Whand for beginners

Alain R Marchand - June 2020

This document is not a systematic presentation, but an overview of Whand based on particular examples. The interested reader may look up WHAND REFERENCE MANUAL for details.

Why you may need Whand

Whand is a language to control automated objects. A program usually follows sequential programming, with a sequence of operations and conditional branches. It is often a challenge for a non-programmer. For instance, a program for a Skinner box to distribute a reward for each press on a lever could be built on the following logic (in a pseudo-language). The program will stop either after a certain time, or after a certain number of rewards have been earned, whichever comes first.

```
# comment: initialization
Present Lever
Start timer
Initialize reward count to zero
# comment: processing loop
REPEAT {
        IF time elapsed then QUIT
        IF reward count = max reward then QUIT
        IF lever pressed then {
                give reward
                increment reward count
                }       # end of IF block
        }     # loop on REPEAT
# comment: QUIT is supposed to retract lever and terminate the program
```

Even such a simple script may stump non-programmers. The order of instructions is critical. Nesting of the REPEAT and IF blocks and exiting to the QUIT procedure are not straightforward. The program must not get stuck waiting for an event that never comes. Variables must be initialized and the counter and timer must be incremented and tested.

Then, if one wants to control events in parallel, to implement pseudo-random sequences or to coordinate different phases of a trial, we reach yet another level.

Alternatives to sequential programming have been proposed, such as graphic representations, or parallel state representations. Still, the syntax of many common languages remains obscure to a non-specialist. The Whand language is non sequential and aims at making programs more easily comprehensible.

Principles of Whand

Whand was primarily designed to be readable, so that it should be as simple as possible to tailor an existing program to your needs. Several features of Whand contribute to this goal:

Explicit names: Syntax, operators and functions are relatively intuitive;

In Whand, **a single operator**, '*when*', underlies all instructions. Each object in the script responds to a set of logical and temporal *conditions* derived from other objects. Each instruction is executed at the specified time.

A Whand script is composed of **parallel parts** that are largely independent. Objects may be defined in any order. Parts of a script may be easily re-used in another script;

Whand easily handles **delays** ('event+delay', 'delay *since* event').

Whand directly **describes the behavior** of each component of the system rather than the *procedure* needed to produce this behavior. Typically, the behavior of a component, say the reward dispenser, depends on other components, say a lever that is pressed. Whand makes this **relationship entirely explicit**, e.g. 'reward *when* lever_pressed'. Both 'reward' and 'lever_pressed' are *events*, meaning that they can take the values '*true*' (ON) or '*false*' (OFF) at various times (see Figure 1).

Whand encourages **a top-down approach**. Essential processes can be spelled out before delving into the details of implementation. Details can be redesigned separately without altering the general function of the script;

According to the top-down approach, 'reward *when* lever_pressed' defines the main function to be performed in the script. The names 'lever_pressed' and 'reward' are arbitrary names, not keywords and they could mean anything. The details of what exactly is a 'reward' and what counts as 'lever_pressed' is deferred to another part of the script.

However, 'lever_pressed' and 'reward' *can* be directly linked to **physical inputs and outputs** (actual press of a lever and delivery of a reward) using a very simple syntax:

```
# Linking 'lever_pressed' and 'reward' to an input and an output
lever_pressed: pin(3)              # equivalence between 'lever_pressed' and 'pin(3)'
output(2): reward                  # equivalence between 'output(2)' and 'reward'
```

In the equivalence indicated by a colon ':', the term on the left, 'lever_pressed', continuously tracks the value of the term on the right, 'pin(3)', a physical input that depends on the hardware. Similarly, the physical output 'output(2)' will be continuously controlled by the value of 'reward'.

Object definition and the top-down approach

In the example 'reward *when* lever_pressed', the behavior of the reward dispenser is not properly specified. We do not want the reward dispenser to remain on forever after the first lever press. This would preclude further reward deliveries or, worse, it could deliver rewards continuously.

Adding a new condition '*until* reward+dispenser_time' will limit the duration of 'reward' by switching 'reward' back to '*false*' a certain time after the onset of the 'reward' event (the '*until*' operator is just another form of the '*when*' operator). Here, 'dispenser_time' is an arbitrary name, defined elsewhere as a duration, e.g. 'dispenser_time: 500ms'.

```
# Definition of 'reward'
reward when lever_pressed                  # activate 'reward'
   until reward+dispenser_time             # terminate 'reward' after 'dispenser_time'
# Definition of 'dispenser_time'
 dispenser_time: 500ms
```

The '*until*' instruction is important because the '*when*' instruction does not establish an equivalence between 'reward' and 'lever_pressed'. A '*when*' or '*until*' instruction is only triggered when its condition becomes '*true*'. Although 'reward' will change to '*true*' when the lever is pressed (condition 'lever_pressed' becomes '*true*'), 'reward' will not return to '*false*' when the lever is released (condition becomes '*false*'). Instead, 'reward' will stay '*true*' as long as the '*until*' condition is not realized.

Together '*when* lever_pressed' and '*until* reward+dispenser_time' constitute the *definition* of object 'reward'.

A Whand script is nothing but a set of object definitions. Because all objects work in parallel, object definitions may appear in any order.

An object cannot have more than one definition. However, the definition may contain multiple expressions and conditions, in no particular order. This allows one to create elaborate scripts when necessary.

The behavior of each object is entirely specified in its definition. There is no way for an object to alter the behavior of another object, unless this dependency is explicitly specified in the other object's definition. This greatly facilitates the debugging of a script.

According to the top-down approach, 'lever_pressed' and 'reward' need not directly correspond to physical inputs and outputs. For instance, a valid lever press could be restricted to an active period.

```
# Definition of 'lever_pressed' requiring the lever to be active
lever_pressed when begin press and lever_active
   until lever_pressed+epsilon
# Definition of 'press' as equivalent to a physical input
press: pin(3)
```

In this example, the 'lever_pressed' event has *epsilon* duration (i.e. extremely brief), and only occurs during a 'lever_active' period. Object 'press', instead of 'lever_pressed', is linked to a physical input using the equivalence "press: *pin*(3)".

Object 'lever_active' also needs a definition of its own, etc.

Conditions based on **'*when*' and '*until*'** always apply to *events*, i.e. objects taking the values '*true*' or '*false*' at various times (Figure 1). The *onset* of the condition (i.e. when the condition switches to '*true*') serves as a trigger to **assign** the *value* computed on the right side of *when*' or '*until*' to the object defined on the left side. The object name appears only once:

```
object_name when some_event: some_value
            when some_other_event: some_other_value
```

An object keeps its value until some condition in its definition is realized.

When defining an event, the value term is usually omitted. It is understood as '*true*' in a '*when*' condition and as '*false*' in an '*until*' condition.

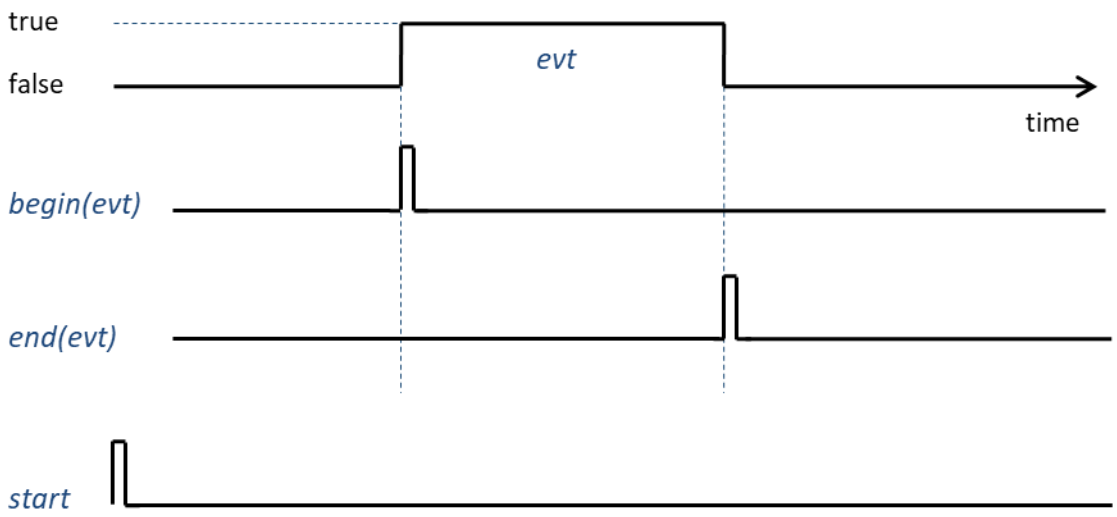


Figure 1. An event can take values '*true*' (ON) or '*false*' (OFF) across time. Function '*begin*' generates a very brief event at event onset. Function '*end*' generates a very brief event at event offset. Function '*start*' generates a very brief event when script execution begins.

The Skinner box

A possible script for the Skinner box is shown below. Keywords shown in *italics.* Comments are preceded by '#'.

```
# Parameters
output(3): reward                                  # select reward dispenser here
output(5): present_lever                           # select lever to present here
press: pin(4)                                      # select lever to press here
max_rewards: 20                                    # equivalence for a number
max_session: 15min                                 # equivalence for a duration
eating_delay: 5s                                   # time at the end for dispenser and consumption
dispenser_time: 500ms                              # standard pulse duration for dispenser
# exit condition
exit when start+max_session                        # time limit
   when (count(reward)=max_rewards)+eating_delay       # reward number limit
# Procedure
present_lever                                      # remains ON for the whole session
reward when press and count(reward)<max_rewards    # definition of reward
   until reward+dispenser_time                     # pulse duration
```

As can be seen, the syntax of Whand is parsimonious and largely intuitive.

At the top of the script there are equivalences to **name each parameter**. It is recommended, as in other programming languages. Thus, for instance, you only need to change line 5 to replace 'max_rewards: 20' with 'max_rewards: 30'. You don't need to search the script for a place where the value '20' is used.

Object '*exit*' terminates the execution of the script when '*true*'. The definition of '*exit*' includes two conditions. Whatever condition is realized first will terminate execution.

The first condition is a time limit ('*start*' is briefly '*true*' at the beginning of script execution). The **delayed event** '*start*+max_session' will terminate execution after time 'max_session' has elapsed (Figure 2).

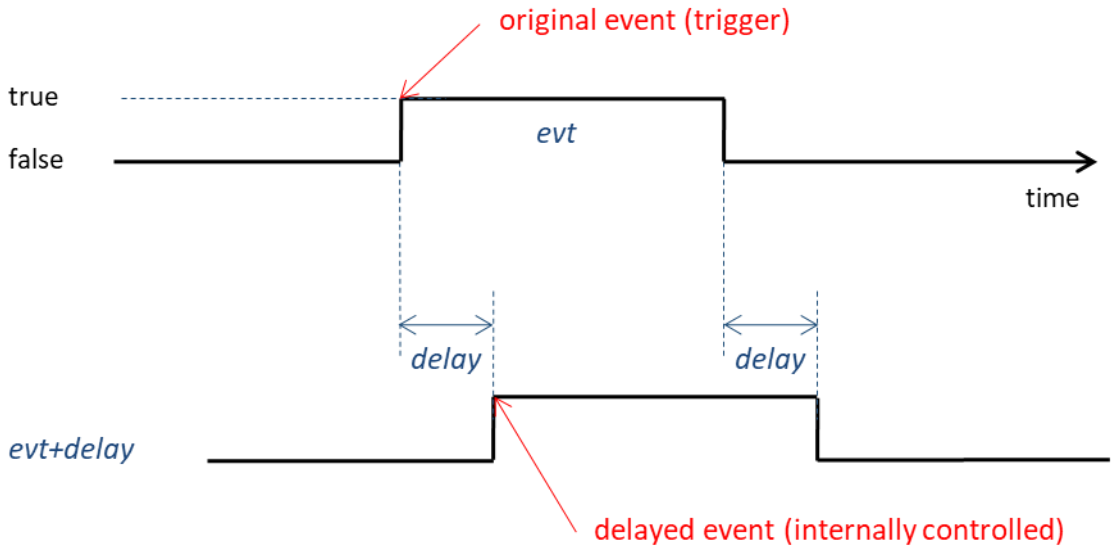


Figure 2. A delayed event is created by adding a delay to an event (*evt+delay*). It becomes '*true*' or '*false*' a certain time (*delay*) after the original event becomes '*true*' (onset) or '*false*' (offset), respectively. The delayed event has the same duration as the original event. Delayed events are internally controlled and cannot be cancelled or relocated in time after the onset of the original event.

The second condition limits the number of rewards (‘*count*’ is a predefined function). ‘*count*(reward)=max_rewards’ compares two numbers and gives an event (‘*true*’ or ‘*false*’). Because we want to leave some time to deliver and consume the reward, the event has to be delayed as in ‘(*count*(reward)=max_rewards)+eating_delay’.

Instruction ‘present_lever’ means ‘present_lever *when start*: *true*’. It defines object ‘present_lever’ as always ‘*true*’. This object is linked to physical output 5 at the top of the script.

In the definition of reward, we also added ‘*and count*(reward)<max_rewards’ to ensure that no reward is delivered after the reward count reaches ‘max_reward’ (i.e. during the eating delay).

‘reward+dispenser_time’ is another delayed event that becomes ‘*true*’ or ‘*false*’ exactly ‘dispenser_time’ after ‘reward’ becomes ‘*true*’ (onset) or ‘*false*’ (offset), respectively (Figure 2).

The cat flap

Suppose you want to control an automated cat flap door with a movement detector, so that the flap opens when a movement is detected. Of course, we do not want the flap to close on the cat, but only after the cat is either in or out. This means that we cannot set a fixed open time for the flap.

In this case a delayed event is not appropriate, we need to use **the ‘*since*’ operator**.

```
# Parameters
output(1): open                          # flap opens when true
movement: pin(1)                         # movement detector
safe_time: 3s                            # time after the last movement
# Procedure
open when movement
   until safe_time since movement        # do not use ‘movement+safe_time’ here
```

The flap only closes when there has been no movement in the last ‘safe-time’. The delay in ‘*since* movement’ starts at the *end* of ‘movement’ (unless ‘*since begin* movement’ is specified instead).

Here ‘safe_time *since* movement’ is more appropriate than ‘movement+safe_time’ because it resets the delay if any movement occurs during the ‘safe_time’ (Figure 3). On the contrary, with ‘movement+safe_time’, the flap might close on the cat because delayed events cannot be cancelled. Each movement would trigger a delayed event so that the flap would repeatedly close and open.

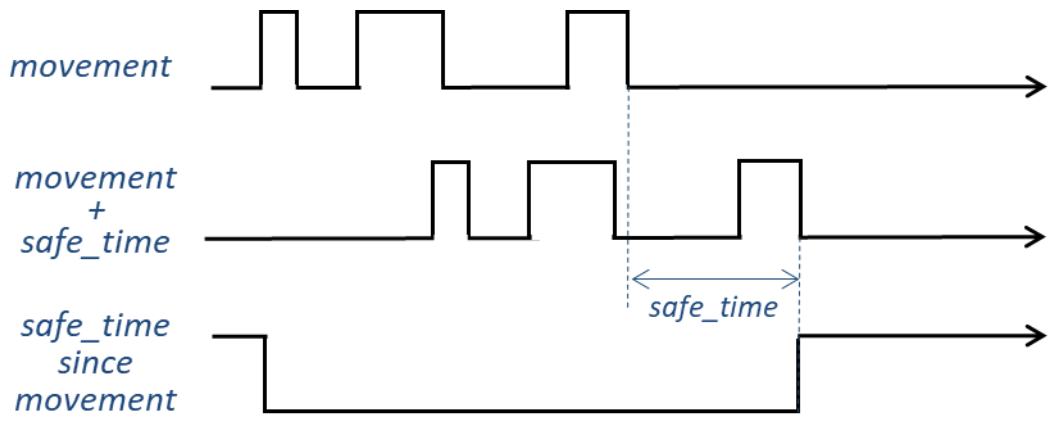


Figure 3. The ‘*since*’ operator is reset at each onset of the event (here ‘movement’), and it has an indefinite duration. One may also combine ‘*since*’ conditions, e.g. ‘*when* 1mn *since open and* 4s *since begin* movement’.

Because delayed events cannot be cancelled, a "**boomerang effect**" may occur in circumstances where a delayed event is triggered but not actually used because some other condition is satisfied before. The delayed event may then occur at an inappropriate moment. This will not happen with a '*since*' operator because it resets the delay.

The dog feeder

The following examples demonstrate how to count events within a certain interval and how to generate a periodic event.

Suppose that you want to control how much food your dog gets each day. You could set up an automatic dispenser with a lever, as in the Skinner box with rewards (here a piece of kibble), and when enough rewards have been obtained, you inactivate the lever.

Instead of resetting a count at intervals, you may use the function '*count*', which is never reset, and subtract its value at the start of the interval from its value at the end of the interval. In addition, to define the interval, you need a pulse that re-occurs every day.

The script very similar to that of the Skinner box. Instead of 'max_reward', we now have 'last_count+max_reward'. Rewards stop after we reach 'last_count+max_reward'.

```
# Parameters
max_rewards: 50                    # during each interval
output(3): reward                  # select reward dispenser here
press: pin(4)                      # select lever to press here
dispenser_time: 500ms              # standard pulse duration for dispenser
```

```
# Procedure
reward when press and (count(reward)< last_count+max_rewards
   until reward+dispenser_time
last_count when start: 0           # definition for last_count
   when resetting: count reward    # store value of counter
resetting when start               # create a resetting pulse every day
   until resetting+epsilon         # stop pulse (very brief)
   when 24h since begin resetting  # auto-restart
```

The value of 'last_count' is a copy of the total number of rewards '*count* reward' made on each resetting pulse. This value remains frozen until the next resetting pulse.

The resetting pulse is switched on at *start* (or any other time, e.g. '*start*+6h') and stopped almost immediately (*epsilon* duration). However, the delayed event '24h *since begin* resetting' starts it again the next day at exactly the same hour. The expression '24h *since begin* resetting' is also reset as soon as resetting becomes '*true*', so it repeats itself every day.

You could add a button to adjust resetting time. For instance, with the additional condition '*when* button', the press on the button will set the hour and resetting will occur every day on that same hour. With '24h *since begin* resetting', resetting can only occur once every 24h.

On the other hand, with a delayed event such as '*when* resetting+24h', you can have several resetting events occurring in parallel every day at fixed hours (so your dog may eat twice a day).

You can use **absolute time** to reset the counter every day at 4h30 am and 6 pm (Figure 4):

```
resetting when time is 4h30mn      # reset at 4h30 am
   when time is 18h                # reset at 6 pm
   until resetting+epsilon
   when resetting+24h              # twice every day
```

Note that function '*time is*' just creates a single event after the start of execution, on the same day or the day after. The bulk of resetting events is generated by '*when* resetting+24h'.

There are other absolute time functions ('*day is*', '*week is*', '*date is*') and they can be combined, e.g. '*when day is tuesday and time is* 14h'.

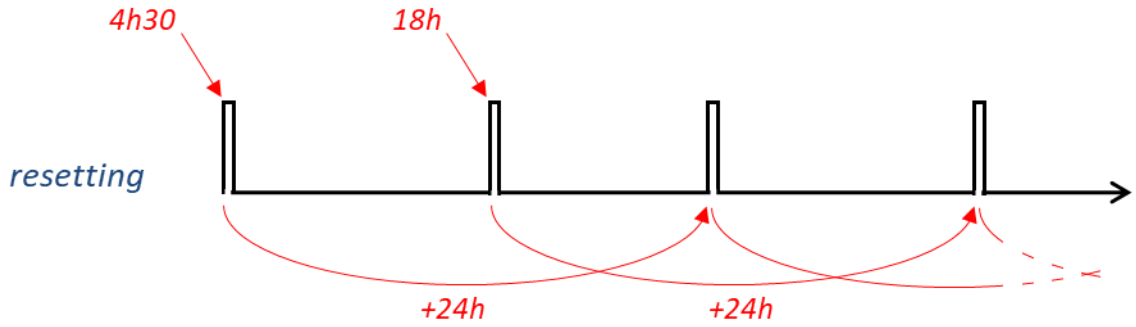


Figure 4. Using delayed events to set up parallel series of periodic events.

Resetting counter

In the previous example, another way to count rewards in the interval is a resetting counter.

```
# Procedure
reward when press and counter<max_rewards
   until reward+dispenser_time

counter when start: 0                    # definition for counter
   when resetting+epsilon: 0             # the counter is reset after 'resetting' event
   when reward: old+1                    # increment counter on each reward

resetting when start                     # create a resetting pulse every day
   until resetting+epsilon               # stop pulse (very brief)
   when button                           # use button to synchronize (e.g. 'button: pin 6')
   when 24h since begin resetting        # auto-restart
```

'*when* reward: *old*+1': **Function '*old*'** *must* be used if the value of an object (here 'counter') is modified with respect to its previous value (e.g. incremented). '*old*' is compulsory in order to prevent an infinite updating loop. Within the definition of an object, '*old*' designates the previous value of the object, i.e. its value at the current time step *before any modification* occurs. Thus '*old*(counter)' will not be affected by the change in value and 'counter' will only be incremented by one.

Periodic events can be useful in various situations. For instance, you may want to create short bursts of regularly spaced pulses.

```
burst when trigger                       # event to start each burst
   until trigger+burst_duration          # total duration of each burst

pulse when burst
   when pulse+pulse_period and burst     # restart pulse if burst is not terminated
   until pulse+pulse_duration            # stop pulse after pulse duration
   until end burst                       # stop pulse if burst terminates
```

Alternatively, you can use a resetting counter to define the number of pulses. The definition of a pulse does not change.

```
counter when start: 0                    # resetting counter
   when burst: 0                         # reset at the beginning of each burst
   when end pulse: old+1                 # increment counter after each pulse
burst when trigger                       # event to start each burst
   until counter=pulse_number            # number of pulses in each burst
pulse when burst
   when pulse+pulse_period and burst     # restart pulse if burst is not terminated
   until pulse+pulse_duration            # stop pulse after pulse duration
   until end burst                       # stop pulse if burst terminates
```

The above examples show how parts of various scripts can be recombined and/or altered as you wish. Event names link the different parts together.

Moreover, Whand allows you to **define your own functions** and to use them in several places in a script (with the same or different arguments).

```
# Example of how to create and use a resetting counter function
resetting_counter(evt', raz') when start: 0      # prime (') indicates virtual function arguments
   when raz'+epsilon: 0                          # reset upon raz'
   when evt': old+1                              # increment upon evt'

burst when trigger                               # event to start each burst
   until resetting_counter(end pulse, burst)=pulse_number      # use resetting counter function
                                                 #                  with real arguments
pulse when burst
   when pulse+pulse_period and burst             # restart pulse if burst is not terminated
   until pulse+pulse_duration                    # stop pulse after pulse duration
   until end burst                               # stop pulse if burst terminates
```

The syntax of a function definition is essentially the same as that of any other object. The only difference is that the function has arguments between parentheses. The virtual arguments, denoted by a prime after their names have no actual existence. They serve as placeholders in a template.

When the function is called with real arguments (i.e. objects defined or used in the rest of the script), a definition of the function-of-these-real-arguments is created internally using the function template. Each call of the function with different arguments defines a different object with the same template. This is possible because the function behaves as a *list* of calls where each *element* gets a separate definition (see below).

Lists

Whand does not provide any loop structure to perform repetitive operations on multiple objects. Instead, Whand provides lists and list functions.

In its simplest form, a list is just a series of object names separated by commas, e.g. 'x: first, middle, last'. A list can be addressed by a numerical index, e.g. x(2) is 'middle', x(-1) is 'last', and x(4) is the same as x(1), i.e. 'first'.

Function '*next*' may be used to extract list elements one by one. So at *start*, *next*(x) is 'first' and on each subsequent *assign*, '*next(x)*' will take values 'middle', 'last', then 'first' again, etc.

A list can also be defined element by element. Each element is identified by the name of the list and an index which may be an object of any category. For instance, the two definitions 'color(jacket):

red' and 'color(shoes): yellow' define a list called 'color' with two elements, 'color(jacket)' and 'color(shoes)'.

Distributivity is a particularly useful property of lists. It is the ability to combine a list with any type of operation. For instance, if list 'z' is '2,3,4,5', then 'z*5mn' is '10mn, 15mn, 20mn, 25mn'.

In addition, various list functions are available to manipulate lists.

Variable ratio

Whand was initially designed for animal learning experiments. In this type of experiments, a subject is often presented with a variable ratio schedule (VR). It means that in order to get a reward, the subject must perform the target response a variable number of times. An advantage of this schedule is to support high rates of responding while limiting the actual number of rewards provided.

For instance, in a VR-10 schedule, the first reward may be obtained after 8 responses, the second after 12 more responses, then maybe 6, 10, 16, 7, 11 etc. The number of responses needed to obtain a reward is called a *ratio*. The ratios given here (mean ratio = 10) form a pseudo-random sequence that can be specified as a list in Whand. One can implement a VR schedule using a resetting counter.

```
# VR-10 schedule
include "tools\resetting counter.txt"
reward when resetting_counter(response, reward)=ratio
   until reward+dispenser_time                  # pulse duration
ratio when start: next ratio_list               # get first ratio from list
   when end reward: next ratio_list             # change ratio after a reward
ratio_list: 8, 12, 6, 10, 16, 7, 11             # etc.
```

Of course, 'response', 'reward' and 'dispenser_time' must be defined somewhere else in the script.

A **separate text file** located in sub-directory 'tools' is used here to provide the definition of the resetting counter function (explained earlier). The instruction '*include* "tools\resetting counter.txt"' simply inserts the content of the file in the current script. This is a convenient way to re-use the same piece of text in multiple scripts.

To **read from file** the ratio list, use instruction '*load*', e.g. 'ratio_list: *load* "ratios.txt"'. Here, "ratios.txt" should be a text file containing a single value on each line. Unless otherwise specified (with a '*when*' instruction), the file is read at *start*.

The power of lists

A number of **list functions** are available. They allow powerful manipulations of lists.

Function '*count*' gives the number of elements in a list, in addition to counting events.

Function '*ramp*' generates a list of consecutive integers, starting at 1, e.g. 'ramp(6)' is 1,2,3,4,5,6.

Function '*cumul*', applied to a list of numbers or durations, computes a cumulative list where the first element is the first element of the list, the second element is the sum of elements 1 and 2, etc. So the last element of list 'L' '*cumul*(L)(-1)' is the sum of all elements, and '*cumul*(L)(-1)/*count*(L)' is the mean of the list.

```
# programming a variable interval VR-10 schedule with list functions
reward when count(response) is in cumul(ratio_list)
   until reward+dispenser_time                  # pulse duration
ratio_list: 8, 12, 6, 10, 16, 7, 11             # etc. (full list must be given for cumul)
```

Here is a way to program a burst of 'pulse_number' pulses with list functions:

```
# Burst of a fixed number of pulses
burst when trigger                                  # event to start each burst
   until trigger+epsilon                            # use very brief trigger
pulse when any(burst+1s*(ramp(pulse_number)-1))  # multiple pulses
   until pulse+pulse_duration                       # stop pulse after pulse duration
```

The expression can be read from right to left: '*ramp*(pulse_number)' is the list of numbers 1, 2… to pulse_number. '*ramp*(pulse_number)-1' uses distributivity to subtract 1 from each number so the list now starts at zero. It is then multiplied by 1s, yielding the list of delays 0s, 1s, 2s…

Adding the list of delays to event 'burst' yields the list of delayed events: burst+0s, burst+1s, burst+2s… Function '*any*' will become '*true*' every time one of these delayed events occurs. Function '*any*' is required because the '*when*' condition needs a single event, not a list.

The same principle can be used to trigger an event at irregular intervals, based on an arbitrary list.

Of course, some practice is needed to exploit the full potential of list functions.

For instance, 'color(jacket): red' and 'color(shoes): yellow' define a list called 'color' with two elements, 'color(jacket)' and 'color(shoes)'.

Moreover, '*has*(color)' retrieves the list of indices of list 'color', i.e. 'jacket, shoes', and '*has*(color) *pick* (color=yellow)' yields 'shoes,' (the comma indicates this is a list and not just an element).

Specifically, 'color=yellow' uses distributivity to compute 'color(jacket)=yellow' and 'color(shoes)=yellow', yielding the list '*false*, *true*', where only the second element is '*true*'. This list is then used to *pick* a sublist of 'jacket, shoes' composed of the second element only, i.e. 'shoes,'. To convert this list into its first (and only) element, it may be subscripted with (1). Thus '(*has*(color) *pick* (color=yellow))(1)' will extract the name 'shoes'.

Timing intervals

With delayed events and '*since*', there is not much need for timing. However, some functions may be useful to compute time intervals when the end of the interval is marked by an event:

Function '*inter*', e.g. '*inter*(press)' gives the time between the last two occurrences (onsets) of an event, here 'press'.

Function '*to*', e.g. '*start to* press' gives the time between two events, here '*start*' and 'press'. '*begin* x *to end* x' gives the duration of event 'x'.

Function '*time*' (different from '*time is*') gives the hour of onset of an event. So '*time*(press)-*time*(*start*)' is equivalent to '*start to* press'.

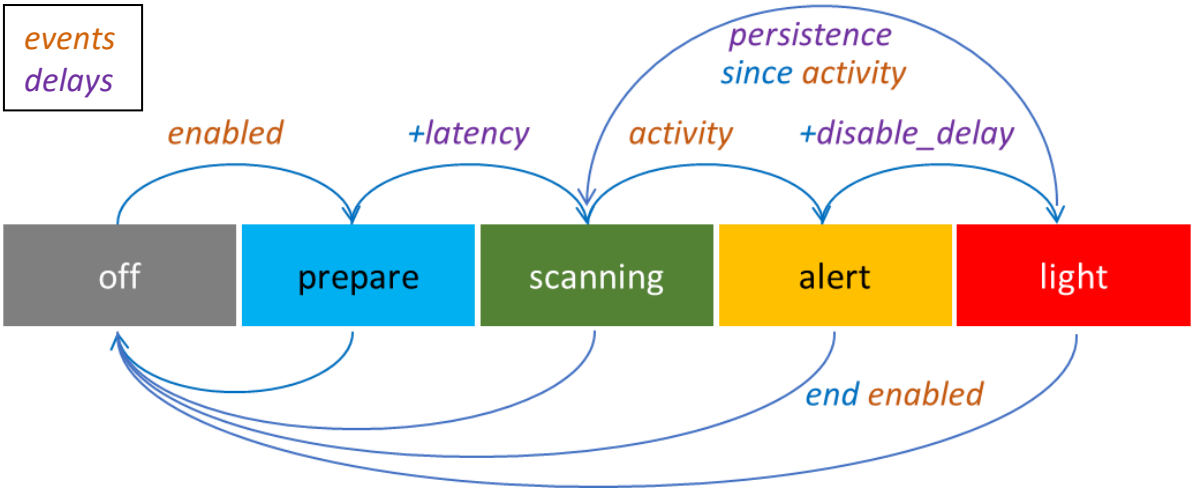
Function '*lasted*' gives the duration since event onset, provided the event is still on. It can be used to follow the time course of an ongoing event.

More timing

Let us now consider a more difficult case. Suppose we want to set up at home an alarm with a floodlight triggered by activity in a given perimeter. We want to be allowed time to leave and also time to switch off the alarm when we return. The light should remain on while there is activity, but not last too long after activity ceases. Programming this with a sequential logic is not so easy.

Here is one way to program the alarm in Whand. *Reading* this script may seem rather intuitive but *writing* it is challenging. Behind the apparently natural syntax, one needs a rigorous understanding of the interplay of events.

Diagram of states:




```
# Parameters
output(1): floodlight                    # connect to hardware (e.g. floodlight)
activity: pin(1)                         # connect to hardware (detector)
enabled: pin(2)                          # connect to hardware (e.g. code)
latency: 1mn                             # time window when starting
disable_delay: 20s                       # time window before starting light
persistence: 40s                         # after activity has stopped
```

```
# Procedure
floodlight: period is light
period when start: off                   # no alarm
   when enabled: prepare                 # ignore activity for a short while
   when begin(period is prepare)+latency: scanning     # now detect activity
   when (period is scanning) and begin activity: alert    # do not start alarm at once
   when enabled and begin(period is alert)+disable_delay: light    # now start alarm
   when (period is light) and persistence since activity: scanning    # stop some time after activity
   when end enabled: off                 # no alarm if disabled
```

Object natures:

'floodlight', 'activity' and 'enabled' are *events*. 'latency', 'disable_delay' and 'persistence' are *delays*.

The first script makes use of *states*, a category of objects distinct from *events*, *numbers* or *delays*. 'period' is such a state and it has a definition so it can change value. A state may take just one value at any instant. 'off', 'prepare', 'scanning', 'alert', and 'light' are such values. These state names are arbitrary and do not require any definition.

The value of a state may be tested using '*is*' or '*is not*'. The result of this comparison is '*true*' or '*false*', i.e. it is an *event*. Adding a time delay to this event makes it a delayed event.

Procedure:

'scanning' for activity starts when the delayed event '*begin*(period *is* prepare)+latency' becomes '*true*', i.e. exactly 'latency' after the alarm is 'enabled' (latency has been defined as a *delay* of 1 min). '*begin* (period *is* prepare)' is a very brief event occurring at the onset of 'period *is* prepare'.

'alert' means activity has been detected, but light will not be immediately turned on. The formula '(period *is* scanning) *and begin* activity' means that '*begin* activity' must occur during the 'scanning'

period, and not before. Here, one must avoid the formula '(period *is* scanning) *and* activity' where '*and*' is symmetrical. This formula could become '*true*' even if activity has begun before 'scanning' (such as when we are leaving the perimeter).

'light' occurs exactly 'disable_delay' after '*begin*(period *is* alert)' provided the alarm is still 'enabled'. This leaves us time to disable the alarm when we return.

After the end of 'activity', 'light' remains on ('*true*') for a duration equal to 'persistence', then returns to 'scanning'. This is the meaning of 'persistence *since* activity'. The '*since*' event remains '*false*' during 'activity'. It counts time from the end of 'activity', but it is cancelled and restarted if 'activity' reoccurs. After the delay, '*since*' becomes '*true*' for an indefinite duration.

Finally, 'light' and the other states are turned 'off' as soon as we disable the alarm. '*end* enabled' is a very brief event occurring at the offset of 'enabled'.

Alternate procedure:

Below is a slightly more compact script to perform the same function with the same parameters. It does not use states, but properties of the '*since*' operator.

```
# Procedure
scanning when enabled and latency since begin enabled    # delayed detection
   until end enabled                                     # stop detection
floodlight when scanning and begin activity+disable_delay    # delayed alarm
   until persistence since activity                      # stop alarm some time after activity
   until end scanning                                    # cancel alarm
```

Here, 'scanning' is an *event*, not a *state*. It starts after a delay equal to 'latency' and stops when 'enabled' is switched off. The condition '*when* enabled *and …*' ensures that 'enabled' has not been switched off during the delay.

'latency *since begin* enabled' is preferred to '*begin* enabled+ latency' because it resets the delay if you switch 'enabled' on, off then on again (e.g. to pick up your keys). By contrast, '*begin* enabled+ latency' will not be cancelled (and may catch you on your way out).

Working with different objects

As we have seen, objects in Whand have five possible natures:

- *events* that can take values '*true*' or '*false*' (e.g. 'reward')
- *numbers* that can take integer or floating-point values, positive or negative (e.g. '20')
- *delays* that are measured in ms, s, min or mn, h, day or wk (e.g. '500ms', 'dispenser_time')
- *states* that take textual values (e.g. '*january*')
- *lists* that can contain values of any of the five categories (e.g. '(4s, 5mn, (1, 2, 3), ready)')

The danger of transitions

A general rule in Whand, as in other types of parallel programming, is to avoid using the value of an object at the moment it is updated. Updating of objects always occurs according to some internally defined sequence, so it is often difficult to know how the old and the new value of an object may enter the calculations.

Whand provides at least two tools to circumvent this difficulty:

1) Thus, a proper way to increment a counter for 'reward' is

```
counter when reward: old+1
        when start: 0                    # initialize counter (order of conditions is irrelevant)
```

2) Duration '*epsilon*' is negligible in terms of physical effects, but it can be used to delay an operation until the operand has been updated, e.g. "*when* ((x=y) + *epsilon*) *and* (table(x)=5)"

In this example, it is not appropriate to use the value of table(x) before it has been properly updated. At the instant where x becomes equal to y and the '*and*' expression is computed, table(x) may still have its old value, and the condition might be wrongly triggered unless '*epsilon*' is used.

At time '(x=y) + *epsilon*', one can be sure that table(x) has been properly updated so there is no risk of error.