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

The standard currently defines only operand types (per 16.5.1). Arguments that are not covered by the current type definitions include “property”, “sequence”, and “events”

New Types are proposed as follows:

- sequence: sequence expressions are passed as type sequence
- property: property expressions are passed as type property
- event: this is used for passing arguments (4.8) that are used for clocking purposes

Deleted: instances

Deleted: instances

Examples have been improved to demonstrate:

- integers must be used for delay and repetition
- automatic type casting occurs for non-temporal arguments, consistent with functions and tasks. (Note: Mantis 1804 will propose a type qualifier that would require equivalent typing as an alternative.)
- Sequence arguments can accept Boolean or sequence expression arguments. Property arguments can accept boolean, sequence expression, or property expression type arguments.
- the passing of events

The following describes the detailed changes relative to Draft 3 that will be required in the standard. All changes are RELATIVE to the revisions (which adds context type), and 1730 (that says the seq and property actual args can be seq and property expressions, respectively)

Adding a sentence that states that passing automatic variables as arguments shall result in an error. This is consistent with the text on 16.5.2 and that for sequence events.

==========

REPLACE A.2.10 Assertion declarations

```
property_formal_type ::=
        sequence_formal_type

…

sequence_formal_type ::=
        data_type_or_implicit
        | context
```

WITH

property_formal_type ::=
    sequence_formal_type
    | **property**

…

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

REPLACE Syntax 16-4 from section 16.7 ~~17.6~~

sequence_formal_type ::=
    data_type_or_implicit
    | context

WITH

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

REPLACE 16.7.1 Typed formal aruments in sequence declarations (form Manits 1601)

A formal argument of a sequence may be typed by specifying the data type prior to the formal argument identifier. A data type shall apply to all formal arguments whose identifiers both follow the data type and precede the next data type, if any, specified in the formal argument list.

The data type specified for a formal argument of a sequence may be the keyword **`context`**. A formal argument of a sequence is said to be *contextual* if either its data type is **`context`** or there is no data type preceding its identifier in the formal argument list. The semantics of binding an actual argument expression to a contextual formal argument shall be the same regardless of which of these two criteria is satisfied. This semantics is described in Subclause 16.7. The keyword **`context`** shall be used if a contextual formal argument follows a data type in the formal argument list.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion expressions (see 16.5.1) and the keyword **`context`**.

Local variable values may be exported from a sequence only through contextual formal arguments.

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 6).

For example, two ways of declaring arguments are shown below. All of the formal arguments of foo1 are contextual. The formal arguments w and y of foo2 are contextual, while the formal argument x has data type bit.

```
sequence foo1(w, x, y);
  w ##1 x ##[2:10] y;
endsequence

sequence foo2(w, bit x, context y);
  w ##1 x ##[2:10] y;
endsequence
```

The following instances of foo1 and foo2 are equivalent:

```
foo1(.w(a), .x(bit'(b)), .y(c))

foo2(.w(a), .x(b), .y(c))
```

WITH

–16.7.1 Typed formal arguments in sequence declarations

A formal argument of a sequence may be typed by specifying the data type prior to the formal argument identifier. A data type shall apply to all formal arguments whose identifiers both follow the data type and precede the next data type, if any, specified in the formal argument list.

The data type specified for a formal argument of a sequence may be the keyword **`context`**. A formal argument of a sequence is said to be *contextual* if either its data type is **`context`** or there is no data type preceding its identifier in the formal argument list. The semantics of binding an actual argument expression to a contextual formal argument shall be the same regardless of which of these two criteria is satisfied. This semantics is described in Subclause 16.7. The keyword **`context`** shall be used if a contextual formal argument follows a data type in the formal argument list.

When a type is specified for a formal argument, it enforces semantic checks over the actual argument. Actual arguments that consist of expressions are checked at compile time for cast compatibility (refer to section 6.22.4) with the types of the corresponding formal arguments. An actual argument is automatically cast to the formal type.

Argument passing is done by substituting each argument with the actual argument cast to the specified type, and using the rewrite rules defined in [Note to Editor – insert reference to the

sectionon the rewrite rules - I think this is F.2.3] to flatten property and sequence instances. When passing expressions as arguments, parentheses are always implicit.

Local variable values may be exported from a sequence only through contextual formal arguments.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion expressions (see 16.5.1) and the keyword **context**. In addition, sequence expressions may be typed using the ***sequence*** type and the ***event*** type. A formal argument of type **sequence** can accept a Boolean expression or a sequence expression actual argument. An actual arg of type **property** would cause an error when the formal type is **sequence.**

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 6).

For example, two ways of declaring arguments are shown below. All of the formal arguments of foo1 are contextual. The formal arguments w and y of foo2 are contextual, while the formal argument x has data type bit.

```
sequence foo1(w, x, y);
  w ##1 x ##[2:10] y;
endsequence

sequence foo2(w, bit x, context y);
  w ##1 x ##[2:10] y;
endsequence
```

The following instances of foo1 and foo2 are equivalent:

```
foo1(.w(a), .x(bit'(b)), .y(c))

foo2(.w(a), .x(b), .y(c))
```

In the above example, if b is actually 8 bits wide, it will be truncated to a bit since it is being passed to an argument of type bit, and then processed. Similarly, if a bit of data is passed to an argument of type byte, the bit is extended to a byte.

Typed formal arguments that are used to pass delay and repetition values must be of a 2-state **int** type or untyped. An actual argument of $ shall only be passed to an untyped argument. For example,

```
sequence delay_arg_example ( shortint delay1, delay2, min, context 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);.
```

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));
```

REPLACE Syntax 16-14 ~~17-14~~ from section 16.12 (~~17.11~~)

```
property_formal_type ::=
        sequence_formal_type
```

WITH

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

REPLACE

16.12.1 Typed formal arguments in property declarations

Formal arguments of properties can optionally be typed. The supported types for property formal arguments include all the types that are allowed for sequences. Refer to 16.7.1.

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
```

WITH

~~17.11.1~~ 16.12.1 Typed formal arguments in property declarations

Formal arguments of properties can optionally be typed. The supported types for property formal arguments include all the types that are allowed for sequences plus the addition of the ***property*** type. A formal argument of type **property** can accept a Boolean expression, sequence expression, or property expression actual argument. The rules for passing arguments to properties is the same as those for sequences. Refer to 16.7.1.

~~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
```