

## OVERVIEW:

Formal arguments to properties and sequences are currently defined for some but not all possible types. The objective of this proposal is to expand the list of types so that everything that is allowed to be passed as an argument can be passed as a typed argument. **(NOTE dependency on 1730 that calls for allowing sequence expressions or property expressions – this was written without regard to that.)**

The standard currently defines only operand types (per 17.4.1). Arguments that are not covered by the current type definitions include “property”, “sequence”, and “events”

New Types are proposed as follows:

- sequence: sequence instances are passed as type sequence
- property: property instances are passed as type property
- event: this is used for passing arguments (4.8) that are used for clocking purposes
- strings: can be passed for use in messaging: (IS THIS COVERED BY DATA\_TYPE\_OR\_IMPLICIT AND IS IT INCLUDED AS AN SEQUENCE\_ACTUAL\_ARG – I THINK WE NEED TO ADD THIS)

Examples have been improved to demonstrate:

- integers must be used for delay and repetition
- “equivalent” typing (reference 6.9.2) is used for passing non-temporal arguments. Sequence arguments can accept Boolean or sequence arguments. Property arguments can accept boolean, sequence, or property type arguments.
- the passing of events

The following describes the detailed changes that will be required in the standard. All changes are RELATIVE to the revisions of Mantis 928 (already reflected in the p1800 draft) and 1601 (which adds context type).

=====

REPLACE

### A.2.10 Assertion declarations

```
property_formal_type ::=
    sequence_formal_type
```

...

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

WITH

### A.2.10 Assertion declarations

```
property_formal_type ::=
    sequence_formal_type
    | property
```

...

```
sequence_formal_type ::=
    data_type_or_implicit
    | context
    | sequence
    | event
```

### REPLACE Syntax 17-2 from section 17.5

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

WITH

```
sequence_formal_type ::=
    data_type_or_implicit
    | context
    | sequence
    | event
```

### REPLACE Syntax 17-4 from section 17.6

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

WITH

```
sequence_formal_type ::=  
    data_type_or_implicit  
    | context  
    | sequence  
    | event
```

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 will be untyped. A type name can refer to a comma separated list of 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, three similar ways of passing arguments are shown below. The first has untyped arguments, and the second and third have 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
```

Another example, in which a local variable is used to sample a formal argument, shows how to get the effect of “pass by value”. Pass by value is not currently supported as a mode of argument passing.

```
sequence foo(bit a, bit b);
  bit loc_a;
  (1'b1, loc_a = a) ##0
  (t == loc_a) [*0:$] ##1 b;
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

When a type is specified for a formal argument, it enforces semantics checks over the actual argument. An integer type requires an actual argument with an equivalent type (equivalent types are defined in (6.9.2)). Sequence type requires an actual argument that is either a Boolean expression or a sequence instance.

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). In addition, strings may be passed for use in messaging, but only for use in messaging or as an argument to a function or task. Sequence instances may be typed using the *sequence* type.

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, three similar ways of passing arguments are shown below. The first has untyped arguments, and the second and third have 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. Arguments “x” and “y” must be of a type that is equivalent to type **bit**, and argument “z” must be a type that is equivalent to type **byte** (i.e. “x” and “y” must be 1 bit and “z” must be 8 bits.)

```
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
```

Typed formal arguments that are used to pass delay and repetition values must be of integer type. For example, two equivalent ways of passing delay and repetition arguments are shown below:

```
sequence delay_arg_example ( shortint delay1, delay2, min, max);
x ##delay1 y[*min:max] ##delay2 z;
endsequence
```

```
`define my_delay 2;
cover property ( delay_arg_example ( `my_delay, `my_delay-1, 3, $) );
```

which is equivalent to:

```
cover property ( x ##2 y[*3:$] ##1 z);
```

Paraphrasing: Parentheses are always implicit for passing expressions as arguments. Actual arguments that consist of expressions are checked at compile time for compatibility with the types of the corresponding formal arguments. Argument passing is done by substitution (refer to the section on formal semantics E.3.2??).  
 |

When an argument type is **event**, semantic checks ensure that the argument is a legal event expression and that it is used for clocking purposes. The event\_expression argument replaces the entire content of the event argument in @(event). Any legal event\_expression is allowed. The following shows an example of passing events:

```
sequence event_arg_example ( event ev )
@(ev) x ##1 y;
endsequence

cover property ( event_arg_example(posedge clk) );
```

is equivalent to:

```
cover property ( @(posedge clk) x ##1 y);
```

If the intent is to pass only a signal that is not an entire event\_expression, then the argument must be passed as a signal type, not event. For example,

```
sequence event_arg_example ( reg sig )
  @(posedge sig) x ##1 y;
endsequence

cover property ( event_arg_example(clk) );
```

is equivalent to:

```
cover property ( @(posedge clk) x ##1 y);
```

Another example, in which a local variable is used to sample a formal argument, shows how to get the effect of “pass by value”. Pass by value is not currently supported as a mode of argument passing.

```
sequence foo(bit a, bit b);
bit loc_a;
(1'b1, loc_a = a) ##0
(t == loc_a) [*1:$] ##1 b;
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
  | property
```

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 can refer to a comma separated list of 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.

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

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

```
property rule6_with_no_type(x, y);
##1 x |-> ##[2:10] y;
endproperty

property rule6_with_type(bit x, bit y);
##1 x |-> ##[2:10] y;
endproperty
```

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.

The supported types for property formal arguments include all the types that are allowed for sequences plus the addition of the **property** type. Specifically, all types that are allowed as operands in assertion expressions (see 17.4.1) are allowed as formal arguments. Sequence instances may be typed using the **sequence** type. Property instances may be typed using the **property** type.

When a type is specified for a formal argument, it enforces semantics checks over the actual argument. An integer type requires an actual argument with an equivalent type (equivalent types are defined in (6.9.2)). **sequence** type requires an actual argument that is either a Boolean expression or a sequence instance. **property** type can accept Boolean expressions, sequence instance, or property instance arguments.

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 arguments to formal arguments, at the time of property instantiation, are the same as the general rules for doing assignment of a typed variable with another typed expression (see [Clause 4](#))

For examples of using formal arguments, refer to section 17.6.1.

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

```
property rule6_with_no_type(x, y):  
##1 x |> ##[2:10] y;  
endproperty
```

```
property rule6_with_type(bit x, bit y):  
##1 x |> ##[2:10] y;  
endproperty
```