Introducing Whand

A.R.J.M. July 2021

Overview

Whand is a simplified computer language to control various appliances, pieces of equipment such as signals, doors, motors, dispensers etc. that are the **outputs** of a system. The **inputs** may be various sensors based e.g. on manipulation, contact, proximity, light, temperature etc.

The present document is not a user manual, but should give a flavor of the originality and capabilities of Whand.

Whand is intended to be highly **readable**. It is **declarative** and not procedural. Whand focuses on the end result (e.g. 'light *when* switch *until end* switch') and not on the means (e.g. *first check switch, if switch is on and light is off, then turn light on* etc.). The script makes explicit the relationship between output and input.

A Whand script specifies the **behavior** each appliance, switching it ON or OFF according to a set of clauses that may be simple or complex, based on delays, counters, states or lists.

An **object** is an input, an output or an intermediate variable. It is characterized by its **name**, its value at any given instant, and its set of clauses.

An object's **value** changes when one of its clauses is fulfilled. A **clause** takes the form: '*when* condition: value' (n.b. keywords are shown here in *italics*). It means that the value is computed and assigned to the object when **condition** becomes *true*

The set of clauses constitutes the **definition** of the object. An object can only have one definition and this definition determines all of its behavior.

A Whand **script** is just a set of object's definitions.

Values are expressions based on names, operators, functions or comparators. The nature (type) of the value determines the nature of the object. There are five possible natures: an **event** (either *true* or *false*), a **number** (e.g. '-3.4'), a **duration** (e.g. '7min20s'), a **state** (just a name, e.g. 'waiting') or a **list** (e.g. '24.6, 200ms, 'locked', *true*'). The nature of a given object is fixed throughout a script.

Conditions are expressions based on names, operators, functions or comparators. A condition must evaluate as *true* or *false* at any given moment, so a condition is necessarily an event.

Some advantages of Whand are:

- Whand syntax is simple with no boilerplate or fancy notation.
- Each object's behavior is entirely contained in its definition.
- Objects function in parallel and are easily integrated within a script.
- Delays do not require timers but are simply added to events.
- Most operators process lists in a single step, without an explicit loop.

On the other hand, writing a Whand script requires particular attention to the timing of events and operations.

Working with events

Events are the fundamental objects in Whand. They correspond to binary inputs, outputs or logical gates. An event can only take value ***true*** or ***false***, corresponding to ON or OFF for an appliance or a sensor. Events may change value repeatedly depending on their clauses.

In Whand, it is essential to visualize events in time (Figure 1).

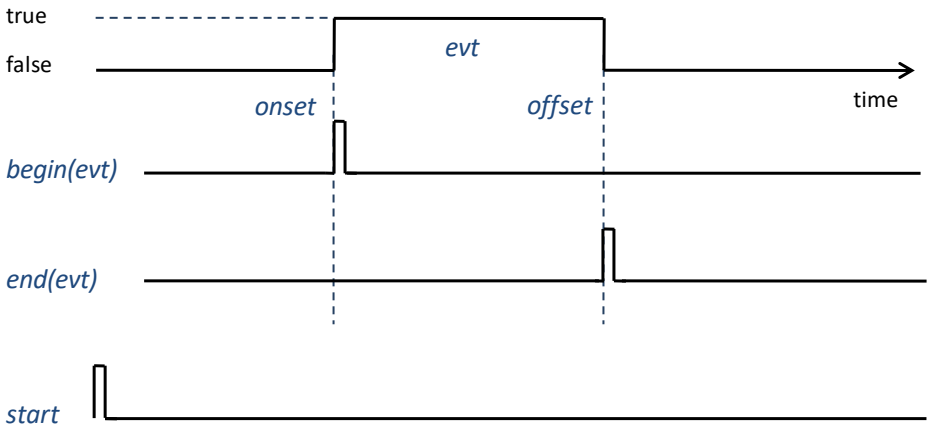

Figure 1. Event value (*true* or *false*), onset and offset, *begin* and *end*. Event *start* marks the beginning of execution of the script.

The **onset** or occurrence of an event is when its value changes from *false* to *true*. The onset of the condition in a clause is when the value is computed and assigned to the object.

The **offset** of an event is the change from *true* to *false*. It plays no role in a clause, i.e. the value assigned to an object persists and can only change at condition onset.

The onset and offset of an event can be extracted by functions ***begin*** and ***end***. Thus, '*begin* some_event' and '*end* some_event' create very brief events corresponding to onset and offset, respectively (Figure 1).

A repeating event imperatively needs an offset clause, e.g.

```
alert when trigger + guard_time: true      # condition for delayed 'alert' onset: 'trigger'
 when disable: false                       # condition for 'alert' offset: 'disable'
```

N.b. in a script, anything after # is a comment

The above piece of script is a complete definition of event 'alert'. The name 'alert' is arbitrary and has no particular meaning in Whand. Names 'trigger' and 'disable' also designate arbitrary events that must be defined elsewhere. 'guard_time' is a delay (duration) that must be defined elsewhere.

Inputs, outputs and special events

Inputs and outputs are special events that allow the program to interact with the physical world via hardware.

Output events (*output*), *exit*, and *controlpanel* require a definition.

Input events (*pin*) and *start* are internally defined and cannot be redefined. Their behavior is entirely determined by the external world.

pin designates a standard input, e.g. '*pin*(3)' or '*pin* 3'. The number of available pins depends on hardware. The state of the corresponding hardware line, ON or OFF, continuously determines the value, *true* or *false*, of the *pin*. Pins are never explicitly defined in the script. They are used directly when needed as values or conditions.

output designates a standard output, e.g. '*output*(4)' or '*output* 4'. The number of available outputs depends on hardware. The value, *true* or *false*, of the output object continuously determines the state of the corresponding hardware line, ON or OFF. Outputs must have a definition in the script to specify their behavior, e.g. '*output* 2: alert'.

start is a very brief event marking the beginning of execution of the Whand script. *start* is internally generated and used both as a time reference and as a condition to initialize objects.

exit is recognized as a special event. When *exit* becomes *true*, it terminates the execution of the script, e.g. '*exit when start* + 30 min' determines a 30 min session. Various *exit* conditions may be used.

controlpanel is recognized as a special event. If *controlpanel* appears in the script, a dashboard with a selection of objects is displayed. The user may then monitor the value of the objects and interact with them. The objects to display are selected via instruction ***show***, e.g. '*show*(guard_time, *count*(alert))'. The selected object's history is saved on *exit*.

print is a used to display information on the screen. Only one definition of object *print* is allowed, i.e. all prints must be declared in the same block of script, each with its own condition, e.g.

```
print when start: "Hello World! "                    # same as    print: "Hello World! "
  when start + 1h: "One hour has elapsed"
  when exit: "Bye! "
  when alert: "Please check equipment! "
```

N.b. two prints may not occur simultaneously, e.g. at *start*. Like any object, *print* can only have one value at any given instant.

Other special events include input functions *load*, *measure* and *read*, and output functions *store*, *command, write* and *display*, depending on hardware configuration.

Simplified syntax

The definition of an object starts with its name followed by clauses. A definition may contain several *when* clauses. Each definition ends where the next definition starts. The order of definitions is indifferent. Clauses are tested in parallel, so the order of clauses is also indifferent.

All Whand clauses may be written in the form '*when* condition: value' ('condition' and 'value' being any names or expressions). However, for convenience, some simplified syntax is allowed.

- *when* condition means '*when* condition: *true*'
- *until* condition means '*when* condition: *false*'
- name (i.e. an empty definition) means 'name *when start*: *true*' (permanently *true*)
- name: value means the same as two clauses:

    name *when start*: value
    when change value: value

meaning that the object will continuously track value, without timing or condition. With this syntax, one can create various aliases for constants, expressions, inputs and outputs, e.g.

```
guard_time: 30s                    # alias for a constant (recommended)
mean(x'): cumul(x')(-1)/count x'   # define new user-function 'mean'
output 2: light                    # alias for an output (keyword on the left)
switch: pin 5                      # alias for an input (keyword on the right)
```

Durations and numbers

Durations can be expressed in milliseconds (ms), seconds (s), minutes (min or mn), hours (h), days (day) or weeks (wk), before being internally converted to seconds, e.g.

2wk 1 day 5h 10mn 12s300 ms represent 1 314 612.3s

epsilon is a predefined internal duration. It is extremely brief, but very useful to avoid simultaneity. Simultaneity in a parallel system tends to render the order of operations unpredictable and to interfere with value updating.

Numbers can be positive or negative, integer or floating point. Their internal representation is floating point values. When comparing numbers or durations, Whand allows for rounding errors, e.g. 1.99999995 is considered equal to 2.

Durations are quite distinct from numbers, but they can be combined provided the result is either a duration or a number. Thus, the following operations are legal in Whand:

1s-0.4min+200ms 1+4.9*3 -2.5*3.3s 4s*12 2mn/4 3mn/5s

Conversely, the following operations are *not* legal in Whand:

30s+1 10s*10s 60/1s

Working with delays

Delays are another essential feature of Whand. They are typically used to trigger a clause. A **non-resetting delay** is created by adding a specified duration to an event, e.g. '*start* + 30 min'. It is appropriate when the delay is fixed.

This syntax creates an event with the following properties (Figure 2):

- the delayed event is internally generated and triggered. It is initially *false*. Its onset is delayed by the specified duration with respect to the *onset* of the original event.
- the offset of the delayed event is also delayed by the specified duration, so the delayed event has the same duration as the original event.
- the delayed event cannot be cancelled, reset or postponed after the onset of the original event.

A delayed event can make reference to the original event, e.g.

clock_tick *when* (*start or* clock_tick + 1s) *until* clock_tick + 50ms # repeating ticks

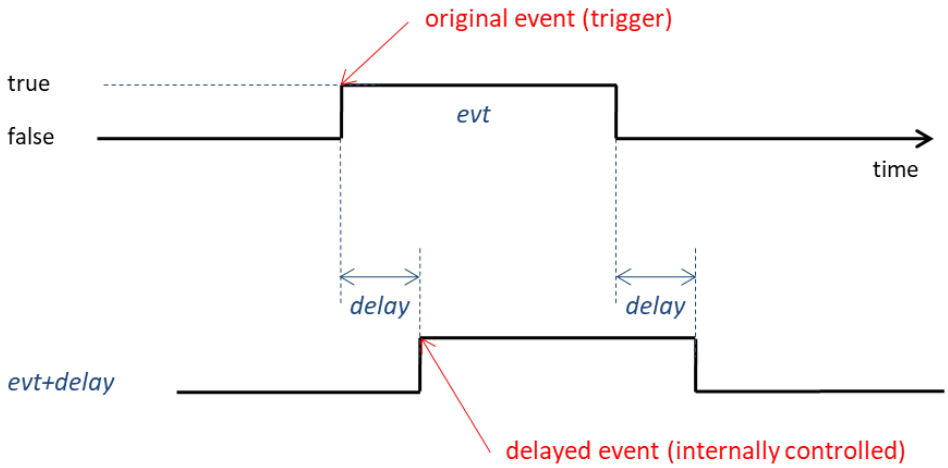

Figure 2. A non-resetting delay.

A **resetting delay** is created with function *since*. It is appropriate when the duration of the delay is unknown, e.g. 'open *when* movement *until* safe_time *since* movement'.

A delayed event created with *since* has the following properties (Figure 3):

- the delayed event is delayed by the specified duration after the *offset* of the original event, unless '*since begin*' is used.
- if the original event re-occurs before the end of the delay, the delayed event is cancelled and the delay starts again (resets) at event offset.
- after the delay has fully elapsed, the delayed event remains *true* as long as the original event does not re-occur (indefinite duration).

The following script uses a resetting delay (Figure 3):

```
# AUTOMATIC CAT FLAP WITH MOVEMENT DETECTOR
# Parameters
safe_time: 3s                          # time after last movement

# Outputs and inputs
output(1): open                        # to open flap
movement: pin(1)                       # from movement detector

# Procedure
open when movement
   until safe_time since movement      # resetting delay
```

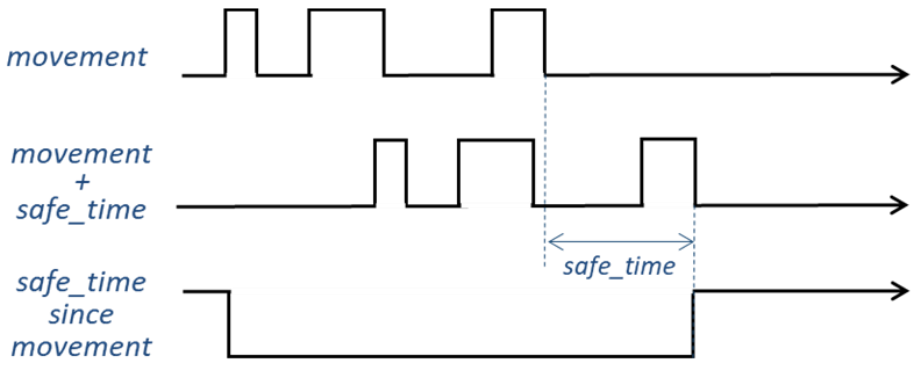

Figure 3. Comparing non-resetting (movement+safe_time) and resetting delays (safe_time *since* movement). Both are internally generated in response to event. Obviously, a non-resetting delay would not be appropriate for a cat flap.

Working with counters

Counting is very useful in programming. Function *count* immediately provides the total number of occurrences of an event (also, when applied to a list, the number of elements in the list). Combined with delays, *count* can be extremely powerful, e.g.

```
frequency_per_minute: count contact – count(contact + 1mn)
# because the delayed events from the previous minute have yet to occur
```

The following script uses a counter for rewards. An animal in a box will receive a reward every time a lever is pressed.

A delay is provided for reward consumption. The lever press must begin after the delay has elapsed. The indefinite duration of the resetting delay (*since*) is used here so that the lever press period is not limited in time.

There are two *exit* conditions, one based on a time limit (non-resetting delay) and the other based on reward count, whichever occurs first.

A time calibrated pulse is sent to activate the reward dispenser.

```
# SKINNER BOX
# Parameters
max_rewards: 20                          # alias for a number
max_session: 15min                       # alias for a duration
dispenser_time: 500ms                    # pulse duration for reward dispenser
eating_delay: 5s                         # time for reward consumption
```

```
# Outputs and inputs
output 3: reward                         # to reward dispenser
output 5: house_light                    # to light on ceiling
press: pin 4                             # from lever press detector
```

```
# exit conditions
exit when start + max_session            # time limit
   when (count reward = max_rewards) + eating_delay   # reward number limit
```

```
# Procedure
house_light                              # ON for the whole session
reward when (count reward=0 or eating_delay since reward) and begin press
   until reward + dispenser_time         # fixed pulse duration
```

A disadvantage of function *count* is that there is no way to reset it to zero. One solution to this problem is to keep a memory of counts:

```
current_count: count click – prior_count
prior_count when start: 0                # definition of 'prior_count' (two clauses)
when reset + epsilon: count click        # epsilon is needed if reset depends on current_count
```

Another solution is to create an incrementing counter as in the following example:

```
# PACKING AUTOMATON
# Parameters
capacity: 12                             # bottles per box
packing_time: 1.5s
command_pulse_time: 100ms                # fixed duration
```

```
# Outputs and inputs
output 1: close_box                      # to hardware
output 2: get_new_box
bottle: pin 1                            # from sensor
```

```
# Procedure
get_new_box when start                   # onset condition
when close_box + packing_time            # other onset condition
until get_new_box + command_pulse_time   # offset condition (fixed duration)
```

```
close_box when counter=capacity
until close_box + command_pulse_time     # offset condition (fixed duration)
```

```
counter when start: 0
when (counter=capacity) + epsilon: 0     # reset condition (briefly delayed)
when bottle: old + 1                     # old is previous value of counter
```

```
# Display
controlpanel
show capacity, get_new_box, close_box, bottle, counter, count(get_new_box)
```

Function ***old*** refers to the value of the object at the current time step before any change in value. It is needed every time an object must change in reference to itself (e.g. increment).

Writing '*when* bottle: counter + 1' would instead create an infinite updating loop, because 'counter' could never be equal to 'counter + 1'.

Working with lists

In Whand, a **list** is a collection of objects of any nature. It is typically either declared as a sequence of object names, separated by commas, or loaded from a text file, e.g. 'my_list *when start*: ***load***(myfile.txt)'. Function ***ramp*** generates a list of consecutive integers e.g. '*ramp* 4' is list (1,2,3,4).

List elements may be successively extracted using function ***next***:

```
random_value: 3,5,2,9,4,1,6,8,7          # declare a list

current_value when start: next random_value   # initialize current value to 3
when get_new_value: next random_value         # get next value from list
```

Lists are cyclic, so here *next* may retrieve the sequence 3,5,2,9,4,1,6,8,7,3,5,2,9,4,1,6…

Specific elements of a list can be addressed by index, e.g. 'random_value(4)' is 9. Negative indices work backwards from the end of the list, e.g. 'random_value(-1)' is the last element (7) and 'random_value(-3)' is 6.

An **association list** is defined element-by-element, rather than as a single chunk, e.g. 'height(table): 800' 'height(chair): 400', etc. Here, the list is 'height' and the subscript of each element may be a name ('table', 'chair') rather than a number. Retrieving information associated with a name is often easier than when a list is indexed by a number.

Distributivity on lists

A list can be used as a vector to avoid repeating the same operation element by element. Indeed, most operations with lists can be automatically performed on all elements, eliminating the need for a loop e.g. '(1,2,3)*100ms' equals list '(0.1s, 0.2s, 0.3s)'. This is **distributivity**.

Consider for instance symbol '=' used to compare two simple values. Applying it to two lists involves distributivity and produces a *list* of *true/false* results. To further check whether the two lists are equal, operator ***all*** or ***match*** must be used, e.g. *'all*(a,b,c = 1,2,3)' or '(a,b,c) *match* (1,2,3)'.

With distributivity, a list may be used to index another list, e.g. '(a,b,c,d)(-*ramp* 3)'. '-*ramp* 3' gives (-1,-2,-3), so the result is (d,c,b).

Besides distributivity, several list functions are available, e.g. *count*, *cumul*, *is in*, *find*, *sort*, *pick*, *match*, *all*, *any*, etc. The following example combines list functions and distributivity:

```
L: 2,3,2,1,3,4,2,2,4                          # list with duplicates
no_duplicates: L pick ((L find L) = ramp(count L))   # gives list 2,3,1,4
```

This formula may be decomposed as follows:

'*ramp*(*count* L)' is *ramp*(9) i.e. (1,2,3,4,5,6,7,8,9)

Function ***find*** gives the first position of an element in a list, e.g. 'L *find* 4' is 6. 'L *find* L' is a distributive use of *find* that gives the list of positions of all elements, i.e. (1,2,1,4,2,6,1,1,6).

Comparing both lists gives (*true, true, false, true, false, true, false, false, false*)

Finally, operator ***pick*** extracts from L the elements corresponding to the *true* values, i.e. (2,3,1,4).

States

Unlike an event that can only take value *true* or *false*, a **state** can take one of several values designated by names, e.g. 'off', 'standby' and 'on'.

The main operations allowed with states are ***print***, ***is*** or ***is in***, e.g. '*print when* phase *is* wait: "This operation may take a few minutes." '. Quotes (") are allowed around a state name. They are mandatory if the name contains spaces.

A variable state needs a definition (e.g. 'phase'), but a constant state may be used without any definition (e.g. 'wait', 'This operation may…').

The following (simplified) vending machine script illustrates the use of states:

```
# COFFEE VENDING MACHINE
# Parameters: duration of each delivery
cup_time: 1s
milk_time: 3s
sugar_time: 1.5s
coffee_time: 15s
```

```
# Outputs and inputs
output 1: cup                                   # activate delivery
output 2: milk
output 3: sugar
output 4: coffee
coin_inserted: pin 1                            # actions or
coffee_chosen: pin 2                            # buttons to press
latte_chosen: pin 3
cup_removed: pin 4
```

```
# Procedure
current_state when start: standby               # states
   when current_state is standby and coin_inserted: select
   when current_state is select and coffee_chosen: make_coffee
   when current_state is select and latte_chosen: make_latte
   when end coffee: finished
   when cup_removed: standby
cup when current_state is in (make_coffee, make_latte)   # timing of deliveries
   until cup+cup_time
milk when current_state is make_latte and end cup
   until milk+milk_time
sugar when current_state is make_coffee and end cup
   when current_state is make_latte and end milk
   until sugar+sugar_time
coffee when end sugar
   until coffee+coffee_time
```

```
# Display
print when current_state is standby: "Insert coin"
   when current_state is select: "Select beverage"
   when current_state is in (make_coffee, make_latte): "Preparing... "
   when current_state is finished: "Please take your cup"
controlpanel
show current_state, cup, milk, sugar, coffee
```

Outputs ‘cup’, ‘milk’, ‘sugar’ and ‘coffee’ each have a full definition.

‘current_state’ is a variable state that can take the following values: ‘standby’, ‘select’, ‘make_coffee’, ‘make_latte’ and ‘finished’ (states without definitions).

‘(make_coffee, make_latte)’ is a list of two states.

Syntax: extra spaces are ignored. Order of definitions is indifferent.

A more complete example:

```
# ELEVATOR SIMULATION (simplified)
# parameters ==================================================
sample_time: 100ms                                  # for position
height_step: 0.025                                  # elementary move
waiting_time: 2s                                    # pause when reaching floor
accuracy: 0.02                                      # position vs. floor

# outputs =====================================================
output 1: cabin is moving_up                        # to motor
output 2: cabin is moving_down

# inputs ======================================================
goto(2): pin(12)                                    # association list
goto(1): pin(11)                                    # buttons pressed
goto(0): pin(10)

# sample ======================================================
sample when start                                   # create a clock
        when sample+sample_time
        until sample+epsilon

# compute goal ================================================
queue_goto when start: empty                        # operations on list
        when any begin goto: old add ((have goto) pick (begin goto)) # new button pressed
        when cabin is immobile and sample+epsilon: old pick (old!=position) # remove floor

goal when sample: queue_goto(1)                     # get first element of list

up: position<goal-accuracy                          # goal is up
down: position>goal+accuracy                        # goal is down
stay: absv(goal-position)<=accuracy                 # goal is here

# elevator status =============================================
cabin when stay: waiting                            # states
        when (cabin is waiting)+waiting_time: immobile
        when sample+epsilon and up and cabin is not waiting: moving_up
        when sample+epsilon and down and cabin is not waiting: moving_down

position when start: 0                              # in reality, position is read from a sensor
        when sample+2*epsilon and (cabin is moving_up): old+height_step
        when sample+2*epsilon and (cabin is moving_down): old-height_step
        when (cabin is waiting)+sample_time: intg(old+.5)       # round to floor number

# display =====================================================
controlpanel
show goal, position, cabin, queue_goto
```

Here is how this script was designed:

1) outputs: one for move up, another for move down
2) inputs: buttons for destination that may be pressed at any time
3) destination ('queue_goto'): a list incremented with each button press and trimmed when each goal is reached
4) elevator position needs to be sampled
5) a sequence is needed to read buttons, compute goal, choose direction, monitor position

To detect that a button is pressed, association list 'goto' is used (0 is not a valid index for an ordinary list). Each element of 'goto' is associated with an input *pin*.

'*any begin* goto' detects that one of the elements becomes *true*, i.e. a button is pressed.

Which button is pressed is detected by first extracting the list of buttons: '*have* goto' which is list (2, 1, 0) according to the order of definition of 'goto' elements, and then picking the one that just begins: '(*have* goto) *pick* (*begin* goto)'. The result is a list of one element (2, 1 or 0) which is then appended (*add*) to the current (*old*) 'queue_goto' list.

To remove the current floor from the list, one simply selects elements of 'queue_goto' that are not equal to the current floor (2, 1 or 0): '*old* pick (*old*!=position)'. '*old*!=position' is a distributive use of '!=' that gives *false* in each place where 'queue_goto' is equal to 'position'.

'sample' is a standard clock that gives a brief (*epsilon*) pulse every sample_time (100ms).

'up', 'down' and 'stay' (events) are directions computed from goal and current position.

'goal' (number) is updated on the 'sample' clock pulse.

'cabin' (state) is updated *epsilon* time after the clock pulse, i.e. after goal has been updated. Movement is initiated only when the state is not 'waiting', i.e. after a pause on the goal floor.

'position' (number) is updated 2**epsilon* time after the clock pulse, i.e. after goal and state have been updated. '*intg*(*old*+.5)' rounds position to the nearest integer (*intg* truncates a number to the lower integer).

This mode of functioning is only illustrative and rudimentary. To control an actual elevator, more sophisticated conditions would be needed.

User-defined functions

Defining one's own functions may be useful in order to use them multiple times, or to make the script more readable. Suppose we wish to compute a percentage of occurrence between two events, e.g. 100**count*(a)/(*count*(a)+*count*(b)), but also want to avoid dividing by zero at start (not a fatal error, but Whand would still give a warning):

```
percent(x', y'): 100*count x'/nonzero'                    # define a user function
nonzero': when start: 1                                   # intermediate variable
when change(count x'+count y'): count x'+count y'
```

Note that function arguments x' and y', as well as intermediate variable nonzero' end with a prime ('). This identifies virtual arguments in a user-defined function. Virtual arguments are placeholders for real arguments. They are never defined and have no actual value.

To call the function, we can now write 'percent(a, b)' with a and b being real (defined) events. The function may be used multiple times with various real arguments.

Examples of user-defined functions:

```
# change event duration
between(e', f'): when e'                    # duration of e' does not matter
until f'                                    # if f' never occurs, remains true

# smallest element in a list
minimum(L'): (sort(L'))(1)                  # first element in sorted list

# summate elements in a list
sum(v'): cumul(v')(-1)                      # last element in cumulated list

# mean of a list of numbers or durations
mean(my_list'): sum(my_list')/count my_list'   # applies user function 'sum'

# clock ticks, perpetual or not (perpetual if gate' is always true)
tick(gate', period', duration'): when gate'
when old+period' and gate'                  # tick repeat time
until old+duration'                         # tick duration
until end gate'                             # not exceeding gate

# sequencing function                       # applied as when t(1)… when t(2)
t(n'): tick(true, 50ms, epsilon)+n'*epsilon # delay with respect to perpetual ticks

# events from a list of intervals
trigger(intervals'): any(start+cumul intervals')   # adding a list of delays to start
```

To re-use a piece of script such as a function definition, it is possible to save it as a text file, and then to import it with instruction *include*, e.g. *include* "standard_functions.txt".

Conclusion

In brief, Whand describes the behavior of objects in a straightforward manner, using simple or abstract conditions. Whand renders implicit and automatic a number of functions that need to be explicit in a procedural language. Delays are seamlessly integrated with events. Lists are treated globally using distributivity and list functions instead of loops. The result is a code that is compact yet intuitive.

Whand is currently used in our lab with Imetronic® equipment.