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 17.4.1). Arguments that are not covered by the current type definitions include “property”, “sequence”, “events”, and “implicit”. The “implicit” is introduced to allow arguments that do not have any data type restrictions to be mixed freely with those that do.

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
- implicit: used when there are no data type restrictions, meaning that any type is acceptible. The implicit type (that of the declaration of the argument) is used for any semantic checks. This is equivalent to listing the argument prior to any typed arguments.

Examples have been improved.

The following describes the detailed changes that will be required in the standard. All changes are RELATIVE to the revisions of Mantis 928. Also, the changes of Mantis 1532 is assumed: “Remove @sequence_instance from event_control to take care of sequences with arguments”

==

REPLACE

A.2.10 Assertion declarations

property_actual_arg ::=
    sequence_actual_arg
    | property_instance

property_formal_type ::=
    data_type_or_implicit

sequence_formal_type ::=
    data_type_or_implicit

sequence_actual_arg ::=
    event_expression

WITH

A.2.10 Assertion declarations

property_actual_arg ::=
    sequence_actual_arg
    | property_instance

property_formal_type ::=
    ~~data_type_or_implicit~~
    sequence_formal_type
    | property

sequence_formal_type ::=
    data_type_or_implicit
    | sequence
    | event
    | implicit

sequence_actual_arg ::=
    event_expression

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
    | sequence
    | event
    | implicit

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
```

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. When a type is specified, that type is enforced by semantic checks. 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). Sequence instances may be typed using the ***sequence*** type. ***implicit*** is used to specify that the argument can have any type that is legal for a sequence actual argument. There are two ways to achieve implicit typing of arguments. The first is to write the implicitly type arguments first in the list prior to specifying any type. The second is to use the ***implicit*** type.

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 example, “w” has no specified type so the type of the actual argument instantiated is used for semantic checks. Arguments “x” and “y” will be truncated to type **bit**, and argument “z” will be truncated or extended as necessary to make it of type **`byte`**.

```
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
```

```
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, implicit w, byte z);
w ##1 x ##[2:10] y ##1 z == 8’hFF;
endsequence
```

Any integer type can be used to pass delay and repetition values. 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);
```

Parentheses are implicit for passing complex expressions as arguments. Actual arguments that consist of complex expressions are checked at compile time for compatibility with the types of the corresponding formal arguments.

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 clock )
@(clock) 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 clock )
@(posedge clock) 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 name can refer to a comma separated list of arguments. Untyped arguments must therefore be listed before any typed arguments.

The supported data types for property formal arguments are the types that are allowed for operands in assertion expressions (see 17.4.1). 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 ~~name~~ can refer to a comma separated list of arguments~~. Untyped arguments must therefore be listed before any typed arguments.~~

The supported ~~data~~ 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. **implicit** is used to specify that the argument can have any type that is legal for a property actual argument. There are two ways to achieve implicit typing of arguments. The first is to write the implicitly type arguments first in the list prior to specifying any type. The second is to use the **implicit** type.

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
```

REPLACE in section 23.4

The *version_specifier* `"1800-2005"` specifies that only the identifiers listed as reserved keywords in the IEEE Std 1800-2005 are considered to be reserved words. These identifiers are listed in Table 23-1. The `‘begin_keywords` and `‘end_keywords` directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the SystemVerilog Verilog language.

WITH (note: table 23-2 is a copy of tabe 23-2 with the implicit keyword added)

The *version_specifier* `"1800-2005"` specifies that only the identifiers listed as reserved keywords in the IEEE Std 1800-2005 are considered to be reserved words. These identifiers are listed in Table 23-1. The *version_specifier* `"1800-200?"` specifies that only the identifiers listed as reserved keywords in the IEEE Std 1800-200? (NOTE TO EDITOR: fill in the date of publication) are considered to be reserved words. These identifiers are listed in Table 23-2. The
`‘begin_keywords` and `‘end_keywords` directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the SystemVerilog Verilog language

Table 23-2—IEEE Std 1800-2006 reserved keywords

```
alias
always
always_comb
always_ff
always_latch
and
assert
assign
assume
automatic
before
begin
bind
bins
binsof
bit
break
buf
bufif0
bufif1
byte
case
casex
casez
cell
chandle
class
clocking
```

cmos
config
const
constraint
context
continue
cover
covergroup
coverpoint
cross
deassign
default
defparam
design
disable
dist
do
edge
else
end
endcase
endclass
endclocking
endconfig
endfunction
endgenerate
endgroup
endinterface
endmodule
endpackage
endprimitive
endprogram
endproperty
endspecify
endsequence
endtable
endtask
enum
event
expect
export
extends
extern
final
first_match
for
force
foreach
forever
fork
forkjoin
function
generate
genvar
highz0
highz1
if
iff
ifnone
ignore_bins
illegal_bins
import
implicit
incdir
include
initial
inout
input
inside
instance
int

```
integer
interface
intersect
join
join_any
join_none
large
liblist
library
local
localparam
logic
longint
macromodule
matches
medium
modport
module
nand
negedge
new
nmos
nor
noshowcancelled
not
notif0
notif1
null
or
output
package
packed
parameter
pmos
posedge
primitive
priority
program
property
protected
pull0
pull1
pulldown
pullup
pulsestyle_onevent
pulsestyle_ondetect
pure
rand
randc
randcase
randsequence
rcmos
real
realtime
ref
reg
release
repeat
return
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
sequence
shortint
shortreal
showcancelled
signed
small
```

```
solve
specify
specparam
static
string
strong0
strong1
struct
super
supply0
supply1
table
tagged
task
this
throughout
time
timeprecision
timeunit
tran
tranif0
tranif1
tri
tri0
tri1
triand
trior
trireg
type
typedef
union
unique
unsigned
use
uwire
var
vectored
virtual
void
wait
wait_order
wand
weak0
weak1
while
wildcard
wire
with
within
wor
xnor
xor
```

REPLACE Annex B Table B-1 – Reserved keywords

```
illegal_bins*
import*
incdir
include
```

WITH

```
illegal_bins*
import*
implicit*
incdir
include
```