type T is a floating point type that is not a descendant of a formal scalar type, the value is rounded to the nearest machine number of T if T'Machine_Rounds is true, and is truncated otherwise. Check that if rounding is performed, and the value is exactly halfway between two machine numbers, the rounding is performed away from zero.

C490002

Check that, for a real static expression that is not part of a larger static expression, and whose expected type T is an ordinary fixed point type that is not a descendant of a formal scalar type, the value is rounded to the nearest integral multiple of the small of T if T'Machine_Rounds is true, and is truncated otherwise. Check that if

rounding is performed, and the value is exactly halfway between two multiples of the small, the rounding is performed away from zero.

C490003

Check that a static expression is legal if its evaluation fails no language-defined check other than Overflow_Check. Check that such a static expression is legal if it is part of a larger static expression, even if its value is outside the base range of the expected type. Check that if a static expression is part of the right operand of a short circuit control form whose value is determined by its left operand, it is not evaluated. Check that a static expression in a non-static context is evaluated exactly.

C540001

Check that an expression in a case statement may be of a generic formal type. Check that a function call may be used as a case statement expression. Check that a call to a generic formal function may be used as a case statement expression. Check that a call to an inherited function may be used as a case statement expression even if its result type does not correspond to any nameable subtype.

C631001

Check that if different forms of a name are used in the default expression of a discriminant part, the selector may be an operator symbol or a character literal.

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.

C641001

Check that actual parameters passed by reference are view converted to the nominal subtype of the formal parameter.

C650001

Check that, for a function result type that is a return-by-reference type, Program_Error is raised if the return expression is a name that denotes an object view whose accessibility level is deeper than that of the master that elaborated the function body. Check for cases where the result type is: (a) A tagged limited type. (b) A task type. (c) A protected type. (d) A composite type with a subcomponent of a return-by-reference type (task type).

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.

C730004

Check that for a type declared in a package, descendants of the package use the full view of type. Specifically check that full view of the limited type is visible only in private descendants (children) and in the private parts and bodies of public descendants (children). Check that a limited type may be used as an out parameter outside the package that defines 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.

C760009

Check that for an extension_aggregate whose ancestor_part is a subtype_mark (i.e. Typemark'(Subtype with Field => x, etc.)) Initialize is called on all controlled subcomponents of the ancestor part; if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract. Check that the utilization of a controlled type for a generic actual parameter supports the correct behavior in the instantiated package.

C760010

Check that explicit calls to Initialize, Adjust and Finalize procedures that raise exceptions propagate the exception raised, not Program_Error. Check this for both a user defined exception and a language defined exception. Check that implicit calls to initialize procedures that raise an exception propagate the exception raised, not Program_Error; Check that the utilization of a controlled type as the actual for a generic formal tagged private parameter supports the correct behavior in the instantiated software.

C760011

Check that the anonymous objects of a controlled type associated with function results and aggregates are finalized no later than the end of the innermost enclosing declarative_item or statement. Also check this for function calls and aggregates of a noncontrolled type with controlled components.

C760012

Check that record components that have per-object access discriminant constraints are initialized in the order of their component declarations, and after any components that are not so constrained. Check that record components that have per-object access discriminant constraints are finalized in the reverse order of their component declarations, and before any components that are not so constrained.

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.

C761006

Check that Program_Error is raised when: * an exception is raised if Finalize invoked as part of an assignment operation; or * an exception is raised if Adjust invoked as part of an assignment operation, after any other adjustment due to be performed are performed; or * an exception is raised if Finalize invoked as part of a call on Unchecked_Deallocation, after any other finalizations to be performed are performed.

C761007

Check that if a finalize procedure invoked by a transfer of control due to selection of a terminate alternative attempts to propagate an exception, the exception is ignored, but any other finalizations due to be performed are performed.

C761008   (This test has been removed)

Check that when an exception occurs in a Finalize operation invoked by a "normal" transfer of control (exit, return, goto), Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Check that other finalizations due to be performed are performed prior to raising Program_Error. Check that for Finalize invoked by a transfer of control due to an exception, any other finalizations due to be performed for the same master are performed, then Program_Error is raised immediately after leaving the master. Check that other finalizations are performed first. Check that no other processing may occur after leaving the master.

C761009   (This test has been removed)

Check that when an exception occurs in a Finalize operation invoked by the transfer of control of a requeue statement, Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Check that other finalizations due to be performed are performed prior to raising Program_Error.

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.

C910002

Check that the contents of a task object include the values of its discriminants. Check that selected_component notation can be used to denote a discriminant of a task.

C930001

Check when a dependent task and its master both terminate as a result of a terminate alternative that finalization is performed and that the finalization is performed in the proper order.

C940001

Check that a protected object provides coordinated 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

C940004

Check that a protected record can be used to control access to resources (data internal to the protected record).

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.

C940014

Check that as part of the finalization of a protected object each call remaining on an entry queue of the objet is removed from its queue and Program_Error is raised at the place of the corresponding entry_call_statement.

C940015

Check that the component_declarations of a protected_operation are elaborated in the proper order. Check that per-object constraints are elaborated for each object.

C940016

Check that an Unchecked_Deallocation of a protected object performs the required finalization on the protected object.

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.

C953001

Check that if the evaluation of an entry_barrier condition propagates an exception, the exception Program_Error is propagated to all current callers of all entries of the protected object.

C953002

Check that the servicing of entry queues of a protected object continues until there are no open entries with queued calls and that this takes place as part of a single protected operation.

C953003

Check that the servicing of entry queues of a protected object continues until there are no open entries with queued (or requeued) calls and that internal requeues are handled as part of a single protected operation.

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.

C954025

Check that if the original entry call was a conditional entry call, the call is cancelled if a requeue-with-abort of the call is not selected immediately. Check that if the original entry call was a timed entry call, the expiration time for a requeue-with-abort is the original expiration time.

C954026

Check that if the original protected entry call was a conditional entry call, the call is cancelled if a requeue-with-abort of the call is not selected immediately. Check that if the original protected entry call was a timed entry call, the expiration time for a requeue-with-abort is the original expiration time.

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.

C980001

Check that when a construct is aborted the execution of an Initialize procedure as the last step of the default initialization of a controlled object is abort-deferred. Check that when a construct is aborted the execution of a Finalize procedure as part of the finalization of a controlled object is abort-deferred. Check that an assignment operation to an object with a controlled part is an abort-deferred operation.

C980002

Check that aborts are deferred during protected actions.

C980003

Check that aborts are deferred during the execution of an Initialize procedure (as the last step of the default initialization of a controlled object), during the execution of a Finalize procedure (as part of the finalization of a controlled object), and during an assignment operation to an object with a controlled part.

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

See CA110042.AM

CA110041

See CA110042.AM

CA110042

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

CA110050

See CA110051.AM

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

See CA11D013.AM

CA11D011

See CA11D013.AM

CA11D012

See CA11D013.AM

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.

CB40005

Check that exceptions raised in non-generic code can be handled by a procedure in a generic package. Check that the exception identity can be properly retrieved from the generic code and used by the non-generic code.

CB40A01

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

CB40A020

See CB40A021.AM.

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

See CB40A031.AM.

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.

CC30002

Check that an explicit declaration in the private part of an instance does not override an implicit declaration in the instance, unless the corresponding explicit declaration in the generic overrides a corresponding implicit declaration in the generic. Check for primitive subprograms of tagged types.

CC40001

Check that adjust is called on the value of a constant object created by the evaluation of a generic association for a formal object of mode in. Check that those values are also subsequently finalized.

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.

CD10001

Check that representation items may contain nonstatic expressions in the case that each expression in the representation item is a name that statically denotes a constant declared before the entity.

CD20001

Check that for packed records the components are packed as tightly as possible subject to the Size of the component subtypes. Specifically check that Boolean objects are packed one to a bit. Check that the Component_Size for a packed array type is the value which is less than or equal to the Size of the component type, rounded up to the nearest factor of the word size.

CD30001

Check that X'Address produces a useful result when X is an aliased object. Check that X'Address produces a useful result when X is an object of a by-reference type. Check that X'Address produces a useful result when X is an entity whose Address has been specified. Check that aliased objects and subcomponents are allocated on storage element boundaries. Check that objects and subcomponents of by reference types are allocated on storage element boundaries. Check that for an array X, X'Address points at the first component of the array, and not at the array bounds.

CD30002

Check that the implementation supports Alignments for subtypes and objects specified as factors and multiples of the number of storage elements per word, unless those values cannot be loaded and stored. Check that the largest alignment returned by default is supported. Check that the implementation supports Alignments supported by the target linker for stand-alone library-level objects of statically constrained subtypes.

CD30003

Check that a Size clause for an object is supported if the specified size is at least as large as the subtype's size, and correspond to a

size in storage elements that is a multiple of the object's (non-zero) Alignment.

CD30004

Check that the unspecified Size of static discrete subtypes is the number of bits needed to represent each value belonging to the subtype using an unbiased representation, where space for a sign bit is provided only in the event the subtype contains negative values. Check that for first subtypes specified Sizes are supported reflecting this representation. Check that for subtypes implemented with levels of indirection, the Size includes the size of the pointers, not the size of what is pointed at.

CD30005

Check that Address clauses are supported for imported subprograms.

CD33001

Check that Component_Sizes that are a factor of the word size are supported. Check that for such Component_Sizes arrays contain no gaps between components.

CD33002

Check that Component_Sizes that are multiples of the word size are supported. Check that for such Component_Sizes arrays contain no gaps between components.

CD40001

Check that Enumeration_Representation_Clauses are supported for codes in the range System.Min_Int..System.Max_Int.

CD70001

Check that package System includes Max_Base_Digits, Address, Null_Address, Word_Size, functions "<", "<=", ">", ">=", "=" (with Address parameters and Boolean results), Bit_Order, Default_Bit_Order, Any_Priority, Interrupt_Priority, and Default_Priority. Check that package System.Storage_Elements includes all required types and operations.

CD72A01

Check that the package System.Address_To_Access_Conversions may be instantiated for various simple types. Check that To_Pointer and To_Address are inverse operations. Check that To_Pointer(X'Address) equals X'Unchecked_Access for an X that allows Unchecked_Access. Check that To_Pointer(Null_Address) returns null.

CD72A02

Check that the package System.Address_To_Access_Conversions may be instantiated for various composite types. Check that To_Pointer and To_Address are inverse operations. Check that To_Pointer(X'Address)

equals X'Unchecked_Access for an X that allows Unchecked_Access. Check that To_Pointer(Null_Address) returns null.

CD90001

Check that Unchecked_Conversion is supported and is reversible in the cases where: Source'Size = Target'Size
Source'Alignment = Target'Alignment Source and Target are both represented contiguously Bit pattern in Source is a meaningful value of Target type

CD92001

Check that if X denotes a scalar object, X'Valid yields true if an only if the object denoted by X is normal and has a valid representation.

CDB0A01

Check that a storage pool may be user_determined, and that storage is allocated by calling Allocate. Check that a storage.pool may be specified using 'Storage_Pool and that S'Storage_Pool denotes the storage pool of the type S.

CDB0A02

Check that several access types can share the same pool. Check that any exception propagated by Allocate is propagated by the allocator. Check that for an access type S, S'Max_Size_In_Storage_Elements denotes the maximum values for Size_In_Storage_Elements that will be requested via Allocate.

CDE0001

Check that the following names can be used in the declaration of a generic formal parameter (object, array type, or access type) without causing freezing of the named type: (1) The name of a private type, (2) A name that denotes a subtype of a private type, and (3) A name that denotes a composite type with a subcomponent of a private type (or subtype). Check for untagged and tagged types.

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 "*".

CXA4024

Check that the function "-", To_Ranges, To_Domain, and To_Range are available in the package Ada.Strings.Maps, and that they produce correct results based on the Character_Set/Character_Mapping input provided.

CXA4025

Check that the functionality found in packages Ada.Strings.Wide_Maps, Ada.Strings.Wide_Fixed, and Ada.Strings.Wide_Maps.Wide_Constants is available and produces correct results.

CXA4026

Check that Ada.Strings.Fixed procedures Head, Tail, and Trim, as well as the versions of subprograms Translate (procedure and function), Index, and Count, available in the package which use a Maps.Character_Mapping_Function input parameter, produce correct results.

CXA4027

Check that versions of Ada.Strings.Bounded subprograms Translate, (procedure and function), Index, and Count, which use the Maps.Character_Mapping_Function input parameter, produce correct results.

CXA4028

Check that Ada.Strings.Bounded procedures Append, Head, Tail, and Trim, and relational operator functions "=", ">", ">=", "<", "<=" with parameter combinations of type String and Bounded_String, produce correct results.

CXA4029

Check that the functionality found in packages Ada.Strings.Wide_Maps, Ada.Strings.Wide_Bounded, and Ada.Strings.Wide_Maps.Wide_Constants is available and produces correct results.

CXA4030

Check that Ada.Strings.Unbounded versions of subprograms Translate (procedure and function), Index, and Count, which use a Maps.Character_Mapping_Function input parameter, produce correct results.

CXA4031

Check that the subprograms defined in package Ada.Strings.Unbounded are available, and that they produce correct results. Specifically, check the functions To_Unbounded_String (version with Length parameter), "=", "<", "<=", ">", ">=" (all with String-Unbounded String parameter mix), as well as three versions of Procedure Append.

CXA4032

Check that procedures defined in package Ada.Strings.Unbounded are available, and that they produce correct results. Specifically, check the procedures Replace_Slice, Insert, Overwrite, Delete, Trim (2 versions), Head, and Tail.

CXA4033

Check that the functionality found in packages Ada.Strings.Wide_Maps, Ada.Strings.Wide_Unbounded, and Ada.Strings.Wide_Maps.Wide_Constants is available and produces correct results.

CXA5011

Check that, for both Float_Random and Discrete_Random packages, the following are true: 1) two objects of type Generator are initialized to the same state. 2) when the Function Reset is used to reset two generators to different time-dependent states, the resulting random values from each generator are different. 3) when the Function Reset uses the same integer initiator to reset two generators to the same state, the resulting random values from each generator are identical. 4) when the Function Reset uses different integer initiator values to reset two generators, the resulting random numbers are different.

CXA5012

Check that, for both Float_Random and Discrete_Random packages, the following are true: 1) the procedures Save and Reset can be used to save the specific state of a random number generator, and then restore the specific state to the generator following some intermediate generator activity. 2) the Function Image can be used to obtain a string representation of the state of a generator; and that the Function Value will transform a string representation of the state of a random number generator into the actual state object. 3) a call to Function Value, with a string value that is not the image of any generator state, will raise Constraint_Error.

CXA5013

Check that a discrete random number generator will yield each value in its result subtype in a finite number of calls, provided that the number of such values does not exceed 2**15.

CXA5015

Check that the following representation-oriented attributes are available and that the produce correct results: 'Denorm, 'Signed_Zeros, 'Exponent 'Fraction, 'Compose, 'Scaling, 'Floor, 'Ceiling, 'Rounding, 'Unbiased_Rounding, 'Truncation, 'Remainder, 'Adjacent, 'Copy_Sign, 'Leading_Part, 'Machine, and 'Model_Small.

CXA5A01

Check that the functions Sin and Sinh provide correct results.

CXA5A02

Check that the functions Cos and Cosh provide correct results.

CXA5A03

Check that the functions Tan, Tanh, and Arctanh provide correct results.

CXA5A04

Check that the functions Cot, Coth, and Arccoth provide correct results.

CXA5A05

Check that the functions Arcsin and Arcsinh provide correct results.

CXA5A06

Check that the functions Arccos and Arccosh provide correct results.

CXA5A07

Check that the function Arctan provides correct results.

CXA5A08

Check that the function Arccot provides correct results.

CXA5A09

Check that the function Log provides correct results.

CXA5A10

Check that the functions Exp and Sqrt, and the exponentiation operator "**" provide correct results.

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.

CXAA016

Check that the type File_Access is available in Ada.Text_IO, and that objects of this type designate File_Type objects. Check that function Set_Error will set the current default error file. Check that versions of Ada.Text_IO functions Standard_Input, Standard_Output, Standard_Error return File_Access values designating the standard system input, output, and error files. Check that versions of Ada.Text_IO functions Current_Input, Current_Output, Current_Error return File_Access values designating the current system input, output, and error files.

CXAA017

Check that Ada.Text_IO function Look_Ahead sets parameter End_Of_Line to True if at the end of a line; otherwise check that it returns the next character from a file (without consuming it), while setting End_Of_Line to False. Check that Ada.Text_IO function Get_Immediate will return the next control or graphic character in parameter Item from the specified file. Check that the version of Ada.Text_IO function Get_Immediate with the Available parameter will, if a character is available in the specified file, return the character in parameter Item, and set parameter Available to True.

CXAA018

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

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.

CXAC004

Check that the Stream_Access type and Stream function found in package Ada.Text_IO.Text_Streams allows a text file to be processed with the functionality of streams.

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.

CXAF001

Check that an implementation supports the functionality defined in Package Ada.Command_Line.

CXB2001

Check that subprograms Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left, and Rotate_Right are available and produce correct results for values of signed and modular integer types of 8 bits.

CXB2002

Check that subprograms Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left, and Rotate_Right are available and produce correct results for values of signed and modular integer types of 16 bits.

CXB2003

Check that subprograms Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left, and Rotate_Right are available and produce correct results for values of signed and modular integer types of 32 bits.

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.

CXB3004

Check that the functions To_C and To_Ada map between the Ada type Character and the C type char. Check that the function Is_Nul_Terminated returns True if the char_array parameter contains nul, and otherwise False. Check that the function To_C produces a correct char_array result, with lower bound of 0, and length dependent upon the Item and Append_Nul parameters. Check that the function To_Ada produces a correct string result, with lower bound of 1, and length dependent upon the Item and Trim_Nul parameters. Check that the function To_Ada raises Terminator_Error if the parameter Trim_Nul is set to True, but the actual Item parameter does not contain the nul char.

CXB3005

Check that the procedure To_C converts the character elements of a string parameter into char elements of the char_array parameter Target, with nul termination if parameter Append_Nul is true. Check that the out parameter Count of procedure To_C is set to the appropriate value for both the nul/no nul terminated cases. Check that Constraint_Error is propagated by procedure To_C if the length of the char_array parameter Target is not sufficient to hold the converted string value. Check that the Procedure To_Ada converts char elements of the char_array parameter Item to the corresponding character elements of string out parameter Target. Check that Constraint_Error is propagated by Procedure To_Ada if the length of string parameter Target is not long enough to hold the converted char_array value. Check that Terminator_Error is propagated by Procedure To_Ada if the parameter Trim_Nul is set to True, but the actual Item parameter contains no nul char.

CXB3006

Check that the function To_C maps between the Ada type Wide_Character and the C type wchar_t. Check that the function To_Ada maps between the C type wchar_t and the Ada type Wide_Character. Check that the

function Is_Nul_Terminated returns True if the wchar_array parameter contains wide_nul, and otherwise False. Check that the function To_C produces a correct wchar_array result, with lower bound of 0, and length dependent upon the Item and Append_Nul parameters. Check that the function To_Ada produces a correct wide_string result, with lower bound of 1, and length dependent upon the Item and Trim_Nul parameters. Check that the function To_Ada raises Terminator_Error if the parameter Trim_Nul is set to True, but the actual Item parameter does not contain the wide_nul wchar_t.

CXB3007

Check that the procedure To_C converts the Wide_Character elements of a Wide_String parameter into wchar_t elements of the wchar_array parameter Target, with wide_nul termination if parameter Append_Nul is true. Check that the out parameter Count of procedure To_C is set to the appropriate value for both the wide_nul/no wide_nul terminated cases. Check that Constraint_Error is propagated by procedure To_C if the length of the wchar_array parameter Target is not sufficient to hold the converted Wide_String value. Check that the Procedure To_Ada converts wchar_t elements of the wchar_array parameter Item to the corresponding Wide_Character elements of Wide_String out parameter Target. Check that Constraint_Error is propagated by Procedure To_Ada if the length of Wide_String parameter Target is not long enough to hold the converted wchar_array value. Check that Terminator_Error is propagated by Procedure To_Ada if the parameter Trim_Nul is set to True, but the actual Item parameter contains no wide_n

CXB3008

Check that functions imported from the C language <string.h> and <stdlib.h> libraries can be called from an Ada program.

CXB3009

Check that the function To_Chars_Ptr will return a Null_Ptr value when the parameter Item is null. If the parameter Item is not null, and references a chars_array object that does contain the char nul, and parameter Nul_Check is True, check that To_Chars_Ptr performs a pointer conversion from char_array_access type to chars_ptr type. Check that if parameter Item is not null, and references a chars_array object that does not contain nul, and parameter Nul_Check is True, the To_Chars_Ptr function will propagate Terminator_Error. Check that the New_Char_Array function will return a chars_ptr type pointer to an allocated object that has been initialized with the value of parameter Chars. Check that the function New_String returns a chars_ptr initialized to a nul-terminated string having the value of the Str parameter.

CXB3010

Check that the Procedure Free resets the parameter Item to Null_Ptr. Check that Free has no effect if Item is Null_Ptr. Check that the version of Function Value with a chars_ptr parameter returning a char_array result returns the prefix of an array of chars. Check that the version of Function Value with a chars_ptr parameter and a size_t parameter returning a char_array result returns the shorter of:

1) the first size_t number of characters, or 2) the characters up to and including the first nul. Check that both of the above versions of Function Value propagate Dereference_Error if the Item parameter is Null_Ptr.

CXB3011

Check that the version of Function Value with a chars_ptr parameter that returns a String result returns an Ada string containing the characters pointed to by the chars_ptr parameter, up to (but not including) the terminating nul. Check that the version of Function Value with a chars_ptr parameter and a size_t parameter that returns a String result returns the shorter of: 1) a String of the first size_t number of characters, or 2) a String of characters up to (but not including) the terminating nul. Check that the Function Strlen returns a size_t result that corresponds to the number of chars in the array pointed to by Item, up to but not including the terminating nul. Check that both of the above versions of Function Value and Function Strlen propagate Dereference_Error if the Item parameter is Null_Ptr.

CXB3012

Check that Procedure Update modifies the value pointed to by the chars_ptr parameter Item, starting at the position corresponding to parameter Offset, using the chars in char_array parameter Chars. Check that the version of Procedure Update with a String parameter behaves in the manner described above, but with the character values in the String overwriting the char values in Item. Check that both of the above versions of Procedure Update will propagate Update_Error if Check is True, and if the length of the new chars in Chars, when overlaid starting from position Offset, will overwrite the first nul in Item.

CXB30132

Check that imported, user-defined C language functions can be called from an Ada program.

CXB3014

Check that the Function Value with Pointer and Element parameters will return an Element_Array result of correct size and content (up to and including the first "terminator" Element). Check that the Function Value with Pointer and Length parameters will return an Element_Array result of appropriate size and content (the first Length elements pointed to by the parameter Ref). Check that both versions of Function Value will propagate Interfaces.C.Strings.Dereference_Error when the value of the Ref pointer parameter is null.

CXB3015

Check that the "+" and "-" functions with Pointer and ptrdiff_t parameters that return Pointer values produce correct results, based on the size of the array elements. Check that the "-" function with two Pointer parameters that returns a ptrdiff_t type parameter produces correct results, based on the size of the array elements. Check that each of the "+" and "-" functions above will propagate

Pointer_Error if a Pointer parameter is null. Check that the Increment and Decrement procedures provide the correct "pointer arithmetic" operations.

CXB3016

Check that function Virtual_Length returns the number of elements in the array referenced by the Pointer parameter Ref, up to (but not including) the (first) instance of the element specified in the Terminator parameter. Check that the procedure Copy_Terminated_Array copies the array of elements referenced by Pointer parameter Source, into the array pointed to by parameter Target, based on which of the following two scenarios occurs first: 1) copying the Terminator element, or 2) copying the number of elements specified in parameter Limit. Check that procedure Copy_Terminated_Array will propagate Dereference_Error if either the Source or Target parameter is null. Check that procedure Copy_Array will copy an array of elements of length specified in parameter Length, referenced by the Pointer parameter Source, into the array pointed to by parameter Target. Check that procedure Copy_Array will propagate Dereference_Error if either the Source or Target parameter is null.

CXB4001

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

CXB4002

Check that the procedure To_COBOL converts the character elements of the String parameter Item into COBOL_Character elements of the Alphanumeric type parameter Target, using the Ada_to_COBOL mapping as the basis of conversion. Check that the parameter Last contains the index of the last element of parameter Target that was assigned by To_COBOL. Check that Constraint_Error is propagated by procedure To_COBOL when the length of String parameter Item exceeds the length of Alphanumeric parameter Target. Check that the procedure To_Ada converts the COBOL_Character elements of the Alphanumeric parameter Item into Character elements of the String parameter Target, using the COBOL_to_Ada mapping array as the basis of conversion. Check that the parameter Last contains the index of the last element of parameter Target that was assigned by To_Ada. Check that Constraint_Error is propagated by procedure To_Ada when the length of Alphanumeric parameter Item exceeds the length of String parameter Target.

CXB4003

Check that function Valid, with the Display_Format parameter set to Unsigned, will return True if Numeric parameter Item comprises one or more decimal digit characters; check that it returns False if the parameter Item is otherwise comprised. Check that function Valid, with Display_Format parameter set to Leading_Separate, will return True if Numeric parameter Item comprises a single occurrence of a Plus_Sign or Minus_Sign character, and then by one or more decimal digit characters; check that it returns False if the parameter Item is otherwise comprised. Check that function Valid, with Display_Format parameter set to Trailing_Separate, will return True if Numeric

parameter Item comprises one or more decimal digit characters, and then by a single occurrence of the Plus_Sign or Minus_Sign character; check that it returns False if the parameter Item is otherwise comprised.

CXB4004

Check that function Length, with Display_Format parameter, will return the minimal length of a Numeric value that will be required to hold the largest value of type Num represented as Format. Check that function To_Decimal will produce a decimal type Num result that corresponds to parameter Item as represented by parameter Format. Check that function To_Decimal propagates Conversion_Error when the value represented by parameter Item is outside the range of the Decimal_Type Num used to instantiate the package Decimal_Conversions Check that function To_Display returns a Numeric type result that represents Item under the specific Display_Format. Check that function To_Display propagates Conversion_Error when parameter Item is negative and the specified Display_Format parameter is Unsigned.

CXB4005

Check that the function To_COBOL will convert a String parameter value into a type Alphanumeric array of COBOL_Characters, with lower bound of one, and length equal to length of the String parameter, based on the mapping Ada_to_COBOL. Check that the function To_Ada will convert a type Alphanumeric parameter value into a String type result, with lower bound of one, and length equal to the length of the Alphanumeric parameter, based on the mapping COBOL_to_Ada. Check that the Ada_to_COBOL and COBOL_to_Ada mapping arrays provide a mapping capability between Ada's type Character and COBOL run-time character sets.

CXB4006

Check that the function Valid with Packed_Decimal and Packed_Format parameters returns True if Item (the Packed_Decimal parameter) has a value consistent with the Packed_Format parameter. Check that the function Length with Packed_Format parameter returns the minimal length of a Packed_Decimal value sufficient to hold any value of type Num when represented according to parameter Format. Check that the function To_Decimal with Packed_Decimal and Packed_Format parameters produces a decimal type value corresponding to the Packed_Decimal parameter value Item, under the conditions of the Packed_Format parameter Format. Check that the function To_Packed with Decimal (Num) and Packed_Format parameters produces a Packed_Decimal result that corresponds to the decimal parameter under conditions of the Packed_Format parameter. Check that Conversion_Error is propagated by function To_Packed if the value of the decimal parameter Item is negative and the specified Packed_Format parameter is Packed_Unsigned.

CXB4007

Check that the function Valid with Byte_Array and Binary_Format parameters returns True if the Byte_Array parameter corresponds to any value inside the range of type Num. Check that function Valid returns False if the Byte_Array parameter corresponds to a value outside the

range of Num. Check that function Length with Binary_Format parameter will return the minimum length of a Byte_Array value required to hold any value of decimal type Num. Check that function To_Decimal with Byte_Array and Binary_Format parameters will return a decimal type value that corresponds to parameter Item (of type Byte_Array) under the specified Format. Check that Conversion_Error is propagated by function To_Decimal if the Byte_Array parameter Item represents a decimal value outside the range of decimal type Num. Check that function To_Binary will produce a Byte_Array result that corresponds to the decimal type parameter Item, under the specified Binary_Format.

CXB4008

Check that the function To_Decimal with Binary parameter will return the corresponding value of the decimal type Num. Check that the function To_Decimal with Long_Binary parameter will return the corresponding value of the decimal type Num. Check that both of the To_Decimal functions described above will propagate Conversion_Error if the converted value Item is outside the range of type Num. Check that the function To_Binary converts a value of the Ada decimal type Num into a Binary type value. Check that the function To_Long_Binary converts a value of the Ada decimal type Num into a Long_Binary type value.

CXB40093

Check that using Pragma Import (which references a COBOL subprogram) as a completion of a procedure declaration will allow the use of the imported subprogram by the calling routine.

CXB5001

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

CXB5002

Check that the Function To_Fortran with a Character parameter will return the corresponding Fortran Character_Set value. Check that the Function To_Ada with a Character_Set parameter will return the corresponding Ada Character value. Check that the Function To_Fortran with a String parameter will return the corresponding Fortran_Character value. Check that the Function To_Ada with a Fortran_Character parameter will return the corresponding Ada String value.

CXB5003

Check that the procedure To_Fortran converts the character elements of the String parameter Item into Character_Set elements of the Fortran_Character type parameter Target. Check that the parameter Last contains the index of the last element of parameter Target that was assigned by To_Fortran. Check that Constraint_Error is propagated by procedure To_Fortran when the length of String parameter Item exceeds the length of Fortran_Character parameter Target. Check that the procedure To_Ada converts the Character_Set elements of the Fortran_Character parameter Item into Character elements of the String parameter Target. Check that the parameter Last contains the index of

the last element of parameter Target that was assigned by To_Ada.
Check that Constraint_Error is propagated by procedure To_Ada when the
length of Fortran_Character parameter Item exceeds the length of
String parameter Target.

C

C   See CXB50042.AM C

C

C   See CXB50042.AM C

CXB50042

Check that using Pragma Import (which references a Fortran subprogram)
as a completion of a subprogram declaration will allow the use of the
imported subprogram by the calling routine.

C

C   See CXB50052.AM C

C

C   See CXB50052.AM C

CXB50052

Check that using Pragmas Import and Convention allow modification of an
array in Fortran's column-major order.

CXC3001

Check that Is_Attached returns False for all non-reserved interrupts to
which no user-defined handler has been attached. Check that a
user-defined handler can be attached to every interrupt for which
Is_Reserved returns False. Check that Is_Attached returns True for all
non-reserved interrupts to which a user-defined handler has been
attached. Check that if Detach_Handler is subsequently called for such
an interrupt, Is_Attached returns False. Check that, for procedures
Attach_Handler and Exchange_Handler, if the parameter New_Handler
designates a protected procedure to which the pragma Interrupt_Handler
does not apply, Program_Error is raised and the existing interrupt
treatment is not modified.

CXC3002

Check that Program_Error is raised if the interrupt corresponding to
that specified by the expression in pragma Attach_Handler is reserved.

CXC3003

Check that when a protected object is finalized, for any of its
procedures that are attached to interrupts, the handler is detached.
Check that if the handler was attached by a pragma Attach_Handler, the
previous handler is restored.

CXC3004

Check that an exception propagated from a handler invoked by an interrupt has no effect. Check that the exception causes further execution of the handler to be abandoned.

CXC3005

Check that Program_Error is raised if an actual parameter of type Ada.Interrupts.Interrupt_ID is passed in a call to any of the following operations in package Ada.Interrupts, and the specified interrupt is reserved: Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler.

CXC3006

Check that Program_Error is raised if, by using the Ada.Interrupts procedure Attach_Handler, Detach_Handler, or Exchange_Handler, an attempt is made to detach an interrupt handler that was attached using the pragma Attach_Handler. Check that, in each case, the handler attached by the pragma is not detached.

CXC3007

Check that if the actual parameter corresponding to the formal parameter New_Handler in a call to either of the procedures Ada.Interrupts.Attach_Handler or Ada.Interrupts.Exchange_Handler has one of the following values, the default treatment for the specified interrupt is restored: The value null. The value returned by the function Current_Handler when no user-defined handler is attached to the specified interrupt.

CXC3008

Check that the procedures Ada.Interrupts.Attach_Handler and Ada.Interrupts.Exchange_Handler attach a specified handler to a specified interrupt, overriding any existing treatment. Check that, for Exchange_Handler, the value returned in Old_Handler designates the previous treatment for the interrupt. Check that the procedure Ada.Interrupts.Current_Handler returns a value that represents the attached handler of the specified interrupt. Check that the procedure Ada.Interrupts.Detach_Handler restores the default treatment for the specified interrupt. Check that an attached handler is called once for each delivered interrupt occurrence.

CXC3009

Check that an exception propagated from a handler invoked by an interrupt has no effect. Check that the exception causes further execution of the handler to be abandoned.

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

For volatile composite types that are not by-copy types, and types with volatile subcomponents: check that parameters are passed by copy when an actual parameter is defined as volatile, and the formal parameter is not.

CXC6003

Check that all reads and updates of atomic and volatile objects are performed directly to memory. Check that reads and updates of atomic objects are indivisible. Check that pragma Pack and pragma Atomic_Components can be used together.

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.

CXC7002

Check that when an instance of package Task_Attributes is elaborated, an object of the actual type corresponding to the formal type Attribute is implicitly created for each task that exists and is not yet terminated. Check that Value returns the value set by Set_Value. Check that Tasking_Error is raised if a Task_Attributes operation is attempted on a terminated task. Check that Program_Error is raised if a Task_Attributes operation is attempted on a null Task_Id.

CXC7003

Check that the Task_Attributes operations Set_Value and Reinitialize performs finalization on the old value of the attribute of the specified task.

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 subprogram 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.

CXD1006

Check that if there is no expression in an Interrupt_Priority pragma that the priority value is Interrupt_Priority'Last.

CXD1007

Check that a priority pragma has no effect if it occurs in the declarative_part of a subprogram_body other than the main subprogram. Check that the priority specified for the main subprogram sets the priority of the environment task. Check that dynamic values can be specified in the interrupt_priority and priority pragmas.

CXD1008

Check that task scheduling, floating point operations, and exceptions work properly together.

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.

CXD2002

Check that when Task_Dispatching_Policy is FIFO_Within_Priorities and a task executes a delay statement that does not result in blocking, it 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.

CXD2005 (This test has been removed)

Check that when the active priority of a ready task that is not running changes that the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority.

CXD2006

Check that priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists.

CXD2007

Check that a new running task is selected whenever there is a nonempty ready queue with a higher priority than the priority of the running task. Check that when a task is preempted it is added to the head of the ready queue for its active priority.

CXD2008

Check that if the Task_Dispatching_Policy is FIFO_Within_Priorities and a blocked task becomes ready then it is added to the tail of the ready queue for its priority.

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

CXD3003

Check that when Locking_Policy is Ceiling_Locking and no pragma Priority or Interrupt_Priority is specified in a protected definition

but a pragma Interrupt_Handler is specified, the ceiling priority is in the range of System.Interrupt_Priority.

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.

CXD4007

Check that when multiple entry_barriers of a protected object become True and more than one of the respective queues are nonempty, the call with the highest priority is selected. Check that a minimum of 30 different priorities can be specified and that the priorities make a difference in the task scheduling.

CXD4008

Check that when: multiple entry_barriers of a protected object become True, more than one of the respective queues are nonempty, and the callers are all of the same priority then the entries are taken in textual order. Check that when: multiple alternatives of a selective_accept have queued callers and the callers are all of the same priority then the accept_alternative that is textually first in

the selective_accept is selected.

CXD4009

Check that when multiple alternatives of a selective_accept have queued callers and the callers are all of different priority then the accept_alternative that has the highest priority task waiting is selected.

CXD4010

Check that if the expiration time of two open delay_alternatives is the same and no other accept_alternatives are open then the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed.

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.

CXD5002 (This test has been removed)

Check that when setting a task's base priority to a new value that the new value does not take effect while the task is performing a protected action.

CXD6001

Check that an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation.

CXD6002

Check that in an asynchronous transfer of control an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation.

CXD6003

Check that in an asynchronous transfer of control an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation where the abort-deferred operation is the default initialization and finalization of a controlled object, or an assignment to a controlled type object.

CXD8001

Check the basic functions in the Package Ada.Real_Time.

CXD8002

Check that Ada.Real_Time.Time can be used in a delay_until_statement. Check that a delay_statement blocks the task for at least as long as the requested delay as measured by Real_Time.Clock.

CXD8003

Check that the Ada.Real_Time package operations Split and Time_Of operations work properly. Check that the clock does not jump backwards.

CXD9001

Check that when a delay_statement appears in a delay_alternative of a timed_entry_call the entry call is attempted regardless of the specified expiration 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.

CXDA003

Check that Set_False and Set_True can be called during a protected operation that has its ceiling priority in the Interrupt_Priority range.

CXDA004

Check that Set_False and Set_True can be called from an interrupt handler. Check that a Suspension_Object is a by reference type. Check that Current_State returns the current state of a suspension object. Check that Program_Error is raised upon calling Suspend_Until_True if another task is waiting on that 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.

CXDB005 (This test has been removed)

Check that Hold-ing a task causes that task to no longer actively contribute to the priority inheritance of other tasks.

CXDC001 (This test has been removed)

Check that Unchecked_Deallocation is supported for terminated tasks that are designated by access types and has the effect of releasing all the storage associated with the task.

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.

CXE2001

Check that only one copy of the data in a shared passive library unit is present in a program. Check that a protected object declared in a shared passive library unit can be used from both partitions of a two partition program.

CXE4001

Check that exception propagation between partitions is properly handled. In particular check that: a predefined exception can be raised in one partition and handled in another; an exception declared in a remote call interface library unit can be raised in one partition and handled in another partition; an exception declared in partition A and not visible to partition B can be raised in partition A and handled in partition B with an others clause; an exception declared in a partition A and not visible to partition B can be raised in partition A, propagated through partition B, and handled back in A.

CXE4002

Check that parameter passing to remote procedures is handled properly when the size of the parameters can be determined at compile time. Check that the following types can be passed as parameters: integer, float, static sized arrays, and simple records. Check the parameter passing using all three modes and check that function results of the various types are handled properly. Check that both direct subprogram calls and indirect calls through a value of a remote access to subprogram can be used for the call.

CXE4003

Check that the task executing a remote subprogram call blocks until the subprogram in the called partition returns. Check that a remote procedure call can be aborted. Check that remote subprogram calls are executed at most once. Check that potentially concurrent calls from multiple tasks can be handled by the PCS.

CXE4004

Check that parameter passing to remote procedures is handled properly when the parameters are of a dynamic size or have discriminants. Check that the following types can be passed as parameters: dynamic sized arrays, constrained discriminated records, unconstrained discriminated records, and tagged records. Check the parameter passing using all three modes and check that function results of the various types are handled properly. Check that both direct subprogram calls and indirect calls through a value of a remote access to subprogram can be used for the call.

CXE4005

Check that calls can be made to remote procedures when a dispatching call is made to a remote access to class wide type. (5) Check that Program_Error is raised if the tag of the actual parameter identifies a tagged type declared in a normal package or in the body of a remote call interface package. (18) Check that in a dispatching call with two controlling operands, Constraint_Error is raised if the two remote access-to-class-wide values originated from Access attribute_references in different partitions. (19)

CXE4006

Check that calls can be made to remote procedures when a dispatching call is made where the controlling operand designates a type declared in a remote call interface package. Check that tagged types can be passed between partitions when passed as a class-wide type. In a remote subprogram call with a formal parameter of a class-wide type, check that Program_Error is raised if the actual parameter identifies a tagged type declared in a normal package.

CXE5001

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

CXE5002

Check that the Partition Communication Subsystem is used for handling remote calls. Check that pragma Asynchronous causes procedure Do_APC to be called and that all other calls go through Do_RPC. Check that pragma All_Calls_Remote is honored by making a call to an RCI unit in the same partition.

CXE5003

Check that System.RPC.Establish_RPC_Receiver is called once after elaborating the library units of a partition and prior to invoking the

main procedure for the partition.

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.

CXF2002

Check that the multiplying operators for a decimal fixed point type return values that are integral multiples of the small of the type. Check the case where the operand and result types are the same. Check that if the mathematical result is between multiples of the small of the result type, the result is truncated toward zero. Check that if the attribute 'Round is applied to the mathematical result, however, the result is rounded to the nearest multiple of the small (away from zero if the result is midway between two multiples of the small).

CXF2003

Check that the multiplying operators for a decimal fixed point type return values that are integral multiples of the small of the type. Check the case where the two operands are of different decimal fixed point types. Check that if the mathematical result is between multiples of the small of the result type, the result is truncated toward zero. Check that if the attribute 'Round is applied to the mathematical result, however, the result is rounded to the nearest multiple of the small (away from zero if the result is midway between two multiples of the small).

CXF2004

Check that the multiplying operators for a decimal fixed point type return values that are integral multiples of the small of the type. Check the case where one operand is of an ordinary fixed point type. Check that if the mathematical result is between multiples of the small of the result type, the result is truncated toward zero. Check that if the attribute 'Round is applied to the mathematical result, however, the result is rounded to the nearest multiple of the small (away from zero if the result is midway between two multiples of the small).

CXF2005

Check that the multiplying operators for a decimal fixed point type return values that are integral multiples of the small of the type. Check the case where one operand is of the predefined type Integer.

CXF2A01

Check that the binary adding operators for a decimal fixed point type return values that are integral multiples of the small of the type.

CXF2A02

Check that the multiplying operators for a decimal fixed point type return values that are integral multiples of the small of the type. Check the case where the operand and result types are the same. Check that if the mathematical result is between multiples of the small of the result type, the result is truncated toward zero.

CXF3001

Check that the edited output string value returned by Function Image is correct.

CXF3002

Check that the functionality contained in package Ada.Wide_Text_IO.Editing is available and produces correct results.

CXF3003

Check that statically identifiable picture strings can be used to produce correctly formatted edited output.

CXF3004

Check that statically identifiable picture strings can be used in conjunction with function Image to produce output strings appropriate to foreign currency representations. Check that statically identifiable picture strings will cause function Image to raise Layout_Error under the appropriate conditions.

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.

CXF3A04

Check that the edited output string value returned by Function Image is correct.

CXF3A05

Check that Function Image produces correct results when provided non-default parameters for Currency, Fill, Separator, and Radix_Mark at either the time of package Decimal_Output instantiation, or in a call to Image. Check non-default parameters that are appropriate for foreign currency representations.

CXF3A06

Check that Ada.Text_IO.Editing.Put and Ada.Text_IO.Put have the same effect.

CXF3A07

Check that Ada.Text_IO.Editing.Put and Ada.Strings.Fixed.Move have the same effect in putting edited output results into string variables.

CXF3A08

Check that the version of Ada.Text_IO.Editing.Put with an out String parameter propagates Layout_Error if the edited output string result of Put exceeds the length of the out String parameter.

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.

CXG2001

Check that the floating point attributes Model_Mantissa, Machine_Mantissa, Machine_Radix, and Machine_Rounds are properly reported.

CXG2002

Check that the complex "abs" or modulus function returns results that are within the error bound allowed.

CXG2003

Check that the sqrt function returns results that are within the error bound allowed.

CXG2004

Check that the sin and cos functions return results that are within the error bound allowed.

CXG2005

Check that floating point addition and multiplication have the required accuracy.

CXG2006

Check that the complex Argument function returns results that are within the error bound allowed. Check that Argument_Error is raised if the Cycle parameter is less than or equal to zero.

CXG2007

Check that the complex Compose_From_Polar function returns results that are within the error bound allowed. Check that Argument_Error is raised if the Cycle parameter is less than or equal to zero.

CXG2008

Check that the complex multiplication and division operations return results that are within the allowed error bound. Check that all the required pure Numerics packages are pure.

CXG2009

Check that the real sqrt and complex modulus functions return results that are within the allowed error bound.

CXG2010

Check that the exp function returns results that are within the error bound allowed.

CXG2011

Check that the log function returns results that are within the error bound allowed.

CXG2012

Check that the exponentiation operator returns results that are within the error bound allowed.

CXG2013

Check that the TAN and COT functions return results that are within the error bound allowed.

CXG2014

Check that the SINH and COSH functions return results that are within the error bound allowed.

CXG2015

Check that the ARCSIN and ARCCOS functions return results that are within the error bound allowed.

CXG2016

Check that the ARCTAN function returns a result that is within the error bound allowed.

CXG2017

Check that the TANH function returns a result that is within the error bound allowed.

CXG2018

Check that the complex EXP function returns a result that is within the error bound allowed.

CXG2019

Check that the complex LOG function returns a result that is within the error bound allowed.

CXG2020

Check that the complex SQRT function returns a result that is within the error bound allowed.

CXG2021

Check that the complex SIN and COS functions return a result that is

within the error bound allowed.

CXG2022

Check that multiplication and division of binary fixed point numbers with compatible 'small values produce exact results.

CXG2023

CXG2024

Check that multiplication and division of decimal and binary fixed point numbers that result in a decimal fixed point type produce acceptable results.

CXH1001

Check pragma Normalize_Scalars. Check that this configuration pragma causes uninitialized scalar objects to be set to a predictable value. Check that multiple compilation units are affected. Check for uninitialized scalar objects that are subcomponents of composite objects, unassigned out parameters, objects that have been allocated without an initial value, and objects that are stand alone.

CXH3001

Check pragma Reviewable. Check that pragma Reviewable is accepted as a configuration pragma.

CXH3002

Check that pragma Inspection_Point is allowed whereever a declarative item or statement is allowed. Check that pragma Inspection_Point may have zero or more arguments. Check that the execution of pragma Inspection_Point has no effect.

CXH30030

See CHX30031.AM

CXH30031

Check pragma Reviewable. Check that pragma Reviewable is accepted as a configuration pragma.

F954A00

This file contains foundation code for tests covering the requeue statement.

LA140010

See LA140011.AM.

LA140011

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level function body depends on a unit that is changed.

LA140012

See LA140011.AM.

LA140020

See LA140021.AM.

LA140021

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a unit depends on a package whose declaration is changed.

LA140022

See LA140021.AM.

LA140030

See LA140032.AM.

LA140031

See LA140032.AM.

LA140032

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a package body depends on a package specification that is changed.

LA140033

See LA140032.AM.

LA140040

See LA140041.AM.

LA140041

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic function depends on a library level package.

LA140042

See LA140041.AM.

LA140050

See LA140052.AM.

LA140051

See LA140052.AM.

LA140052

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic package body depends on a generic package specification.

LA140053

See LA140052.AM.

LA140060

See LA140062.AM.

LA140061

See LA140062.AM.

LA140062

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic package depends on another generic package specification.

LA140063

See LA140062.AM.

LA140070

See LA140072.AM.

LA140071

See LA140072.AM.

LA140072

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a separate procedure body depends on a non-generic package specification that is changed.

LA140073

See LA140072.AM.

LA140080

See LA140082.AM.

LA140081

See LA140082.AM.

LA140082

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a subunit function body depends on a unit that is changed.

LA140083

See LA140082.AM.

LA140090

See LA140092.AM.

LA140091

See LA140092.AM.

LA140092

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a subunit package body depends on a unit that is changed.

LA140093

See LA140092.AM.

LA140100

See LA140102.AM.

LA140101

See LA140102.AM.

LA140102

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a task body depends on a package specification.

LA140103

See LA140102.AM.

LA140110

See LA140112.AM.

LA140111

See LA140112.AM.

LA140112

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library procedure depends on a unit that is changed.

LA140113

See LA140112.AM.

LA140120

See LA140122.AM.

LA140121

See LA140122.AM.

LA140122

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level function depends on a unit that is changed.

LA140123

See LA140122.AM.

LA140130

See LA140132.AM.

LA140131

See LA140132.AM.

LA140132

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level package depends on a package specification that is changed.

LA140133

See LA140132.AM.

LA140140

See LA140142.AM.

LA140141

See LA140142.AM.

LA140142

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level procedure depends on another library level procedure that is changed.

LA140143

See LA140142.AM.

LA140150

See LA140152.AM.

LA140151

See LA140152.AM.

LA140152

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level function depends on another library level function that is changed.

LA140153

See LA140152.AM.

LA140160

See LA140162.AM.

LA140161

See LA140162.AM.

LA140162

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a separate procedure depends on a withed generic package that is changed.

LA140163

See LA140162.AM.

LA140170

See LA140172.AM.

LA140171

See LA140172.AM.

LA140172

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a separate function semantically depends on a library level generic function that is changed.

LA140173

See LA140172.AM.

LA140180

See LA140182.AM.

LA140181

See LA140182.AM.

LA140182

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a separate generic package body depends on a library level generic package body that is changed.

LA140183

See LA140182.AM.

LA140190

See LA140192.AM.

LA140191

See LA140192.AM.

LA140192

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level generic procedure depends on library level procedure that is changed.

LA140193

See LA140192.AM.

LA140200

See LA140202.AM.

LA140201

See LA140202.AM.

LA140202

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a library level instance depends on a library level generic function whose body is changed.

LA140203

See LA140202.AM.

LA140210

See LA140211.AM.

LA140211

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic package depends on another generic package that is changed.

LA140212

See LA140211.AM.

LA140220

See LA140221.AM.

LA140221

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic instantiation depends on a generic procedure that is changed.

LA140222

See LA140221.AM.

LA140230

See LA140232.AM.

LA140231

See LA140232.AM.

LA140232

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic instantiation depends on a generic function that is changed.

LA140233

See LA140232.AM.

LA140240

See LA140242.AM.

LA140241

See LA140242.AM.

LA140242

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic instantiation depends on a generic package that is changed.

LA140243

See LA140242.AM.

LA140250

See LA140251.AM.

LA140251

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic instantiation depends on a non-generic package that is changed.

LA140252

See LA140251.AM.

LA140260

See LA140262.AM.

LA140261

See LA140262.AM.

LA140262

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a generic instantiation depends on a generic package instantiation that is changed.

LA140263

See LA140262.AM.

LA140270

See LA140272.AM.

LA140271

See LA140272.AM.

LA140272

Check that a compilation unit may not depend semantically on two different versions of the same compilation unit. Check the case where a task body depends on non-generic package specification.

LA140273

See LA140272.AM.

LXD70010

See LXD70012.AM.

LXD70011

See LXD70012.AM.

LXD70012

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

LXD70030

See LXD70032.AM.

LXD70031

See LXD70032.AM.

LXD70032

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.

LXD70040

See LXD70042.AM.

LXD70041

See LXD70042.AM.

LXD70042

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

LXD70050

See LXD70052.AM.

LXD70051

See LXD70052.AM.

LXD70052

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

LXD70060

See LXD70062.AM.

LXD70061

See LXD70062.AM.

LXD70062

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

LXD70070

See LXD70072.AM

LXD70071

See LXD70072.AM

LXD70072

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

LXD70080

See LXD70082.AM.

LXD70081

See LXD70082.AM.

LXD70082

Check that a partition obeys the restriction if a configuration pragma Restrictions (No_Asynchronous_Control) is included

LXD70090

See LXD70092.AM.

LXD70091

See LXD70092.AM.

LXD70092

Check that a partition obeys the restriction if the following configuration restrictions are included: pragma Restrictions (Max_Select_Alternatives => 0) pragma Restrictions (Max_Task_Entries => 0) pragma Restrictions (Max_Protected_Entries => 0)

LXE30010

This test checks that an inconsistent distributed program is properly detected.

LXE30011

See LXE30010.AM

LXE30020

Check that an inconsistent distributed program is properly detected.

LXE30021

See LXE30020.AM.

LXH40010

See file LXH40012.AM for test objective.

LXH40011

See file LXH40012.AM for objective.

LXH40012

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Protected_Types disallows protected types in the units previously compiled into the program library.

LXH40020

See file LXH40022.AM for details on this test

LXH40021

See file LXH40022.AM for details on this test

LXH40022

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Allocators disallows allocators in the units previously compiled into the program library.

LXH40030

See file LXH40033.AM for details on this test

LXH40031

See file LXH40033.AM for details on this test

LXH40032

See file LXH40033.AM for details on this test

LXH40033

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Local_Allocators disallows local allocators in the units previously compiled into the program library.

LXH40040

See file LXH40043.AM for details on this test

LXH40041

See file LXH40041.AM for details on this test

LXH40042

See file LXH40043.AM for details on this test

LXH40043

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Unchecked_Deallocation disallows the use of Unchecked_Deallocation in the units previously compiled into the program library.

LXH40050

See file LXH40053.AM for details on this test

LXH40051

See file LXH40053.AM for details on this test

LXH40052

See file LXH40053.AM for details on this test

LXH40053

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of

the configuration pragma Restrictions with the specific restriction: No_Exceptions disallows exceptions in the units previously compiled into the program library.

LXH40060

See file LXH40063.AM for details on this test

LXH40061

See file LXH40063.AM for details on this test

LXH40062

See file LXH40063.AM for details on this test

LXH40063

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Floating_Point disallows the use of floating point in the units previously compiled into the program library.

LXH40070

See file LXH40073.AM for details on this test

LXH40071

See file LXH40073.AM for details on this test

LXH40072

See file LXH40073.AM for details on this test

LXH40073

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Fixed_Point disallows the use of fixed point in the units previously compiled into the program library.

LXH40080

See file LXH40084.AM for details on this test

LXH40081

See file LXH40084.AM for details on this test

LXH40082

See file LXH40084.AM for details on this test

LXH40083

See file LXH40084.AM for details on this test

LXH40084

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Unchecked_Conversion disallows the use of Unchecked_Conversion in the units previously compiled into the program library.

LXH40090

See file LXH40093.AM for details on this test

LXH40091

See file LXH40093.AM for details on this test

LXH40092

See file LXH40093.AM for details on this test

LXH40093

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Access_Subprograms disallows access to subprograms in the units previously compiled into the program library.

LXH40100

See file LXH40103.AM for details on this test

LXH40101

See file LXH40103.AM for details on this test

LXH40102

See file LXH40103.AM for details on this test

LXH40103

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Unchecked_Access disallows the use of the Unchecked_Access attribute in the units previously compiled into the program library.

LXH40110

See file LXH40112.AM for details on this test

LXH40111

See file LXH40112.AM for details on this test

LXH40112

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_Dispatch disallows T'Class in the units previously compiled into the program library.

LXH40120

See file LXH40123.AM for details on this test

LXH40121

See file LXH40123.AM for details on this test

LXH40122

See file LXH40123.AM for details on this test

LXH40123

Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition. Check that the application of the configuration pragma Restrictions with the specific restriction: No_IO disallows I/O in the units previously compiled into the program library.

LXH40130

See file LXH40133.AM for details on this test

LXH40131

See file LXH40133.AM for details on this test

LXH40132

See file LXH40133.AM for details on this test

LXH40133
Check that pragma Restrictions (using the restrictions defined in Annex H) applies to all units in a partition.
Check that the application of the configuration pragma Restrictions with the specific restriction: No_Delay
disallows delay statements in the units previously compiled into the program library.

LXH40140

See file LXH40142.AM for details on this test

LXH40141

See file LXH40142.AM for details on this test

LXH40142

Check that pragma Restrictions (using the restrictions defined
in Annex H) applies to all units in a partition.
Check that the application of the configuration pragma Restrictions
with the specific restriction: No_Dispatch
disallows T'Class in units compiled after the configuration pragma.