

OVERVIEW:

With mantis 928, formal arguments to properties and sequences are defined to apply to a list of arguments that follow, much like tasks and function arguments. Previously, the type had to be replicated for each list item. Untyped arguments must therefore be listed first before any typed arguments. The “context” type is introduced to allow arguments that do not have any data type restrictions to be mixed freely with those that do. Use of “context” is equivalent to listing the argument prior to any typed arguments.

The following describes the detailed changes that will be required in the standard. All changes are RELATIVE to the revisions of Mantis 928. It does not include the changes of proposal 1549 since that has not been approved.

=====

REPLACE

A.2.10 Assertion declarations

```
sequence_formal_type ::=
    data_type_or_implicit
```

```
property_formal_type ::=
    data_type_or_implicit
```

WITH

A.2.10 Assertion declarations

```
sequence_formal_type ::=
    data_type_or_implicit
    | context
```

```
property_formal_type ::=
    data_type_or_implicit
    sequence_formal_type
```

REPLACE Syntax 17-2 and Syntax 17-4 from section 17.5 and 17.6, respectively

```
sequence_formal_type ::=
    data_type_or_implicit
```

WITH

```
sequence_formal_type ::=
    data_type_or_implicit
    | context
```

REPLACE

17.6.1 Typed formal arguments in sequence declarations

Formal arguments of sequences can optionally be typed. To declare a type for a formal argument of a sequence, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type shall be untyped.

A type name can refer to a comma separated list of arguments. Untyped arguments must therefore be listed before any typed arguments.

Exporting values of local variables through typed formal arguments is not supported.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion

expressions (see 17.4.1). The assignment rules for assigning actual argument expressions to formal arguments, at the time of sequence instantiation, are the same as the general rules for doing assignment of a typed variable with a typed expression (see Clause 4).

For example, two equivalent ways of passing arguments are shown below. The first has untyped arguments, and the second has typed arguments:

```
sequence rule6_with_no_type(x, y);  
##1 x ##[2:10] y;  
endsequence  
  
sequence rule6_with_type(bit x, bit y);  
##1 x ##[2:10] y;  
endsequence
```

WITH

17.6.1 Typed formal arguments in sequence declarations

Formal arguments of sequences can optionally be typed. To declare a type for a formal argument of a sequence, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type will be untyped. A type name can refer to a comma separated list of arguments. ~~Untyped arguments must therefore be listed before any typed arguments.~~

Exporting values of local variables through typed formal arguments is not supported.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion expressions (see 17.4.1). There are two ways to achieve implicit typing of arguments. The first is to write the implicitly typed arguments at the beginning of the formal argument list, prior to any typed argument. The second is to use the *context* type. Because a type applies to multiple comma-separated arguments, the *context* type is required if an implicitly typed argument is to be placed after a typed argument in the formal argument list. The *context* type specifies that the

semantics for binding to the argument shall be as though the argument were written at the beginning of the formal argument list, prior to any typed argument.

The assignment rules for assigning actual argument expressions to formal arguments, at the time of sequence instantiation, are the same as the general rules for doing assignment of a typed variable with a typed expression (see [Clause 4](#)).

For example, ~~two~~-three similar ways of passing arguments are shown below. The first has untyped arguments, and the second and third have ~~has~~ equivalent typed arguments. The first example does not specify any types, so the types of the actual arguments instantiated are used for semantic checks. Similarly, in the second and third example, “w” has no specified type so the type of the actual argument instantiated is used for semantic checks.

```
sequence rule6_with_no_type(w, x, y, z);  
w ##1 x ##[2:10] y ##1 (z == 8'hFF);  
endsequence
```

```
sequence rule6_with_type_1(w, bit x, y, byte z);  
w ##1 x ##[2:10] y ##1 (z == 8'hFF);  
endsequence
```

```
sequence rule6_with_type_2(bit x, y, context w, byte z);  
w ##1 x ##[2:10] y ##1 (z == 8'hFF);  
endsequence
```

REPLACE Syntax 17-14 from section 17.11

```
property_formal_type ::=  
    data_type_or_implicit
```

WITH

```
property_formal_type ::=  
    data_type_or_implicit  
    sequence_formal_type
```

REPLACE

17.11.1 Typed formal arguments in property declarations

Formal arguments of properties can optionally be typed. To declare a type for a formal argument of a property, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type shall be untyped. A type name can refer to a comma separated list of arguments. Untyped arguments must therefore be listed before any typed arguments.

The supported types for property formal arguments include all the types that are allowed for sequences.

WITH

17.11.1 Typed formal arguments in property declarations

Formal arguments of properties can optionally be typed. To declare a type for a formal argument of a property, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type shall be untyped. A type can refer to a comma separated list of arguments.—~~Untyped arguments must therefore be listed before any typed arguments.~~

The supported types for property formal arguments include all the types that are allowed for sequences. There are two ways to achieve implicit typing of arguments. The first is to write the implicitly typed arguments at the beginning of the formal argument list, prior to any typed argument. The second is to use the **context** type. Because a type applies to multiple comma-separated arguments, the **context** type is required if an implicitly typed argument is to be placed after a typed argument in the formal argument list. The **context** type specifies that the semantics for binding to the argument shall be as though the argument were written at the beginning of the formal argument list, prior to any typed argument.