IEEE 200X Fast Track Change Proposal

ID: FT-14 and FT-15

Proposer: Ryan Hinton
Email: ryan.w.hinton@L-3com.com

Status: Proposed
Proposed: 12/03
Analyzed: Date
Resolved: Date

Enhancement Summary:
Arrays of unconstrained arrays and records with unconstrained arrays
Related issues:
Relevant LRM section: section 3.2.1.1, section 3.2.2

Enhancement Detail:
Allow composite types with unconstrained elements. This entails expanding the definition of a constraint and often including partially constrained array or record types/subtypes with unconstrained array types/subtypes. It also requires more power in a subtype indication to specify several constraints applied to several different types.

Analysis

Create notions of a “partially constrained” and “fully constrained” composite type. Eventually, every object must be fully constrained, but partially constrained subtypes in interface lists can improve code reuse. Add a production for “aggregate_index_constraint” to constrain an array element subtype and optionally the array subtype. Add productions for “record_constraint” and “record_element_constraint” to constrain partially constrained element subtypes. Extend index range resolution rules to handle all partially constrained composite types. Extend mention of unconstrained array types to include partially constrained composite types where necessary. Extend notion of subtypes to record types so explicit and implicit (constrained) record subtypes are possible, and allow implicit subtype conversions on records where necessary. Also extend the record type definition to always create a record type and to also create a record subtype when the element subtype indications include constraints.

Constraints are extended so an index constraint “applies” to a given unconstrained array type and a record constraint “applies” to a given record type. The (potentially recursive) element subtype constraint given in an aggregate index constraint “applies” to the element subtype, although it may not impose an index constraint on the subtype. This is accomplished using the “(open)” sequence. Similarly, the record element constraint

"applies" to the subtype of the indicated element, although the provided constraint may not impose a constraint on the subtype.

Files of unconstrained, one-dimensional arrays whose element subtype is fully constrained retain implicitly defined READ and WRITE subprograms. Files of composite subtypes whose elements are not fully constrained (i.e. all other partially constrained subtypes) do not have implicitly defined READ and WRITE subprograms.

To access attributes of the element type of an array, an attribute A'ELEMENT (for parallelism to A'BASE, or A'ELEMENT_TYPE to be more explicit) is added that returns the constrained element type. The attributes of elements of a record type are accessible by specifying the field name in parenthesis, e.g. R'ELEMENT(fieldname)'RANGE.

LRM Changes

- The following text in section 1.1.1.2 needs to be changed.

"If a formal port is associated with an actual port, signal, or expression, then the formal port is said to be *connected*. If a formal port is instead associated with the reserved word **open**, then the formal is said to be *unconnected*. It is an error if a port of mode **in** is unconnected or unassociated (see 4.3.2.2) unless its declaration includes a default expression (see 4.3.2). It is an error if a port of any mode other than **in** is unconnected or unassociated and its type is ~~an unconstrained array~~ partially constrained composite type. It is an error if some of the subelements of a composite formal port are connected and others are either unconnected or unassociated.

- The following text in section 2.1.1.1 needs to be changed.

"For a formal parameter of a constrained array subtype of mode **in** or **inout**, it is an error if the value of the associated actual parameter (after application of any conversion function or type conversion present in the actual part) does not contain a matching element for each element of the formal. For a formal parameter whose declaration contains a subtype indication denoting ~~an unconstrained array~~ a partially constrained composite type, the subtype of the formal in any call to the subprogram is taken from the actual associated with that formal in the call to the subprogram. It is also an error if, in either case, the value of each element of the actual ~~array~~ object (after applying any conversion function or type conversion present in the actual part) does not belong to the element subtype of the formal. If the formal parameter is of mode **out** or **inout**, it is also an error if, at the end of the subprogram call, the value of each element of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the element subtype of the actual."

- The following rule needs to be added to the rules in section 3.2.1.

"
optional_index_constraint ::= index_constraint | (open)
augmented_index_constraint ::= optional_index_constraint constraint"

- Section 3.2.1 should include an example of an array of unconstrained arrays. Here are a few possibilities.

"**type** SIGNED_FXPT **is array** (INTEGER **range <>**) **of** BIT;
type SIGNED_FXPT_VECTOR **is array** (NATURAL **range <>**) **of** SIGNED_FXPT;
type SIGNED_FXPT_5x4 **is array** (1 **to** 5, 1 **to** 4) **of** SIGNED_FXPT;

signal VEC : SIGNED_FXPT_VECTOR(1 **to** 20)(9 **downto** 0);
variable SMATRIX : SIGNED_FXPT_5x4(**open**)(11 **downto** 0);"

- Section 3.2.1.1 should be changed to add augmented index constraints.

"**3.2.1.1 Augmented index constraints, index constraints and discrete ranges**

An index constraint determines the index range for every index of ~~an~~ the given array type and, thereby, the corresponding array bounds. An index constraint *applies* to the given array type or the designated type of the given access type.

An augmented index constraint may be used when the given array element subtype is not fully constrained. An augmented index constraint provides a constraint for the given element subtype and optionally provides an index constraint for the given array type. The element subtype constraint applies to the given element subtype. If provided, the optional index constraint imposes a constraint on the given array type or the designated type of the given access type. If an augmented index constraint is used to constrain the element subtype of a constrained array subtype, the augmented constraint must not impose an index constraint on the constrained array subtype by using the OPEN keyword.

An array subtype is *fully constrained* if the array subtype is constrained and its element subtype is fully constrained. An array subtype is *partially constrained* if it is not fully constrained.

For a discrete range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if each bound is either a numeric literal or an attribute, and if the type of both bounds (prior to the implicit conversion) is the type *universal_integer*. Otherwise, both bounds must be of the same discrete type, other than *universal_integer*; this type must be determined independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration scheme (see 8.9) or a generation scheme (see 9.7).

If an index constraint appears ~~after a type mark~~ in a subtype indication, then the applicable type or subtype ~~denoted by the type mark~~ *must* not already impose an index constraint. The applicable type ~~mark~~ *must* ~~denote either~~ be an unconstrained array ~~type or an access type whose designated type is such an array type~~. ~~In either case~~ Also, the index constraint *must* provide a discrete range for each index of the array type, and the type of each discrete range *must* be the same as that of the corresponding index."

- The bulleted section in section 3.2.1.1 needs to be changed.

"The index range for each index of an array object and each index of the subelements of an array or record object is determined as follows:

- For a variable or signal declared by an object declaration, the subtype indication of the corresponding object declaration must define a fully constrained ~~array subtype~~ composite subtype (and thereby, the index range for each index of the object and its subelements). ~~The same requirement exists for the subtype indication of an element declaration, if the type of the record element is an array type, and for the element subtype indication of an array type definition, if the type of the array element is itself an array type.~~
- For a constant declared by an object declaration, the index ranges are defined by the initial value, if the subtype of the constant is ~~unconstrained~~ not fully constrained; otherwise, they are defined by this subtype (in which case the initial value is the result of an implicit subtype conversion).
- For an attribute whose value is specified by an attribute specification, the index ranges are defined by the expression given in the specification, if the subtype of the attribute is ~~unconstrained~~ not fully constrained; otherwise, they are defined by this subtype (in which case the value of the attribute is the result of an implicit subtype conversion).
- For an array object designated by an access value, the index ranges are defined by the allocator that creates the array object (see 7.3.6).
- For an interface object declared with a subtype indication that defines a constrained array subtype, the index ranges are defined by that subtype or subnature.
- For a formal parameter or a subelement of a formal parameter of a subprogram that is of an unconstrained array type and that is associated in whole (see 4.3.2.2), the index ranges are obtained from the corresponding association element in the applicable subprogram call.

- For a formal parameter or a subelement of a formal parameter of a subprogram that is of an unconstrained array type and whose subelements are associated individually (see 4.3.2.2), the index ranges are obtained as follows: The directions of the index ranges of the formal parameter are that of the type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.
- For a formal generic or a formal port of a design entity or of a block statement or a subelement of one of these that is of an unconstrained array type and that is associated in whole, the index ranges are obtained from the corresponding association element in the generic map aspect (in the case of a formal generic) or port map aspect (in the case of a formal port) of the applicable (implicit or explicit) binding indication.
- For a formal generic or a formal port of a design entity or of a block statement or a subelement of one of these that is of an unconstrained array type and whose subelements are associated individually, the index ranges are obtained as follows: The directions of the index ranges of the formal generic or formal port are that of the type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.
- For a local generic or a local port of a component or a subelement of one of these that is of an unconstrained array type and that is associated in whole, the index ranges are obtained from the corresponding association element in the generic map aspect (in the case of a local generic) or port map aspect (in the case of a local port) of the applicable component instantiation statement.
- For a local generic or a local port of a component or a subelement of one of these that is of an unconstrained array type and whose subelements are associated individually, the index ranges are obtained as follows: The directions of the index ranges of the local generic or local port are that of the type of the local; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the local.

If the index ranges for an interface object or member of an interface object or a subelement of one of these are obtained from the corresponding association element (when associating in whole) or elements (when associating individually), then they are determined either by the actual part(s) or by the formal part(s) of the association element(s), depending upon the mode of the interface object, as follows:

- For an interface object or member of an interface object whose mode is **in**, **inout**, or **linkage**, if the actual part includes a conversion function or a type conversion, then the result type of that function or the type mark of the type conversion must be a fully constrained ~~array~~ composite subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object or value denoted by the actual designator(s).
- For an interface object or member of an interface object whose mode is **out**, **buffer**, **inout**, or **linkage**, if the formal part includes a conversion function or a type conversion, then the parameter subtype of that function or the type mark of the type conversion must be a fully constrained ~~array~~ composite subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object denoted by the actual designator(s).”

- A refined version of the example in this proposal may be added to the end of section 3.2.1.1 for clarification. Here is one unspectacular possibility. It is probably not necessary, though, since the only new thing is how constraints are specified, and an example of that is suggested in this document for section 3.2.1.

“
```
type SIGNED_FXPT is array (INTEGER range <>) of BIT;
type SIGNED_FXPT_VECTOR is array (NATURAL range <>) of SIGNED_FXPT;
type SIGNED_FXPT_5x4 is array (1 to 5, 1 to 4) of SIGNED_FXPT;

entity MATRIX_PRODUCT is
        port (    Matrix : in SIGNED_FXPT_5x4;
                  Vector : in SIGNED_FXPT_VECTOR(1 to 4);
                  Product : out SIGNED_FXPT_VECTOR(1 to 5));
```

```
end entity VECTOR_SUM;

signal SMatrix : SIGNED_FXPT_5x4(open)(11 downto 0);
signal SProd : SIGNED_FXPT_VECTOR(0 to 4)(15 downto 0);

MP_Instance : entity MATRIX_PRODUCT
        port map (
                Matrix => SMatrix,
                Vector => (others => (9 downto 0 => '1')), -- check my syntax here
                Procut => SProd) "
```

- The following rules need to be added to the rules in section 3.2.2.

"
record_element_constraint ::= identifier (constraint)
record_constraint ::= (record_element_constraint {, record_element_constraint })"

- Section 3.2.2 needs to be extended to define record subtypes.

"A record type definition that includes any constraints in its element subtype definitions creates both a record type and a subtype of this type:~~; it consists of the element declarations in the order in which they appear in the type definition.~~

- The record type is an implicitly declared anonymous type; this type is defined by an (implicit) record definition without any constraints in its element subtype definitions.
- The record subtype is the subtype obtained by imposition of any constraints in its element subtype definitions.

If a record type definition includes any constraints in its element subtype definitions, the simple name declared in the record definition denotes the record subtype. Otherwise, the record type definition defines a record type and the simple name in that definition denotes this type."

- A section 3.2.2.1 should be added as follows.

"**3.2.2.1 Record constraints**

A record constraint may be used in a subtype indication when elements in the record subtype are not fully constrained. A record constraint consists of one or more record element constraints. A record element constraint provides a constraint for the subtype of the element indicated by the given identifier. A record constraint *applies* to the given record subtype or the designated subtype of the given access type. A record element constraint *applies* to the subtype of the indicated element. A record constraint does not impose constraints on the subtypes of any elements for which record element constraints are not given.

A record type is *fully constrained* if the subtype indication (after applying any constraints) of each of its elements denotes a fully constrained type. A record type is *partially constrained* if it is not fully constrained.

A record constraint is *compatible* with the type or subtype denoted by the type mark if, and only if, each record element constraint corresponds to a valid element name and the constraint given for that element is compatible with the subtype of that element. A record value *satisfies* a record constraint if for each element the value for that element satisfies the constraint for that element.

It is an error if a record constraint applies to a record type or subtype that is fully constrained or if an element constraint is applied to an element whose subtype is fully constrained. However, a record constraint need not provide a constraint for each element of the record whose subtype is not fully constrained. Instead, incremental constraints are allowed to successively constrain a record subtype.

The index range for each index of each subelement of a record object is determined by the rules given in section 3.2.1.1.

Examples:

```
type SIGNED_FXPT is array (INTEGER range <>) of BIT;
type COMPLEX is
        record
                re : SIGNED_FXPT;
                im : SIGNED_FXPT;
        end record;
type COMPLEX_2DIM is array (NATURAL range <>, NATURAL range <>) of COMPLEX;

entity RANK is
        generic (WIDTH : POSITIVE);
        port (  Matrix : in COMPLEX_2DIM(open)(re(WIDTH-1 downto 0), im(WIDTH-1 downto
0))
                Matrix_rank : out UNSIGNED);"
```

- The following text in section 3.3 needs to be changed.

"The only forms of constraint that ~~is~~ are allowed after the name of an access type in a subtype indication ~~is~~ are an index constraint, an augmented index constraint, and a record constraint. An access value belongs to a corresponding subtype of an access type either if the access value is the null value or if the value of the designated object satisfies the constraint."

- Implicit file operation definitions need to be properly restricted with the following modifications to section 3.4.1.

"where type mark TM denotes a scalar type, a fully constrained record subtype, or a fully constrained array subtype, the following operations are implicitly declared immediately following the file type declaration:"

"For a file type declaration in which the type mark denotes an unconstrained array type whose element subtype is fully constrained, the same operations are implicitly declared, except that the READ operation is declared as follows:

procedure READ (**file** F: FT; VALUE: **out** TM; LENGTH: **out** Natural);

The READ operation for such a type performs the same function as the READ operation for other types, but in addition it returns a value in parameter LENGTH that specifies the actual length of the array value read by the operation. If the object associated with formal parameter VALUE is shorter than this length, then only that portion of the array value read by the operation that can be contained in the object is returned by the READ operation, and the rest of the value is lost. If the object associated with formal parameter VALUE is longer than this length, then the entire value is returned and remaining elements of the object are unaffected by the READ operation.

For a file type declaration in which the type mark denotes a partially constrained composite subtype (other than an unconstrained array whose element subtype is fully constrained), no READ or WRITE operations are implicitly declared."

- The production rule in section 4.2 needs to be changed.

" constraint ::=
        range_constraint
      | index_constraint
      | augmented_index_constraint
      | record_constraint"

- The following text in section 4.2 needs to be changed.

" A subtype indication denoting an access type, a file type, or a protected type may not contain a resolution function. Furthermore, the only allowable constraints on a subtype indication denoting an access type ~~is~~ are an index constraint, an augmented index constraint, and a record constraint (and then only if the designated

type is an unconstrained array type, a partially constrained array type, or partially constrained record type, respectively).

A subtype indication denoting a subtype of ~~a record type,~~ a file type~~,~~ or a protected type may not contain a constraint."

- Object aliases need to be refined with the following changes to section 4.3.3.1.

"The name must be a static name (see 6.1) that denotes an object. The base type of the name specified in an alias declaration must be the same as the base type of the type mark in the subtype indication (if the subtype indication is present); this type must not be a multidimensional array type. When the object denoted by the name is referenced via the alias defined by the alias declaration, the following rules apply:

1) If the subtype indication is absent or if it is present and denotes ~~an unconstrained array~~ a partially constrained composite type
   - If the alias designator denotes a slice of an object, then the subtype of the object is viewed as if it were of the subtype specified by the slice.
   - Otherwise, the object is viewed as if it were of the subtype specified in the declaration of the object denoted by the name.
2) If the subtype indication is present and denotes a fully constrained ~~array~~ composite subtype, then the object is viewed as if it were of the subtype specified by the subtype indication; moreover, the subtype denoted by the subtype indication must include a matching element (see 7.2.2) for each element of the object denoted by the name.
3) If the subtype indication denotes a scalar subtype, then the object is viewed as if it were of the subtype specified by the subtype indication; moreover, it is an error if this subtype does not have the same bounds and direction as the subtype denoted by the object name."

- Matching elements need to be extended to matching subelements with the following changes to section 7.2.2.

"Two scalar values of the same type are equal if and only if the values are the same. Two composite values of the same type are equal if and only if for each subelement of the left operand there is a *matching subelement* of the right operand and vice versa, and the values of matching subelements are equal, as given by the predefined equality operator for the subelement type. In particular, two null arrays of the same type are always equal. Two values of an access type are equal if and only if they both designate the same object or they both are equal to the null value for the access type.

For two record values, matching elements are those that have the same element identifier. For two one-dimensional array values, matching elements are those (if any) whose index values match in the following sense: the left bounds of the index ranges are defined to match; if two elements match, the elements immediately to their right are also defined to match. For two multidimensional array values, matching elements are those whose indices match in successive positions. Composite element values of record and array types match only if the element values match using these same rules."

- Subtype conversions for record subtypes need to be added with the following change to section 7.3.5.

"If the type mark denotes a subtype, conversion consists of conversion to the target type followed by a check that the result of the conversion belongs to the subtype. If the type mark denotes a record subtype, conversion consists of a check that the expression belongs to the subtype."

- Type conversions need to be extended to partially constrained types with a change to section 7.3.5 similar to the following.

"In the case of conversions between array types, a check is made that any constraints on the subelement subtypes ~~is~~ are the same for the operand array type as for the target array type. If the type mark denotes an unconstrained array type, then, for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type. If the type mark denotes a constrained array subtype, a check is made that for each element of the operand there is a matching element of the target subtype, and vice versa. It is an error if any of these checks fail."

- The following text in section 7.3.6 needs to be changed.

"The only allowed forms of constraint in the subtype indication of an allocator ~~is~~ are an index constraint, an augmented index constraint, and a record constraint. If an allocator includes a subtype indication and if the type of the object created is a~~n array~~ composite type, then the subtype indication must either denote a fully constrained subtype or include an explicit ~~index~~ constraint such that the defined subtype is fully constrained. A subtype indication that is part of an allocator must not include a resolution function."

- An example showing more types of constraints could be included after section 7.3.6.

- The following text needs to be added to section 7.4.1.

"A locally static index constraint is an index constraint for which each index subtype of the corresponding array type is locally static and in which each discrete range is locally static. A locally static augmented constraint is a augmented constraint for which the index constraint, if provided, is locally static, and the constraint applied to the element subtype is locally static. A locally static record constraint is a record constraint for which each record element constraint is locally static. A locally static record element constraint is a record element constraint for which the constraint applied to the element subtype is locally static. A locally static array subtype is a constrained array subtype formed by imposing on an unconstrained array type a locally static index constraint. A locally static record subtype is a record type whose fields are all of locally static subtypes. A locally static access subtype is a subtype denoting an access type. A locally static file subtype is a subtype denoting a file type."

- The following text needs to be added to section 7.4.2.

"A globally static index constraint is an index constraint for which each index subtype of the corresponding array type is globally static and in which each discrete range is globally static. A globally static augmented constraint is a augmented constraint for which the index constraint, if provided, is globally static, and the constraint applied to the element subtype is globally static. A globally static record constraint is a record constraint for which each record element constraint is globally static. A globally static record element constraint is a record element constraint for which the constraint applied to the element subtype is globally static. A globally static array subtype is a constrained array subtype formed by imposing on an unconstrained array type a globally static index constraint. A globally static record subtype is a record type whose fields are all of globally static subtypes. A globally static access subtype is a subtype denoting an access type. A globally static file subtype is a subtype denoting a file type."

- The following text should probably be added to the notes following section 7.4.2.

"1—An expression that is required to be a static expression may either be a locally static expression or a globally static expression. Similarly, a range, a range constraint, a scalar subtype, a discrete range, an index constraint, an augmented index constraint, a record constraint, a record element constraint, or an array subtype that is required to be static may either be locally static or globally static."

- Variable assignment implicit subtype conversions need to be extended to record subtypes with the following modifications to section 8.5.

"For the execution of a variable assignment whose target is a variable name, the variable name and the expression are first evaluated. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is ~~an array~~ a composite subtype (in which case the assignment involves a subtype conversion). Finally, the value of the expression becomes the new value of the variable. A design is erroneous if it depends on the order of evaluation of the target and source expressions of an assignment statement."

- The following text in section 8.5.1 also needs to be augmented.

"8.5.1 ~~Array~~ Composite variable assignments

If the target of an assignment statement is a name denoting ~~an array~~ a composite variable (including an array slice), the value assigned to the target is implicitly converted to the subtype of the ~~array~~ composite variable; the result of this subtype conversion becomes the new value of the ~~array~~ composite variable.

This means that the new value of each subelement of the ~~array~~ composite variable is specified by the matching subelement (see 7.2.2) in the corresponding ~~array~~ composite value obtained by evaluation of the expression (including an implicit subtype conversion if the subelement subtype is a composite subtype). The subtype conversion checks that for each subelement of the ~~array~~ composite variable there is a matching subelement in the ~~array~~ composite value, and vice versa. An error occurs if this check fails.

~~NOTE—The implicit subtype conversion described for assignment to an array variable is performed only for the value of the right-hand side expression as a whole; it is not performed for subelements or slices that are array values.~~"

- The second note to section 8.12 should be updated for record subtypes.

"2—If the return type mark of a function denotes a ~~constrained array~~ composite subtype, then no implicit subtype conversions are performed on the values of the expressions of the return statements within the subprogram body of that function. Thus, for each index position of each subelement array value, the bounds of the discrete range must be the same as the discrete range of the return subtype, and the directions must be the same."

- Elaboration of generics needs to be extended with the following changes to section 12.2.2.

"Elaboration of a generic association list consists of the elaboration of each generic association element in the association list. Elaboration of a generic association element consists of the elaboration of the formal part and the evaluation of the actual part. The generic constant or subelement or slice thereof designated by the formal part is then initialized with the value resulting from the evaluation of the corresponding actual part. It is an error if the value of the actual does not belong to the subtype denoted by the subtype indication of the formal. If the subtype denoted by the subtype indication of the declaration of the formal is a ~~constrained array~~ composite subtype, then an implicit subtype conversion is performed prior to this check for constrained array subtypes and any constrained subelement subtypes. It is also an error if the type of the formal is an array type and the value of each element of the actual does not belong to the element subtype of the formal."

- The text of section 12.3.1.3 needs to be changed.

"Elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype does not include a constraint, then the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows:

a) The constraint is first elaborated.
b) A check is then made that ~~the~~ each constraint is compatible with the applicable type or subtype ~~denoted by the type mark~~ (see 3.1, ~~and~~ 3.2.1.1, and 3.2.2.1).

Elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines the bounds and direction of the range. Elaboration of an index constraint consists of the elaboration of each of the discrete ranges in the index constraint in some order that is not defined by the language. Elaboration of an augmented index constraint consists of the elaboration (in some order that is not defined by the language) of the index constraint if it is provided and elaboration of the element subtype constraint given the element subtype. Elaboration of a record constraint consists of the elaboration of each record element constraint in the record constraint in some order that is not defined by the language. Elaboration of a record element constraint consists of elaboration of the constraint given the type or subtype of the indicated element."

- Elaboration of object declarations needs to be extended for record subtypes and partially constrained subtypes with the following modification to section 12.3.1.3.

b) "If the object declaration includes an explicit initialization expression, then the initial value of the object is obtained by evaluating the expression. It is an error if the value of the expression does not belong to the subtype of the object; if the object is ~~an array~~ a composite object, then an implicit subtype conversion is first performed on the value unless the object is a constant whose subtype indication denotes ~~an unconstrained array~~ a partially constrained composite type. For a constant

object whose subtype indication denotes a partially constrained composite type, implicit subtype conversions are performed for any constrained subelement composite types and for the top-level type if it is a constrained composite type. Otherwise, any implicit initial value for the object is determined.

c) The object is created.

d) Any initial value is assigned to the object.

The initialization of such an object (either the declared object or one of its subelements) involves a check that the initial value belongs to the subtype of the object. For ~~an array~~ a composite object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement, unless the object is a constant whose subtype is ~~an unconstrained array~~ a partially constrained composite subtype. If the object is a constant whose subtype is a partially constrained composite subtype, implicit subtype conversions are performed on the expression value if the constant subtype is constrained and on any subelement values whose constant subelement subtypes are constrained. Constraints for unconstrained subelement types are determined by the constraints of the value of the expression."

- Elaboration of attribute specifications needs to be extended in a similar fashion with the following modifications to section 12.3.2.1.

"Elaboration of an attribute specification proceeds as follows:

a) The entity specification is elaborated in order to determine which items are affected by the attribute specification.

b) The expression is evaluated to determine the value of the attribute. It is an error if the value of the expression does not belong to the subtype of the attribute; if the attribute is of ~~an array~~ a composite type, then an implicit subtype conversion is first performed on the value, unless the subtype indication of the attribute denotes ~~an unconstrained array~~ a partially constrained composite type. When the subtype indication of the attribute denotes a partially constrained composite type, implicit subtype conversions are performed for any constrained subelement composite types and for the top-level type if it is a constrained composite type.

c) A new instance of the designated attribute is created and associated with each of the affected items.

d) Each new attribute instance is assigned the value of the expression.

The assignment of a value to an instance of a given attribute involves a check that the value belongs to the subtype of the designated attribute. For an attribute of a ~~constrained array~~ composite type, an implicit subtype conversion is first applied as for an assignment statement unless the attribute is of a partially constrained composite type. ~~No such conversion is necessary for an attribute of an unconstrained array type; the constraints on the value determine the constraints on the attribute.~~ If the attribute is of a partially constrained composite type, implicit subtype conversions are performed on the expression value attribute subtype is constrained composite type and on any subelement values whose attribute subelement subtypes are constrained. Constraints for unconstrained subelement types are determined by the constraints of the value of the expression."

- Implicit subtype conversion for signal assignment is clarified by the following changes to section 12.6.2.

"For a scalar signal S, both the driving and effective values must belong to the subtype of the signal. For a composite signal R, an implicit subtype conversion is performed to the subtype of R; for each subelement of R, there must be a matching subelement in both the driving and the effective value, and vice versa.

In order to update a signal during a given simulation cycle, the kernel process first determines the driving and effective values of that signal. The kernel process then updates the variable containing the current value of the signal with the newly determined effective value, as follows:

a) If S is a signal of some type that is not ~~an array~~ a composite type, the effective value of S is used to update the current value of S. A check is made that the effective value of S belongs to the

subtype of S. An error occurs if this subtype check fails. Finally, the effective value of S is assigned to the variable representing the current value of the signal.

b) If S is ~~an array~~ a composite signal (including a slice of an array), the effective value of S is implicitly converted to the subtype of S. The subtype conversion checks that for each subelement of S there is a matching subelement in the effective value and vice versa. An error occurs if this check fails. The result of this subtype conversion is then assigned to the variable representing the current value of S."

- The ELEMENT attribute for arrays and records needs to be added to section 14.1.

"A'ELEMENT

Kind: Type.
Prefix: Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes an array subtype.
Result: The element type of array A.

R'ELEMENT(X)

Kind: Type.
Prefix: Any prefix R that is appropriate for a record object, or an alias thereof, or denotes a record subtype.
Parameter: An identifier matching an element name of the prefix type.
Result: The type of the matching element."

- The text in the non-normative sections needs to be changed appropriately.