UNIVERSITY OF BARI ALDO MORO

DEPARTMENT OF INFORMATION TECHNOLOGY

Master's Degree in Computer Science

Fundamentals of

Artificial Intelligence

Case study

GreMaES

a Prolog-based Greenhouse Management Expert System

Francesco Peragine

f.peragine@studenti.uniba.it

m. 737873

https://github.com/francescoperagine/GreMaES

A.Y. 2021/22

Sommario

Introduction ... 3
Interacting ... 3
Consulting ... 3
Monitoring ... 3
Reasoning ... 4
Requirements ... 4
Domain ... 5
Ontology ... 5
Conceptualization ... 6
Analysis ... 6
Predicates ... 7
Rules ... 9
Overview ... 9
User mode ... 11
Knowledge Base mode ... 18
Monitor mode ... 19
Logger ... 31
Lists, utilities ... 32
Knowledge Base ... 35
Plant domain ... 35
Nutrient deficiencies ... 35
Pests ... 38
Diseases ... 41
Conclusions ... 46
Future developments ... 46

Introduction

The present case study illustrates the development details of a Prolog-based *Expert System* (ES) within the domain of the ambient intelligence, in the context of the management of a greenhouse.

The plant domain has been inevitably oversimplified to be addressed in both size and complexity, and several assumptions also take place, like those relative to light (intensity, duration, spectrum), water (composition, hardness, pH) and so on.

The ES can be use in three ways: **interacting** by submitting queries, **consulting** the knowledge base and by **monitoring** the greenhouse itself.

Interacting

The user is guided by an interactive shell interface to submit the symptoms that are manifested on a plant, to obtain a diagnosis of its health problems. The user may provide any number of captions.

For the sake of explainability, every diagnosis is provided with the set of rules that led to the actual results.

Consulting

The user can fully explore every information stored in the knowledge base.

Monitoring

To ensure safety and the correct lifecycle of hosted plants, the agent carries out the following actions:

- continuous monitoring by the means of sensor devices
- storing environment readings
- identification of the health statuses and conditions that may affect the plants based on stored environment percepts
- controlling the environment thanks to actuator devices
- showing notifications to the user through the Command Line Interface
- log all details in a text file.

Sensors devices provides the inputs: readings for temperature and humidity, image captions for textual descriptions of any strange manifestation occurred on the plant. All readings are simulated through simple random sampling strategy performed with timed cadence. At the end of every batch the user is asked if he wishes to continue with the sampling.

Problems that have been identified and that can be directly addressed through actuator devices are automatically fixed, like a low humidity level can be fixed activating the watering system.

At the end of the process, a detailed report is shown.

Debugging

A debug mode has been introduced to speed up the testing process: it interactively asked to the user to activate it and, if so, the sampling process is skipped and a set of readings is loaded from the *percepts_samples* file.

Reasoning

The implementation of the ES is based on backward chaining and hybrid chaining.

The choice has been initially made because of my curiosity towards the backward chaining, for my own readability reasons and to deepen my understanding of this inference engine. The implementation simplicity with respect to the forward chaining would have been a free plus.

I would probably make a different choice now, as has been increasingly difficult to handle the project while adding more modalities of use.

Requirements

The software was developed with YAP (Yet Another Prolog) v.6.2.2.

Some SWI Prolog libraries were also included.

Domain

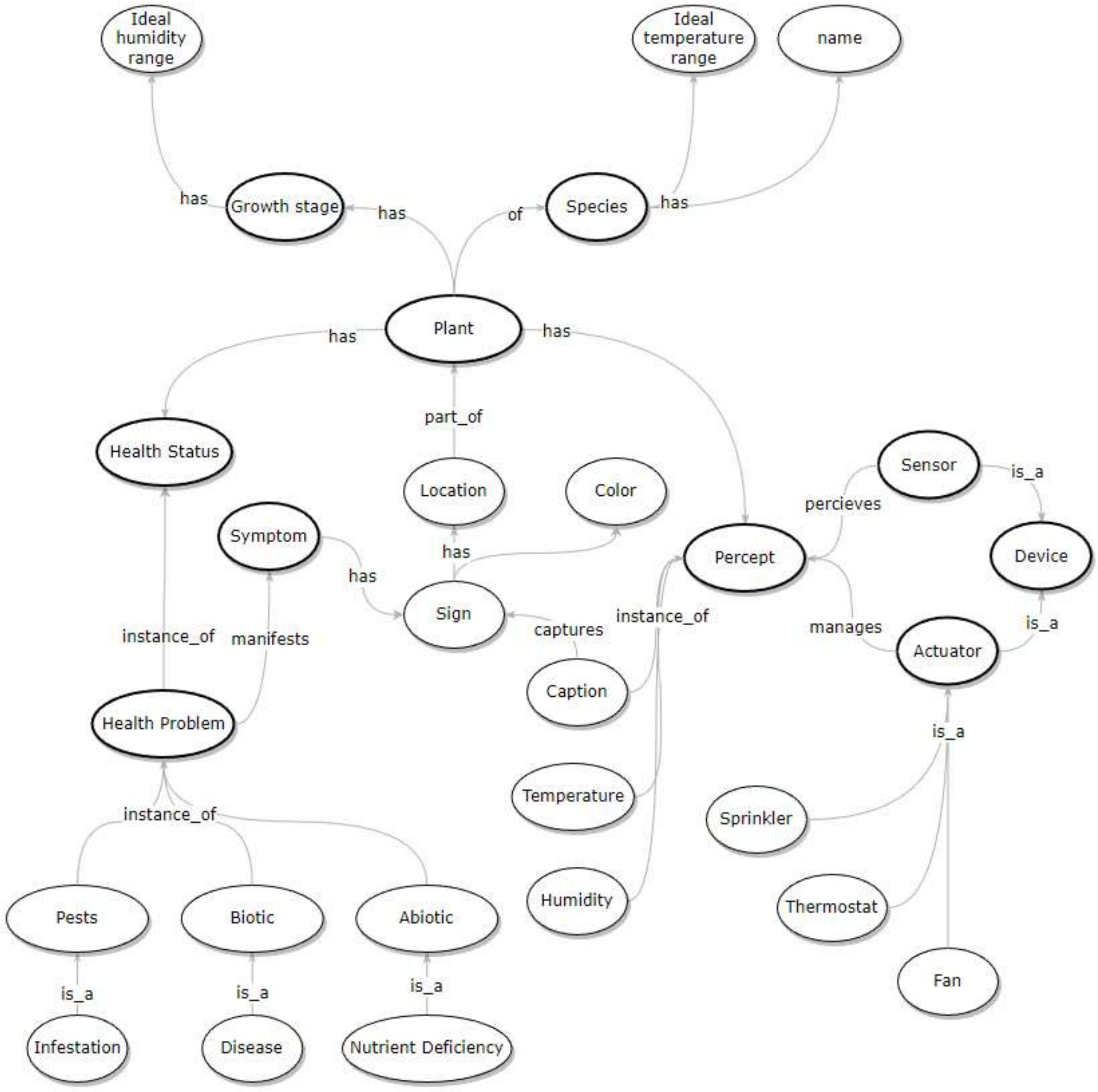
Health problems are families of issues that may affects the plants. There are three main categories:

- **Abiotic disorders**, caused by nonliving factors, like freeze, nutrient deficiencies, overwatering, ...
- **Biotic diseases** caused by living organisms
- **Pests' infestations**

Every health problem has several unique condition *types,* and every type has one or more *symptoms* and may have multiple *treatments*, if any.

Symptoms are manifested through *signs* on *sections* of the plants and may show different *colors*.

Ontology



Conceptualization

Domain: Plants

Goal: distinguishing the health conditions of a plant.

Analysis

- Each **plant** is characterized by:
    - name
    - species
    - growth stage
    - health status
    - set of connected sensor devices
    - set of connected actuator devices
- **Species** have:
    - name
    - temperature range
- **Growth stage:**
    - identifier
    - humidity range
- **Sensor devices** perform readings of the environment's properties
    - identifier
    - metric (temperature, humidity)
    - sensor devices
- **Actuator devices** alter the properties of the environment:
    - identifier
    - type
    - activation status
    - class
- **Health problems**, that may affect plants, have:
    - category
    - name
    - set of manifested conditions
- **Condition**
    - set of symptoms

    - set of treatments
- **Symptoms** are physical manifestations of health problems, are denoted by:
    - sign
    - section
    - color

Predicates

- **plant**(X, Y, Z) = plant X is of species Y and has growth stage Z
    - Constants: {trinidad1, sunflower, p1, p2, ...}
- **health_status**(X, Y) = plant X has status Y
    - Constants: {healthy, abiotic, biotic, pests

      {hot, cold, wet, dry, infestation, disease, nutrient_deficiency}
- **species**(S, X, Y) = species S has ideal range of temperature [X - Y]
    - Constants: {Rudbeckia hirta, ...,

      -5, ..., 40}
- **growth_humidity**(S, X, Y) = stage S has ideal range of humidity [X - Y]
    - Constants: {flowering_mature, vegetative_growing, seed_germination,

      ...,

      0 - 100}
- **problem_condition**(X, Y) = problem X is due to condition Y
    - Constants: {healthy, nutrient_deficiency, disease, infestation,

  nitrogen, phosphorus, potassium, sulfur, ...}
- **sign_location**(X, Y) = sign X occurs on location Y
    - Constants: {none, altered_color, angular_lesions, black_leathery_spot, ...

  all, branches, leaves, lower_leaves, roots, ... }
- **sign_color**(X, Y) = sign X has color Y
    - Constants: {altered_color, cotton_like_downy_substance, flies, ...

      chlorotic, dark_green, blotchy_chlorosis, interveinal_chlorosis, ...}
- **symptom**(X, Y, Z) = symptom occurs on section X with sign Y and color Z
- **treatment**(X, Y) = condition X may be solved with treatment Y
    - Constants: {'spray with neem oil and insecticidal soap', 'cut off and remove the infected leaves or flowers', 'shower the plant once a week', ...}
- **sensor**(X, Y) = sensor X reads percept Y
    - Constants: {t11, t12, t13, t14, ...

      temperature, humidity, caption}

- **plant_sensor**(X, Y) = plant X is connected to input device Y
- **actuator**(X, Y, W, Z) = actuator X manages environment property Y with status W and has class Z
    - Constants: {act1, act2, act3, …

      low, normal, high,

      thermostat, fan, sprinkler}
- **plant_actuator**(X, Y) = plant X is connected to actuator device Y
    - Constants: {act1, act2, act3, act4}
- **actuator_status**(X, Y) = actuator X has status Y
    - Constants: {on, off}

Rules

Conditions are characterized by a limited set of symptoms (up to three within the case study), which are matched with the provided inputs, by users or sensors, to diagnose a health problem.

Overview

The program starts by welcoming the user, cleaning up the working memory and asking which mode is to be executed.

The choices represented are between the diagnostic *user mode*, the *knowledge base mode* and the automatic *monitor mode*.

The selection ensures the loading of the respective module and its initialization.

Once the selected task is completed, the user is asked if he wants to run the program again.

start

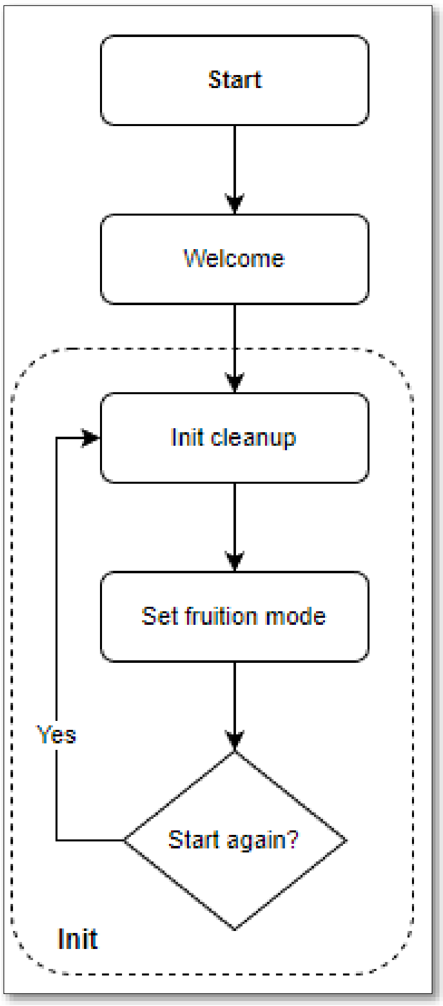
```
start :-
    welcome,
    init.
```

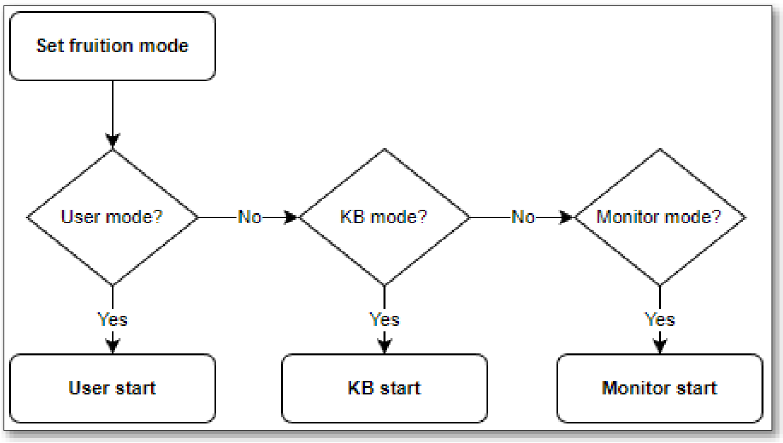
init

```
init :-
    init_cleanup,
    set_fruition_mode,
    restart.
```

init_cleanup

```
init_cleanup :-
    retractall(asked(_,_)),
    retractall(symptom(_,_,_)).
```





set_fruition_mode

```
set_fruition_mode :-
    user_mode,
    ensure_loaded(user_mode),
    user_mode_start.
set_fruition_mode :-
    \+ (user_mode),
    kb_mode,
    ensure_loaded(kb_mode),
    kb_mode_start.
set_fruition_mode :-
    \+ (user_mode),
    \+ (kb_mode),
    monitor_mode,
    ensure_loaded(monitor_mode),
    monitor_mode_start.
```

restart

```
restart :-
    askif(start_again),
    init.
restart :-
    init_cleanup,
    goodbye.
```

user_mode

```
user_mode :- askif(fruition_mode(user_mode)).
```

kb_mode

```
kb_mode :- askif(fruition_mode(kb_mode)).
```

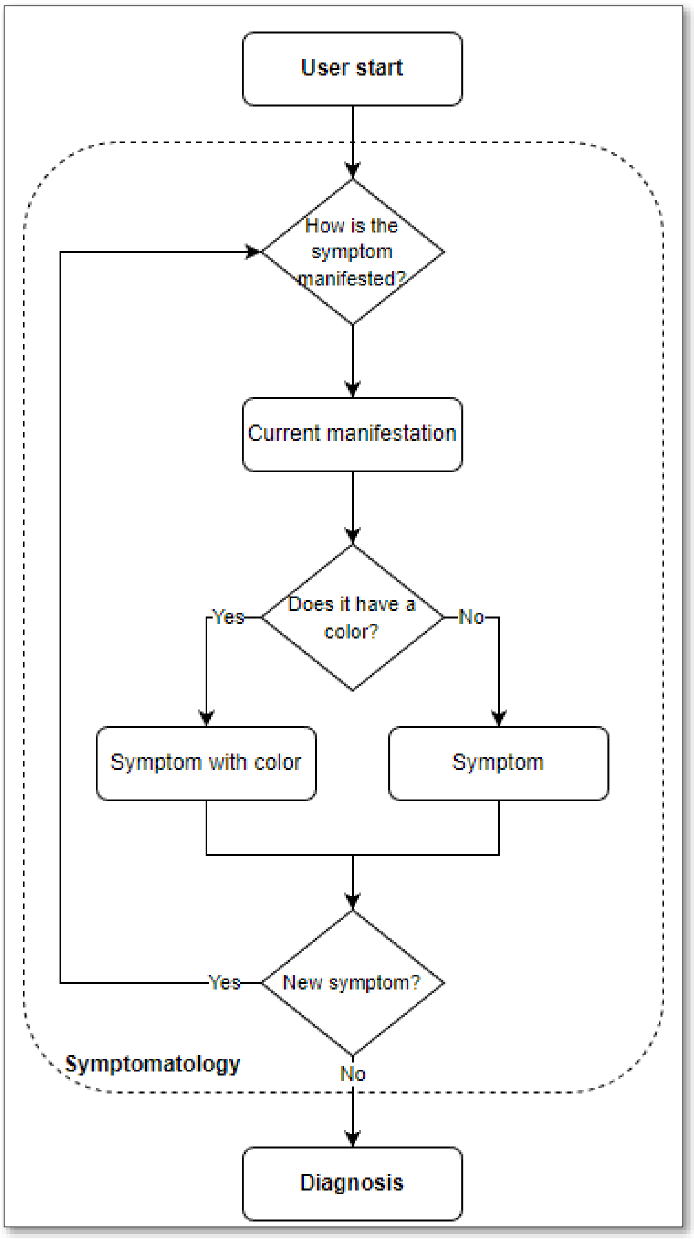
monitor_mode

```
monitor_mode :- askif(fruition_mode(monitor_mode)).
```

User mode

user_mode_start

```
user_mode_start :-
    symptomatology,
    user_diagnosis.
```



The user mode starts by asking how the issue has been manifested and of which color it was, if any.

The **sign** is then temporarily stored to build up the symptom by asking its localization.

To reduce the search and not let the user be overwhelmed by the listing of every possible combination of signs, locations and colors that are to be merged to build every symptom, every choice that is provided to the user will show only the subset of combinations present in real symptoms that manifest health conditions.

Once the symptom has been saved, the temporary information is deleted and the user may keep registering new ones, if any.

The diagnostic process is performed at the end of the gathering phase.

symptomatology

```
symptomatology :-
    repeat,
    ask_symptom,
    \+ (once_again).

once_again :- askif(new_symptom).
```

Asks repeatedly the user to provide symptoms, until the user specifies otherwise.

`ask_symptom`

```
ask_symptom :-
    user_input(signs, Sign),
    assertz(current_sign(Sign)),
    symptomatology_forward,
    symptomatology_cleanup.
```

The user is asked to select the sign shown on the plant. Stores it in the working memory and proceeds with the next steps.

`symptomatology_forward`

```
% If the sign is not associated to a color, sets it to none.
symptomatology_forward :-
    current_sign(Sign),
    \+ sign_color(Sign, _),
    user_input(sign_locations, Location),
    assertz(symptom(Location, Sign, none)).
% If the sign is associated to a color, asks it.
symptomatology_forward :-
    current_sign(Sign),
    sign_color(Sign, _),
    user_input(sign_colors, Color),
    user_input(sign_locations, Location),
    assertz(symptom(Location, Sign, Color)).
```

After the user provided the sign, the KB is consulted to check whether that sign is associated to a color. If so, the user is asked to also provide it.

After that, the user is asked to select the section on which the sign occurs.

`symptomatology_cleanup`

```
symptomatology_cleanup :-
    retractall(asked(_,_)),
    retractall(current_sign(_)).
```

Removes temporary information to prepare to eventually register the next symptom.

`user_input(+Relation, -UserChoice)`

```
user_input(Relation, UserChoice) :-
    call(Relation, Options),
    show_title(Relation), nl,
    show_options(Options),
    read(UserInput),
    input_choice(Options, UserInput, UserChoice),
    write_message(option_selected),
    write(UserInput), write(': '), writeln(UserChoice), nl.
```

Unifies *Relation* with a list of options, reads the user selection from that list and returns it as UserChoice.

sign_location(-Locations)

```
sign_locations(Locations) :-
    current_sign(Sign),
    all(Location, sign_location(Sign, Location), Locations).
```

Unifies *Locations* with the list of all possible locations in which the temporary stored sign can manifest.

sign_colors(-Colors)

```
sign_colors(Colors) :-
    current_sign(Sign),
    all(Color, sign_color(Sign, Color), Colors).
```

Unifies C*olors* with the list of all possible colors that can be manifested on the current selected sign.

show_title(+Relation)

```
show_title(Relation) :-
    writeln_message(Relation).
show_title(Relation) :-
    \+ writeln_message(Relation),
    writeln(Relation).
```

Writes the title of the relation if present within the message_code(Code,Message) facts in *messages.pl* file, otherwise writes reports the relations name.

show_options(+Options)

```
% Shows a numbered list of ordered options, stripping atom names from their
underscores.
show_options(Options) :-
    show_options(Options, 1),
    !.
```

Displays a list of elements from which the user has to choose. The user must provide the option number to perform the selection. The index numbers count starts from the number 1.

show_options(+Options, +N)

```
show_options([], _).
show_options([H|T], N) :-
    atomic_concat([N, '. ', H], A),
    writeln(A),
    N1 is N + 1,
    show_options(T, N1).
```

input_choice(+Options, +UserInput, -UserChoice)

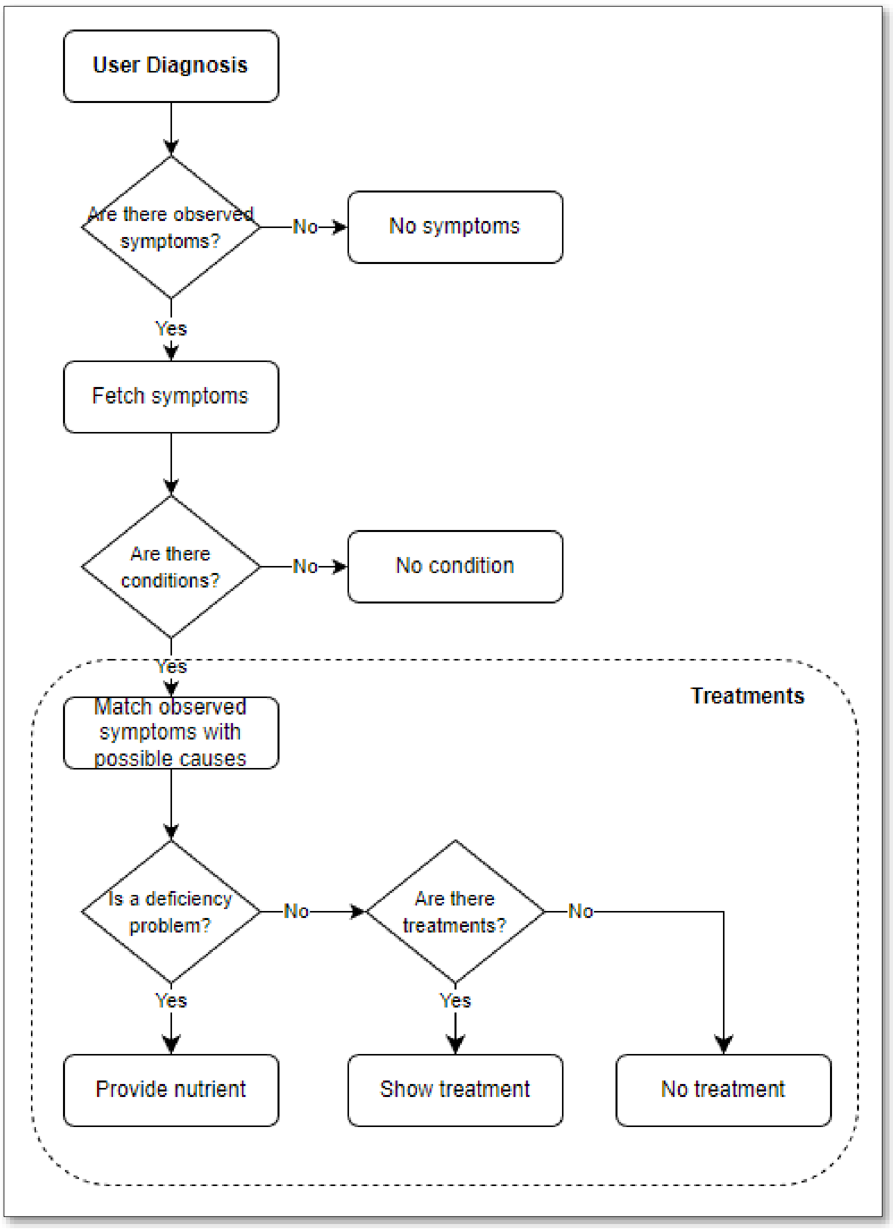
```
% The entry number UserInput of list Options unifies with list entry UserChoice.
input_choice(Options, UserInput, UserChoice) :-
    integer(UserInput),
    nth1(UserInput, Options, UserChoice),
    validate_input(Options, UserChoice).
```

Given the list of options and the option number selected by the user, it verifies that the user provided a valid integer number, retrieves the option from the option list and verifies that the selected option is a valid member of the list.

validate_input(+Options, + UserChoice)

```
validate_input(Options, UserChoice) :-
    member(UserChoice, Options),
    !.
validate_input(Options, UserChoice) :-
    not(member(UserChoice, Options)),
    writeln_message(not_recognized_value).
```

`user_diagnosis`



The diagnosis starts by checking whether there are registered symptoms then it verifies if a condition is present. If so, all the observed symptoms (*ObservedSymptoms*) are collected beforehand (to avoid gathering them every time for every condition) and all the symptoms that can cause the present conditions (*ConditionSymptoms*) are matched against them, so to provide an explanation of the reasoning behind the diagnoses.

Multiple symptoms may not cause any condition, may cause conditions that have no treatments or, most commonly, may lead to conditions that have their own set of treatments to be performed.

In the case of a deficiency problems, the fixed treatment is that of providing the missing nutrients.

```
% No symptoms case
user_diagnosis :-
    \+ has_symptoms,
    writeln_message(no_symptom).
% If there are symptoms but there's no clear diagnosis, the system extracts the
conditions that may be involved (partial match with symptoms)
user_diagnosis :-
    has_symptoms,
    \+ has_condition,
    writeln_message(no_condition).
% Matches the symptoms of every condition with the observed ones
user_diagnosis :-
    has_symptoms,
    all(
        symptom(Location, Sign, Color),
        symptom(Location, Sign, Color),
        ObservedSymptoms
    ),
    all(diagnosis(Condition, ConditionSymptoms),
        (
            condition_symptoms(Condition, ConditionSymptoms),
            match(ConditionSymptoms, ObservedSymptoms)
        ),
        Diagnoses),
    maplist(explain, Diagnoses).
```

has_symptoms

```
has_symptoms :- symptom(_,_,_).
```

has_condition

```
has_condition :- condition(_).
```

condition_symptoms(-Condition, -ConditionSymptoms)

```
% Unifies ConditionSymptoms with the right side of the condition rule
condition_symptoms(Condition, ConditionSymptoms) :-
    condition(Condition),
    clause(condition(Condition), ConditionBody),
    conj_to_list(ConditionBody, ConditionSymptoms).
```

explain(+Diagnosis)

```
% If a diagnosis is reached, the plant is sick. The diagnosis is explained and the
treatment is shown, if present.
explain(Diagnosis) :-
    Diagnosis = diagnosis(Condition, [ConditionSymptoms]),
    show_diagnosis(Condition, ConditionSymptoms),
    show_treatment(Condition).
```

```
explain(Diagnosis) :-
    Diagnosis \= diagnosis(Condition, [ConditionSymptoms]),
    writeln_message(treatment_healthy).
```

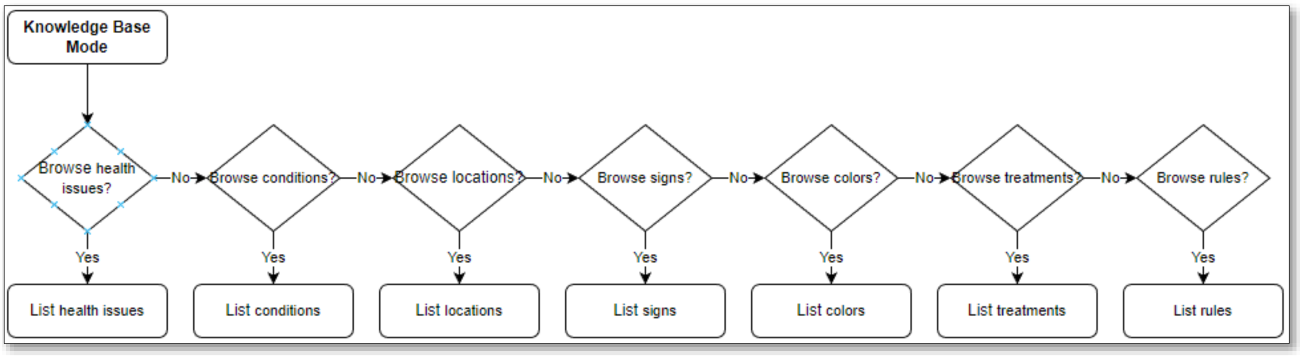
show_diagnosis(+Condition, +ConditionSymptoms)

```
show_diagnosis(Condition, ConditionSymptoms) :-
    problem_condition(Problem, Condition),
    write_message(because_of), write(ConditionSymptoms),
write_message(diagnosis_of), write(Problem), write(' - '), writeln(Condition).
```

show_treatment(+Condition)

```
show_treatment(Condition) :-
    problem_condition(nutrient_deficiency, Condition),
    writeln_message(missing_nutrient).
show_treatment(Condition) :-
    \+ problem_condition(nutrient_deficiency, Condition),
    bagof(Treatment, treatment(Condition, Treatment), Treatments),
    writeln_message(treatment),
    maplist(writeln, Treatments).
show_treatment(Condition) :-
    \+ problem_condition(nutrient_deficiency, Condition),
    \+ treatment(Condition),
    writeln_message(treatment_none).
```

Knowledge Base mode



kb_start

```
kb_mode_start :-
    L = [health_issues, conditions, locations, signs, colors, treatments, rules],
    maplist(kb_browse, L).
```

The kb_mode allows the user, straightforwardly, to browse the knowledge base categories facts. Lists are shown here.

kb_browse(+X)

```
kb_browse(X) :-
    askif(view(X)),
    browse(X).
kb_browse(X) :-
    asked(view(X), A),
    negative(A).
```

browse(X)

```
browse(X) :-
    X \= rules,
    writeln(X),
    call(X, L),
    maplist(writeln, L).
browse(X) :-
    X = rules,
    rules(L),
    maplist(writeln, L).
```

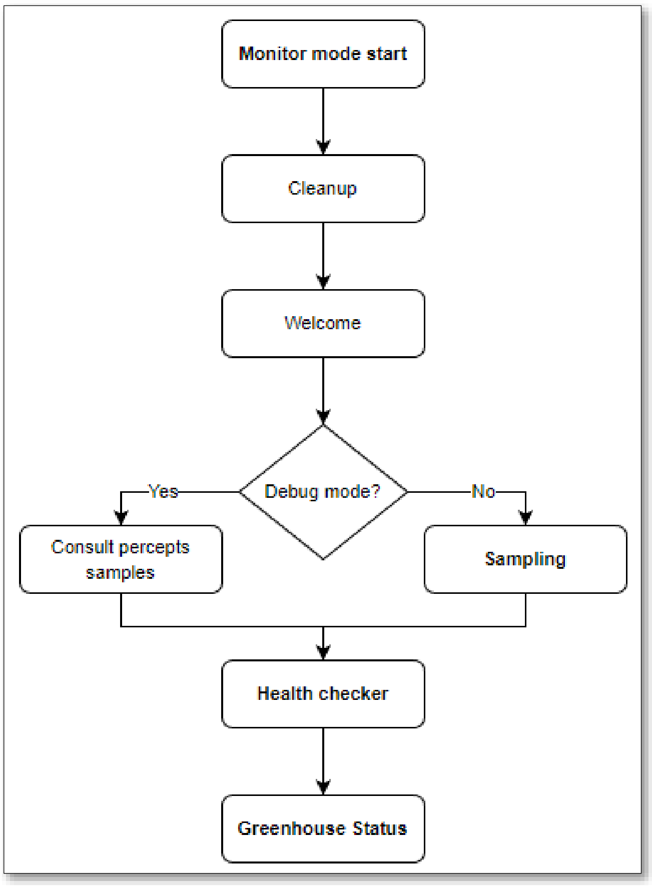
Monitor mode

monitor_mode_start

```
monitor_mode_start :-
    greenhouse_init,
    !,
    monitor_mode_forward,
    health_checker,
    greenhouse_status.
```

greenhouse_init

```
greenhouse_init :-
    monitor_cleanup,
    welcome_monitor,
    plants_reading_ranges(PlantsRanges),
    maplist(writeln, PlantsRanges), nl,
    actuators_init.
```



As for the user mode, the monitor mode starts by welcoming the user and cleaning up the cached informations.

It then acquires the ideal values for the environment temperature and humidity for every hosted plant in the greenhouse, printing them out for the users, and setting all connected actuators to the *off* status.

```
sampling_batch_size(X) :- X is 5.
sampling_interval(X) :- X is 1.
reading_variability(X) :- X is 1.3.
sign_probability(X) :- X is 1.
```

To make the system as realistic and as customizable as possible a few inputs parameters can be modified:

- the size of each batch of sampling
- the number of seconds between each sampling
- the probability of acquiring a sign from a camera sensor
- the variability of each metric reading

monitor_mode_forward

```
monitor_mode_forward :-
    debug_mode,
    consult(percepts_samples).
monitor_mode_forward :-
```

```
    \+ debug_mode,
    monitor_loop_start.
```

debug_mode

```
debug_mode :- askif(debug_mode).
```

monitor_cleanup

```
monitor_cleanup :-
    retractall(asked(_,_)),
    retractall(symptom(_,_,_)),
    retractall(plant_status(_,_,_)),
    retractall(plant_reading(_,_,_,_,_)),
    retractall(actuator_status(_,_)).
```

Returns a formatted timestamp to be attached to percepts.

actuators_init

```
actuators_init :-
    all(actuator_status(Actuator, off), plant_actuator(Plant, Actuator),
ActuatorsStatus),
    maplist(assertz, ActuatorsStatus).
```

All actuators installed on plants are set-up to off.

monitor_loop_start

```
monitor_loop_start :-
    retractall(asked(continue_monitor_loop, _)),
    sampling_repetitions(Repetitions),
    sampling_interval(Interval),
    monitor_loop(Repetitions, Interval).
```

Before starting the actual sampling, the user is asked if he wants to enter the debug mode.

This is due to speed up the testing: if the debug mode is executed, the timed sampling is skipped in favor of loading a pre-captured set of percepts from the *percepts_samples* file, as mentioned before.

monitor_loop(+Repetitions, +Interval)

```
% It simulates the interval between readings
monitor_loop(Repetitions, Interval) :-
    Repetitions > 0,
    sampling_init,
    sleep(Interval),
    RepetitionsNew is Repetitions - 1,
    monitor_loop(RepetitionsNew, Interval).
% at the end of the loop, asks if the user wants more samplings.
monitor_loop(Repetitions, Interval) :-
    Repetitions = 0,
    askif(continue_monitor_loop),
    monitor_loop_start.
% At the end of the sampling, it finds out the diagnoses and the starts the
actuators.
monitor_loop(Repetitions, Interval) :-
    Repetitions = 0.
```

The monitor activity is simulated through timed readings performed by sensor devices.

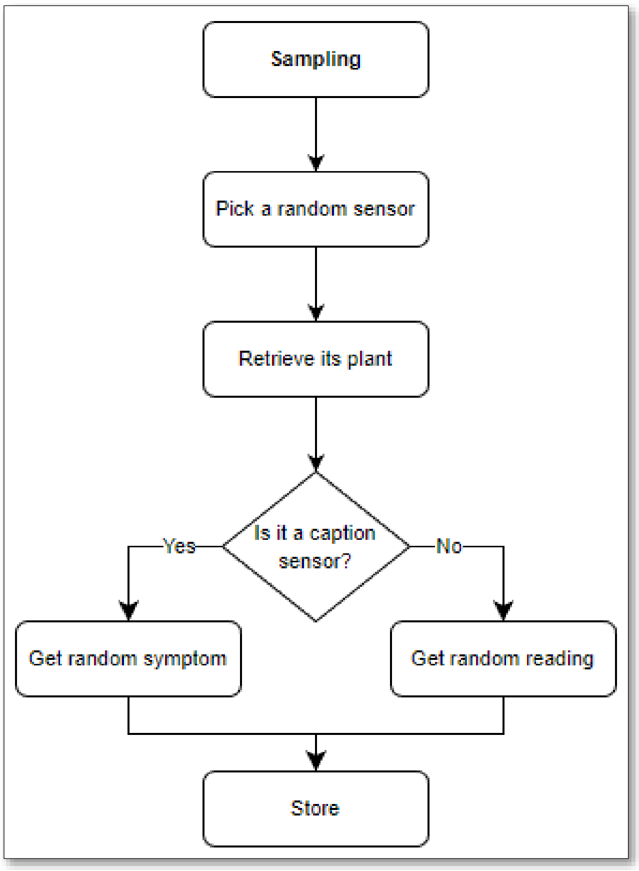
The loop performs repeated interspersed samplings until the batch size is reached, asking the user to continue with the next batch, restarting the loop. A negative answer terminates the process.

sampling_init

```
% the random sensor is picked up from
those already active on plants
sampling_init :-
    all(SensorID, plant_sensor(_,
SensorID), Sensors),
    random_list_element(Sensors,
RandomSensorID),
    sensor(RandomSensorID, SensorType),
    plant_sensor(Plant,
RandomSensorID),
    timestamp(Timestamp),
    sampling(Plant, Timestamp,
SensorType, RandomSensorID).
```

random_list_element(+List, -Element)

```
random_list_element(List, Element) :-
    length(List, Length),
    random(0, Length, RandomNumber),
    nth0(RandomNumber, List, Element).
```



One sensor is selected among all sensors connected to the plants: its type is retrieved and a sampling is generated with a timestamp.

The devices used in this case study are thermometers and hygrometers to capture temperature and humidity, and cameras for captions.

sampling(+Plant, +Timestamp, +SensorType, +SensorID)

```
% Performs a random sampling from a sensor that returns an integer number
sampling(Plant, Timestamp, SensorType, SensorID) :-
    SensorType \= caption,
    reading_variability(Variability),
    plant_range_values(Plant, SensorType, Min, Max),
    MinV is Min / Variability,
    MaxV is Max * Variability,
    (SensorType = humidity, MinV < 0 -> MinV1 = 0 ; MinV1 = MinV),
    (SensorType = humidity, MaxV > 100 -> MaxV1 = 100 ; MaxV1 = MaxV),
    random(MinV1, MaxV1, RandomValue),
    Value is floor(RandomValue),
    PlantReading = plant_reading(Plant, Timestamp, SensorType, SensorID, Value),
    store(PlantReading).
% Performs a random sampling from a camera device that returns a captioned symptom
sampling(Plant, Timestamp, SensorType, SensorID) :-
    SensorType = caption,
    sign_probability(SignProbability),
    random(CaptionIsSign), % Did the camera capture a sign?
    sampling_sign_probability(Plant, Timestamp, SensorType, SensorID,
SignProbability, CaptionIsSign).
```

plant_range_values(+Plant, +SensorType, -Min, -Max)

```
plant_range_values(Plant, SensorType, Min, Max) :-
    SensorType = temperature,
    plant(Plant, Species, _),
    species(Species, Min, Max).
plant_range_values(Plant, SensorType, Min, Max) :-
    SensorType = humidity,
    plant(Plant, _, GrowthStage),
    growth_humidity(GrowthStage, Min, Max).
```

If the selected sensor is not a camera, the reading value is generated by multiplying the *Variability* factor for a random value extracted from the ideal range of the environment metric.

Humidity extracted values are fixed in range between [0-100] and all readings values are rounded.

If the selected sensor happens to be a camera, the probability of captioning a valid sign - meaning a sign that is associated to an health condition – is provided by the *SignProbability.*

sampling_probability(+Plant, +Timestamp, +SensorType, +SensorID, +CaptionIsSign)

```
% Simulates the probability that a plant is healthy reading SignProbability
sampling_sign_probability(Plant, Timestamp, SensorType, SensorID, SignProbability,
CaptionIsSign) :-
    CaptionIsSign =< SignProbability,
    call(signs, Signs),
    random_list_element(Signs, Sign),
    caption_forward(Sign, Value), % symptoms are the values of captions!
    PlantReading = plant_reading(Plant, Timestamp, SensorType, SensorID, Value),
    store(PlantReading).
% sampling_sign_probability/6 If the plant is healthy, the symptom(none, all) is
stored
sampling_sign_probability(Plant, Timestamp, SensorType, SensorID, SignProbability,
CaptionIsSign) :-
    CaptionIsSign > SignProbability,
    caption_forward(none, Value),
    PlantReading = plant_reading(Plant, Timestamp, SensorType, SensorID, Value),
    store(PlantReading).
```

If the *SignProbability* is exceeded, no symptom is captured. The percept is stored at the end of the sampling.

caption_forward(+Sign, -Symptom)

```
% caption_forward/2
% If there is no associated color, sets the 3rd argument to none
caption_forward(Sign, Symptom) :-
    \+ sign_color(Sign, _),
    sampling_sign_location(Sign, RandomLocation),
    Symptom = symptom(RandomLocation, Sign, none).
caption_forward(Sign, Symptom) :-
    all(Color, sign_color(Sign, Color), Colors),
    random_list_element(Colors, RandomColor),
    sampling_sign_location(Sign, RandomLocation),
    Symptom = symptom(RandomLocation, Sign, RandomColor).
```

sampling_sign_location(+Sign, -RandomLocation)

```
% sampling_sign_location/2 - Unifies S with a random location related to the sign
sampling_sign_location(Sign, RandomLocation) :-
    all(Location, sign_location(Sign, Location), Locations),
    random_list_element(Locations, RandomLocation).
```

For the sake of the sampling within a virtualized environment, the percepts are simulated by randomly generating symptoms. The same filtering behavior adopted in the user_mode has been implemented here, as it would take time and computational power to register a valid symptom that could lead to a diagnosis, therefore making hard to illustrate the process.

store(+X)

```
% Stores a percept if it has not yet been observed. Does nothing otherwise
store(PlantReading) :-
    PlantReading = plant_reading(Plant, Timestamp, SensorType, SensorID, Value),
    assertz(PlantReading),
    logln(PlantReading).
store(PlantReading) :-
    PlantReading = plant_reading(Plant, Timestamp, SensorType, SensorID, Value),
    plant_reading(Plant, Timestamp, SensorType, SensorID, Value).
store(Symptom) :-
    Symptom = symptom(Location, Sign, Color),
    \+ symptom(Location, Sign, Color),
    assertz(Symptom),
    logln(Symptom).
store(Symptom) :-
    Symptom = symptom(Location, Sign, Color),
    symptom(Location, Sign, Color).
store(PlantStatus) :-
    PlantStatus = plant_status(Plant, SensorType, ReadingStatus),
    retractall(plant_status(Plant, SensorType, OldStatus)), % removes all previous
stored informations of the same sensor
    assertz(PlantStatus),
    logln(PlantStatus).
```

Percepts are stored as they are, provided that they haven't been observed yet.

Symptom's storing happens when a diagnosis is being searched for, one plant at time. As such, symptoms are stored only if they are newly observed: they would be of no use otherwise and would only use system resources.

As for plant statuses, only the last one is stored at each time: because it is possible that a previously captured state it's not valid anymore.

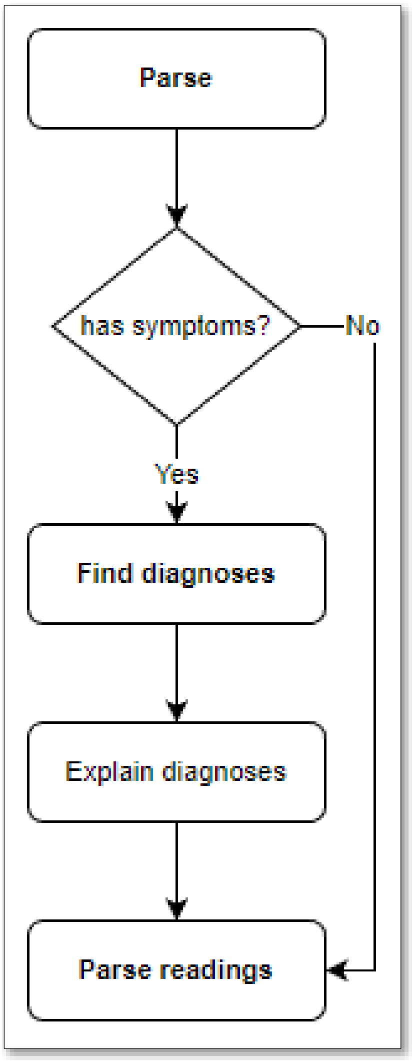
health_checker

```
health_checker :-
    all(Plant, SensorID^plant_sensor(Plant, SensorID), Plants),
    sort(Plants, SortedPlants),
    maplist(parse, SortedPlants).
```

The diagnostic process is performed at the end of the sampling by gathering all plants connected to sensors, sorting and parsing them.

parse(+Plant)

```
parse(Plant) :-
    atomic_concat(['\nPlant ', Plant, ' recap: '],
Message),
    logln(Message),
    parse_plant_symptoms(Plant),
    retractall(symptom(_,_,_)),
    parse_plant_readings(Plant),
    retractall(plant_reading(Plant,_,_,_,_)).
```



For every plant a recap is provided to the user through the CLI.

The parsing starts by considering all symptoms, after that it cleans the working memomry and then proceeds with analyzing the readings.

parse_plant_symptoms(+Plant)

```
parse_plant_symptoms(Plant) :-
    all(Symptom, (plant_reading(Plant, _, SensorType,
_, Symptom), SensorType = caption), PlantSymptoms),
    find_diagnoses(Plant, PlantSymptoms).
parse_plant_symptoms(Plant) :-
    \+ plant_reading(Plant, _, caption, _, Value),
    logln('* shows no symptoms').
```

All values of *plant_reading* taken for caption-type sensors are gathered. The list of all symptoms that affect the plant are then used to search for diagnoses.

If no symptoms are found, the diagnosis process is stopped and a message is shown and logged.

find_diagnoses(+Plant, +PlantSymptoms)

```
find_diagnoses(Plant, PlantSymptoms) :-
    % stores in the working memory the plant symptoms to match them with the
diagnoses causes
    maplist(store, PlantSymptoms),
    all((Condition, Symptoms), (
        condition(Condition),
        clause(condition(Condition), SymptomsConj),
        conj_to_list(SymptomsConj, Symptoms),
        match(Symptoms, PlantSymptoms),
        store(plant_status(Plant, caption, Condition))
    ), Diagnoses),
    explain_plant_diagnoses(Diagnoses).
```

```
find_diagnoses(Plant, PlantSymptoms) :-
    logln('^ no diagnosis').
```

All symptoms of the current plant are firstly stored in the working memory to make use of the backward chaining to find which conditions affects the plant.

To provide the user some explanations about the diagnoses, the symptoms of every observed condition are matched against symptoms of every condition in the KB.

If no diagnosis is reached, a message is returned to the user.

explain_plant_diagnoses(+List)

```
explain_plant_diagnoses([]).
explain_plant_diagnoses([H|T]) :-
    H = (Condition, Symptoms),
    problem_condition(Problem, Condition),
    atomic_concat(['^ diagnosis is of ', Condition, ' ', Problem, ' because of: '],
Message),
    log(Message),
    maplist(logln, Symptoms),
    explain_plant_diagnoses(T).
```

Once diagnoses are found they passed down as a list for every plant. The list is iterated recursively and logged at every step.

Each diagnosis is then displayed as a string, explaining that the condition is caused by a list of symptoms.

parse_plant_readings(+Plant)

```
% Retrieves the SensorType associated with the Plant and starts the parsing
parse_plant_readings(Plant) :-
    all(SensorType, (Plant^plant_sensor(Plant, SensorID), sensor(SensorID,
SensorType), SensorType \= caption), SensorTypes),
    log('- installed sensors '), logln(SensorTypes),
    forall(member(SensorType, SensorTypes), parse_plant_readings(Plant,
SensorType)).
parse_plant_readings(Plant).
```

Parsing the readings starts by gathering every sensor type connected to the plant.

parse_plant_readings(+Plant, +SensorType)

```
parse_plant_readings(Plant, SensorType) :-
    setof(V, (plant_reading(Plant, Timestamp, SensorType, SensorName, V),
SensorType \= caption), Values),
    last(Values, LastValue),
    retractall(plant_reading(Plant, _, SensorType, _, _)),
    (SensorType = temperature -> Type = ' Celsius' ; Type = '%'),
    reading_status(Plant, SensorType, LastValue, ReadingStatus),
    atomic_concat(['* ', SensorType, ' if of ', LastValue, Type, ', status is ',
ReadingStatus], Message),
    logln(Message),
    !,
    actuator_start(Plant, SensorType, ReadingStatus).
parse_plant_readings(Plant, SensorType) :-
    \+ plant_reading(Plant, _, SensorType, _, _),
    atomic_concat(['* ', SensorType, ' no reading'], Message),
    logln(Message).
```

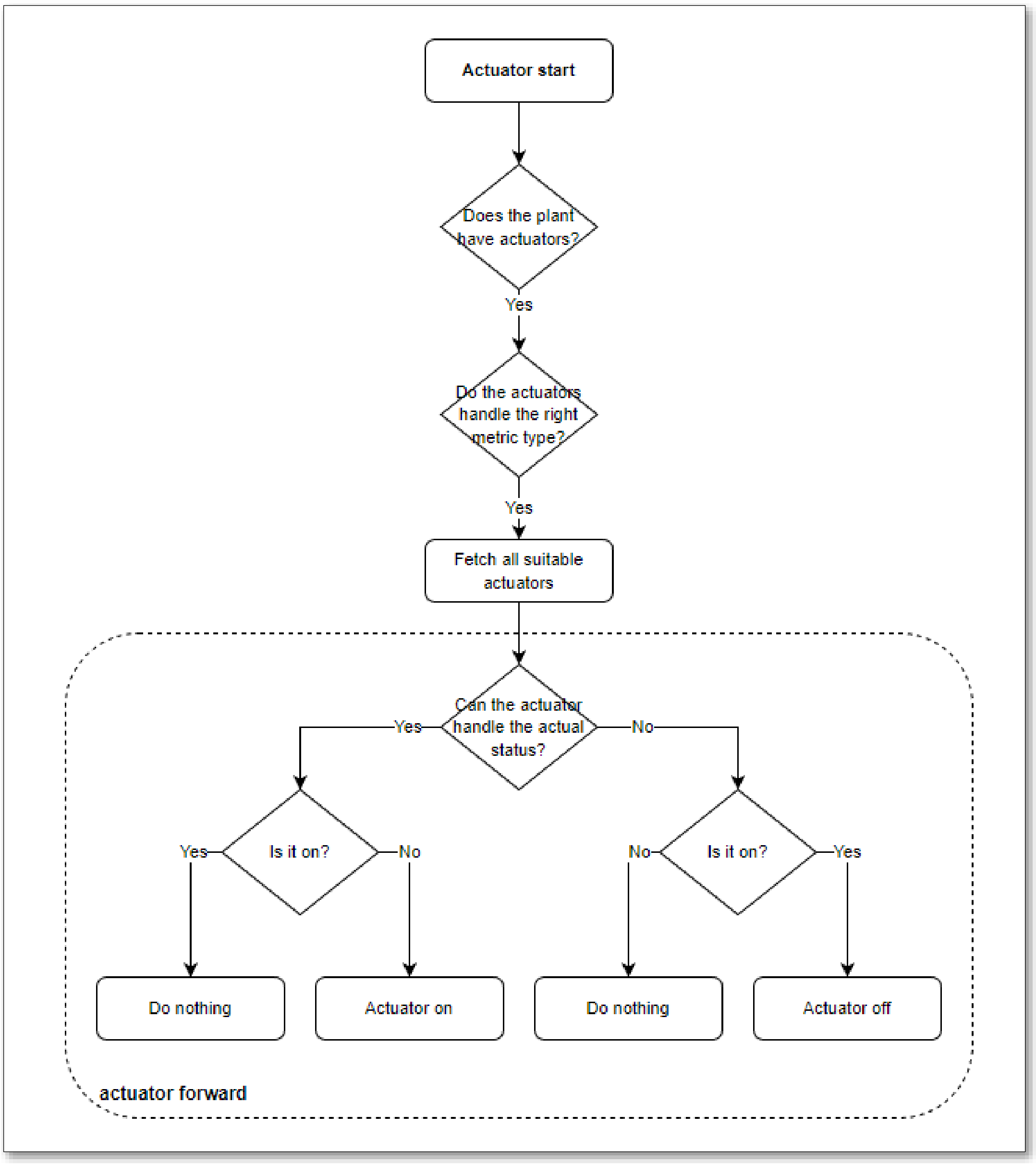
Only the last reading of the set of readings is taken into account, considering that time will flow before actuators effects will be in place and could show results.

Once the reading is forwarded to the actuator handler, the decision cannot backtrack any further and try to find another solution within the handler itself.

reading_status(+Plant, +SensorType, +Value, -ReadingStatus)

```
reading_status(Plant, SensorType, Value, ReadingStatus) :-
    plant_range_values(Plant, SensorType, Min, Max),
    Value < Min,
    ReadingStatus = low,
    PlantStatus = plant_status(Plant, SensorType, ReadingStatus),
    store(PlantStatus).
reading_status(Plant, SensorType, Value, ReadingStatus) :-
    plant_range_values(Plant, SensorType, Min, Max),
    Value > Max,
    ReadingStatus = high,
    PlantStatus = plant_status(Plant, SensorType, ReadingStatus),
    store(PlantStatus).
reading_status(Plant, SensorType, Value, ReadingStatus) :-
    plant_range_values(Plant, SensorType, Min, Max),
    between(Min, Max, Value),
    ReadingStatus = normal.
```

Based on the ideal range of the environment metrics, the plant status is evaluated and then stored. The *ReadingStatus* will be used to activate the actuators in the next steps.


Actuator start
Does the plant have actuators?
Yes
Do the actuators handle the right metric type?
Yes
Fetch all suitable actuators
Can the actuator handle the actual status?
Yes
No
Is it on?
Yes
No
Is it on?
No
Yes
Do nothing
Actuator on
Do nothing
Actuator off
actuator forward

`actuator_start(+Plant, +SensorType, +CurrentStatus)`

```
% There's no actuator on the plant
actuator_start(Plant, _, _) :-
    \+ plant_actuator(Plant, _),
    log(Plant, ' , no actuators').
% There's no actuator of the right type (temp/hum)
actuator_start(Plant, SensorType, _) :-
    \+ actuator(ActuatorID, SensorType, ActivationStatus, Class),
    plant_actuator(Plant, ActuatorID),
    atomic_concat([Plant, ', no ', SensorType, ' actuator'], Message),
    log(Message).
% Retrieves all actuators of the same SensorType (temp/hum) that can handle the
CurrentStatus and activates them simultaneously
actuator_start(Plant, SensorType, CurrentStatus) :-
    all((ActuatorID, Class, CurrentStatus, ActivationStatus),
        (plant_actuator(Plant, ActuatorID), actuator(ActuatorID, SensorType,
ActivationStatus, Class)),
        Actuators),
    maplist(actuator_forward, Actuators).
```

All actuators that can handle the specified SensorType and CurrentStatus for the specified plant are gathered and are forwarded to the next step. Otherwise, if there are no right type for actuators or if there are none at all, a message is provided to the user and the log is updated.

actuator_forward(+X)

```
% it turns the actuator off if it's not required anymore
actuator_forward(X) :-
    X = (ActuatorID, Class, CurrentStatus, ActivationStatus),
    CurrentStatus \= ActivationStatus,
    actuator_status(ActuatorID, on), % If it's on, it shuts it down
    retractall(actuator_status(ActuatorID, _)),
    assertz(actuator_status(ActuatorID, off)),
    atomic_concat(['\t', ActuatorID, ' ', Class, ' has been turned off'], Message),
    logln(Message).
% it's off and doesn't handle the actual status, so it does nothing.
actuator_forward(X) :-
    X = (ActuatorID, Class, CurrentStatus, ActivationStatus),
    CurrentStatus \= ActivationStatus,
    \+ actuator_status(ActuatorID, on),
    atomic_concat(['\t', ActuatorID, ' ', Class, ' has nothing to do'], Message),
    logln(Message).
% turns it on the actuator when needed
actuator_forward(X) :-
    X = (ActuatorID, Class, CurrentStatus, ActivationStatus),
    CurrentStatus = ActivationStatus,
    \+ actuator_status(ActuatorID, on), % If it's off, turns it on.
    retractall(actuator_status(ActuatorID, _)),
    assertz(actuator_status(ActuatorID, on)),
    atomic_concat(['\t', ActuatorID, ' ', Class, ' has been turned on'], Message),
    logln(Message).
actuator_forward(X) :-
    X = (ActuatorID, Class, CurrentStatus, ActivationStatus),
    CurrentStatus = ActivationStatus,
    actuator_status(ActuatorID, on), % If it's already on, does nothing.
    atomic_concat(['\t', ActuatorID, ' ', Class, ' was already on'], Message),
    logln(Message).
```

Actuators are turned on and off depending on the match between the *ActivationStatus* and the *CurrentStatus.*

- if there is *no match* and if they *were previously on,* they are turned *off.*
- if there is *no match* and they *were previously off*, there is *no change.*
- if there *is match* and they *were previously off,* they are *turned on.*
- if there *is match* and they *were previously on*, there is *no change*.

greenhouse_status

```
greenhouse_status :-
    logln('\nGreenhouse status:'),
    all(plant_health(Plant, Health), (plants(Plants), member(Plant, Plants),
plant_health(Plant, Health)), PlantsStatuses),
    maplist(logln, PlantsStatuses),
    logln('\n').
```

At the end of the whole process, all health statuses are displayed to the user as recap for the greenhouse.

Health statuses provide information of the current state for every plant.

plant_health(+Plant, -Health)

```
plant_health(Plant, Health) :-
    plant_status(Plant, SensorType, Reading),
    clause(problem(Problem), reading(SensorType, Reading)),
    issue_problem(Issue, Problem),
    Health = Issue-Problem.
plant_health(Plant, Health) :-
    plant_status(Plant, SensorType, Reading),
    problem_condition(Problem, Reading),
    issue_problem(Issue, Problem),
    Health = Issue-Problem-Reading.
plant_health(Plant, Health) :-
    plant_status(Plant, SensorType, Reading),
    \+ clause(problem(Problem), reading(SensorType, Reading)),
    \+ problem_condition(Problem, Reading),
    Health = healthy.
plant_health(Plant, Health) :-
    \+ plant_status(Plant, SensorType, Reading),
    Health = unknown.
```

At the end, a general diagnosis is reached by backward reasoning. If no problem is found the plant is redeemed healthy otherwise, if there are not enough informations, the status is set to unknown.

Logger

Logs are, currently, used only in the monitor_mode and are stored in the GreMaES.log file. All logs files are deleted at start.

logger_init

```
logger_init :-
    all(LogFile, logFile(File, LogFile), LogFiles),
    (member(LogFile, LogFiles) ->  file_exists(LogFile), delete_file(LogFile)).
```

log(+Message)

```
log(Message) :-
    logFile(File, FilePath),
    open(FilePath, append, Stream),
    write(Stream, Message),
    close(Stream),
    write(Message).
```

logln(+Message)

```
logln(Message) :-
    logFile(File, FilePath),
    open(FilePath, append, Stream),
    write(Stream, Message),
    nl(Stream),
    close(Stream),
    writeln(Message).
```

Lists, utilities

The following are called from different modules of the software, so they have been included in the *main.pl* file.

health_issues(-HealthIssues)

```
health_issues(HealthIssues) :-
    all(
        Issue-Problem,
        clause(health_issue(Issue), problem(Problem)),
        HealthIssues).
```

conditions(-Conditions)

```
conditions(ProblemsConditions) :-
    all(
        Problem-Condition,
        (
            clause(health_issue(Issue), problem(Problem)),
            clause(problem(Problem), condition(Condition))
        ),
        ProblemsConditions).
```

locations(-Locations)

```
locations(Locations) :- all(Location, sign_location(_, Location), Locations).
```

signs(-Signs)

```
signs(Signs) :- all(Sign, sign_location(Sign, _), Signs).
```

colors(-Colors)

```
colors(Colors) :- all(Color, sign_color(_, Color), Colors).
```

treatments(-Treatments)

```
treatments(Treatments) :-
    all(Condition-Treatment, treatment(Condition, Treatment), Treatments).
```

rules(-Rules)

```
rules(Rules) :-
    all(
        (Issue, Problem, Condition, Symptoms),
        (
            clause(health_issue(Issue), problem(Problem)),
            clause(problem(Problem), condition(Condition)),
            clause(condition(Condition), SymptomsConj),
            conj_to_list(SymptomsConj, Symptoms)
        ),
        Rules).
```

`problem_condition(Problem, Condition)`

```
problem_condition(Problem, Condition) :-
    clause(problem(Problem), condition(Condition)).
```

`issue_problem(Issue, Condition)`

```
issue_problem(Issue, Problem) :-
    clause(health_issue(Issue), problem(Problem)).
```

`plants(-SortedPlants)`

```
plants(SortedPlants) :-
    all(Plant, plant_sensor(Plant, _), Plants),
    sort(Plants, SortedPlants).
```

Sorted list of plants connected to sensors.

`plants_reading_ranges(-SortedPlants)`

```
plants_reading_ranges(SortedPlants) :-
    all(
        Plant-Species-temperature_range-TemperatureMin-TemperatureMax-
growth_humidity-GrowthStage-HumidityMin-HumidityMax,
        (plant(Plant, Species, GrowthStage), species(Species, TemperatureMin,
TemperatureMax), growth_humidity(GrowthStage, HumidityMin, HumidityMax)),
        Plants
    ),
    sort(Plants, SortedPlants).
```

Returns the sorted list of plants with all environment metrics and their respective ranges.

`timestamp(-T)`

```
timestamp(T) :-
    datime(datime(Year, Month, Day, Hour, Minute, Second)),
    T = timestamp(Year-Month-Day, Hour:Minute:Second).
```

`write_message(+MessageCode)`

```
write_message(MessageCode) :-
    message_code(MessageCode, Message),
    write(Message).
% writeln_message/1
writeln_message(MessageCode) :-
```

```
    message_code(MessageCode, Message),
    writeln(Message).
```

match(+L1, +L2)

```
match(L1, L2):- forall(member(X, L1), member(X, L2)).
```

conj_to_list((H, C), [H|T])

```
conj_to_list((H, C), [H|T]) :-
    !,
    conj_to_list(C, T).
conj_to_list(H, [H]).
```

list_to_conj([H|T], (H, C))

```
list_to_conj([H|T], (H, C)) :-
    !,
    list_to_conj(T, C).
list_to_conj([H], H).
```

Knowledge Base

Plant domain

What it follows is the basic knowledge of the Plant Domain that has been gathered for this case study. Being the product of an autonomous research conducted without any prior knowledge of the domain itself and not being verified by any domain expert, it may contain errors and omissions.

Nutrient deficiencies

Nitrogen

- pale yellow color (chlorosis)
- older leaves turn completely yellow.
- flowering, fruitings, protein and starch contents are reduced
- reduction in protein results in stunted growth and dormant lateral buds

Phosphorus

- smaller leaf sizes
- lessened number of leaves
- slower rate maturation
- leaves and stems appear dark green or purple
- older leaves are affected first

Potassium

- reduced growth
- chlorosis and necrosis occurring in older leaves in later growth stages
- older leaves show mottled or chlorotic areas with leaf burn at the margins, usually leaving the midrib alive and green
- brown scorching and curling of leaf tips as well as chlorosis (yellowing) between leaf veins
- purple spots may also appear on the leaf undersides
- plant growth, root development, and seed and fruit development are usually reduced in potassium-deficient plants

Sulfur

- resembles nitrogen deficiency except yellowing occurs in new, younger leaves, rather than old, lower leaves.

Magnesium

- interveinal chlorosis with green mid-ribs
- Leaf margins become yellow or reddish-purple

Boron

- chlorotic young leaves and death of the main growing point
- leaves may develop dark brown, irregular lesions
- whitish-yellow spots may form at the base of the leaves
- leaves may become thickened, distorted and curled
- stems may be stunted
- flower buds may fail to form or be misshapen

Calcium

- localized tissue necrosis leading to stunted plant growth
- necrotic leaf margins on young leaves or curling of the leaves, and eventual death of terminal buds and root tips
- new growth and rapidly growing tissues of the plant are affected first
- the mature leaves are rarely if ever affected
- reduced height, fewer nodes, and less leaf area

Chloride

- chlorotic and necrotic spotting along leaves with abrupt boundaries between dead and alive tissue
- wilting of leaves along margins
- highly branched roots

Copper

- chlorotic younger leaves
- stunted growth
- delayed maturity
- excessive tillering
- lodging and sometimes brown discoloration

Iron

- yellowing (Chlorosis) occurs in the newly emerging leaves instead of the older leaves and usually seen in the interveinal region

- fruit would be of poor quality and quantity
- the yellowing may turn a pale white or the whole leaf may be affected

Manganese

- plant disorder that is often confused with, and occurs with, iron deficiency
- most common in poorly drained soils, also where organic matter levels are high
- manganese may be unavailable to plants where pH is high
- yellowing of leaves with smallest leaf veins remaining green to produce a ‘chequered’ effect
- younger leaves may appear to be unaffected
- brown spots may appear on leaf surfaces
- severely affected leaves turn brown and wither

Zinc

- growth is limited because the plant
- cannot take up sufficient quantities of this essential micronutrient from its growing medium.
- chlorosis
- necrotic spots
- bronzing of leaves
- resetting of leaves
- stunting of plants
- dwarf leaves
- malformed leaves

Pests

Aphids

Aphids live only about a week, but a mature female can reproduce rapidly. The tiny sucking pests, often found growing en masse on the underside of leaves, emit a sticky substance that draws ants and attracts sooty mold. Control aphids with neem oil or insecticidal soap.

- mass in large number
- sticky honeydew deposits
- white or grey "husks" littering the soil
- leaves become chlorotic in random patches
- growth may become distorted
- Treatment
- spray with neem oil and insecticidal soap

Thrips

Thrips are tiny flying insects with fringed wings. The sap-sucking insects discolor and distort nearly any type of plant. They leave tiny black specks of excrement on the leaves and often create white patches on leaves and petals. Thrips are difficult to control and often require a combination of methods such as sticky traps and insecticidal soap or neem oil.

- mottling, streaking, browning or yellowing on the leaves
- Treatment
- cut off and remove the infected leaves or flowers.
- spray with neem oil or natural pyrethrum

Spider mites

Spider mites are difficult to see with the naked eye, but they are easily recognized by the fine webs. The pests cause streaking, spotting and discolored leaves that may fall off the plant if not controlled. Neem oil and insecticidal soap are effective. Water properly, as mites are drawn to dry, dusty conditions.

- sticky webbing
- mottled leaves with lots of brown dots

Treatment

- shower the plant once a week
- purchasing the predatory mite Phytosieulus persimilis
- spray with neem oil and insecticidal soap

Scale insects

Scale damage can be devastating, as the tiny pests suck out the sweet nectar. There are two types of scale: hard scale, found primarily on woody tissue such as branches, trunks and twigs; and soft scale, which has a waxy protective covering. Control can be difficult, but neem oil works well by suffocating the pests. Regular use of insecticidal soap is also effective.

- sticky honeydew

Treatment

- spray neem oil and insecticidal soap
- dab individual scales with alcohol

White flies

Whiteflies are yet another type of sap-sucking pest. Small numbers are relatively harmless but large infestations can cause yellow or dry leaves that may fall off the plants. Like other sap-sucking pests, the sweet substance created by whiteflies attracts ants and sooty mold. To control whiteflies, try sticky traps and insecticidal soap or neem oil.

- leaves chlorosis
- dry leaves

Treatment

- spray neem oil and insecticidal soap
- sticky trap

Cutworms

Cutworms are the larval stage of certain moths. The destructive pests hide under leaves or other plant debris, emerging to lay masses of eggs on plants. They eat nearly anything in their paths, often cutting through stems of young plants at ground level. Remove plant debris. Pick off the pests by hand in late afternoon or evening. Create barriers with cardboard collars or gritty substances like eggshells, coffee grounds, or diatomaceous earth. Encourage birds to visit your garden.

Fungus gnats

Fungus gnats are tiny, annoying pests that wreak havoc on houseplants or in gardens or greenhouses. The swarms of flying insects are annoying, but it’s the larvae that does the most harm by eating plant roots. Fungus gnats may also carry disease from plant to plant. Control adults with bright yellow sticky traps and/or insecticidal soap.

- small black flies around 2mm long

Treatment

- keep the soil less moist until they leave
- use the bottom watering method
- mix the nematodes with water and water directly

Mealy bugs

Mealybugs are common both indoors and outdoors, where they cause stunted growth, withering and yellowing of plants. The pests are easily recognized by the cottony protective covering. Insecticidal soap works well against the pests. Light infestations on indoor plants can also be removed with a toothpick or a cotton swab dipped in rubbing alcohol.

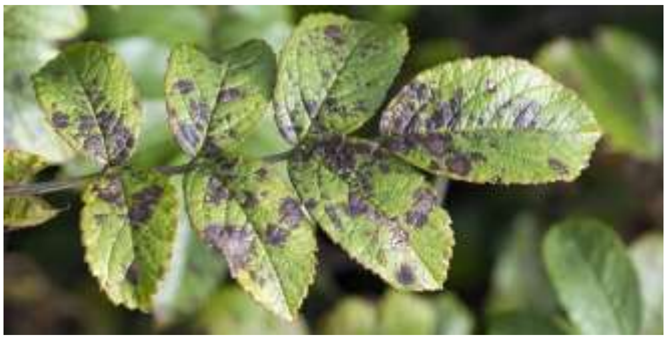
- clustering cottony covering under leaves and in the leaf joints
- plants look dehydrated
- plants may lose leaves rapidly
- stunted growth
- chlorotic leaves eventually drop off
- sticky honeydew residue

Treatment

- poke them off with a shake
- spraying with water
- spray neem oil and insecticidal soap

Diseases

Black spot

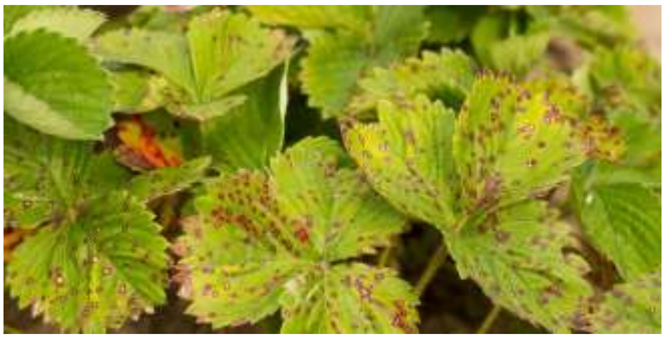


fungal, black round spots, upper side leaves (lower ones infected first). Infected leaves turn yellow and fall off. It occurs in extended wet weather periods or when leaves are wet for 6+ hours.

Tips for Controlling Black Spots on Leaves

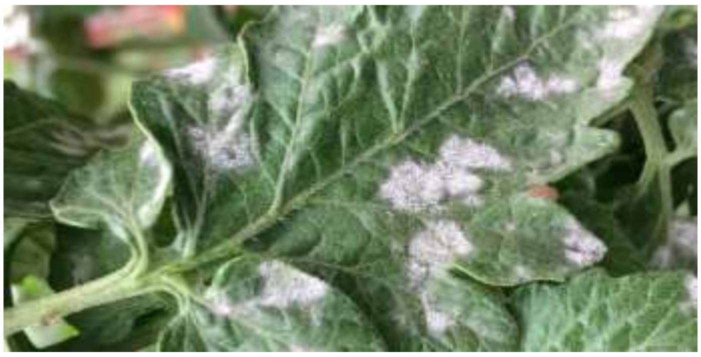
- Plant in well-draining soil. Keep your plants healthy by providing regular feedings of organic fertilizer. This will help prevent fungal disease in plants.
- The fungus spores overwinter in plant debris. Remove dead leaves and infected canes from around the plants and disguard in the trash. Do not add to the compost pile.
- Disinfect your pruners with a household disinfectant after every use. Ethanol or isopropyl alcohol can be used straight out of the bottle.
- Because water (not wind) spreads the fungal spores, avoid applying water on the leaves. When you water, apply water directly to the roots. Use a soaker hose to water plants prone to the disease.

Leaf Spots



Fungal leaf spot disease can be found both indoors on houseplants, and outdoors in the landscape. This occurs during warm, wet conditions. As the disease progresses, the fungal spots grow large enough to touch each other. At this point the leaf surface appears more like blotches than spots. Leaf spot may result in defoliation of a plant. Follow the same tips as the ones to control black spot.

Powdery Mildew

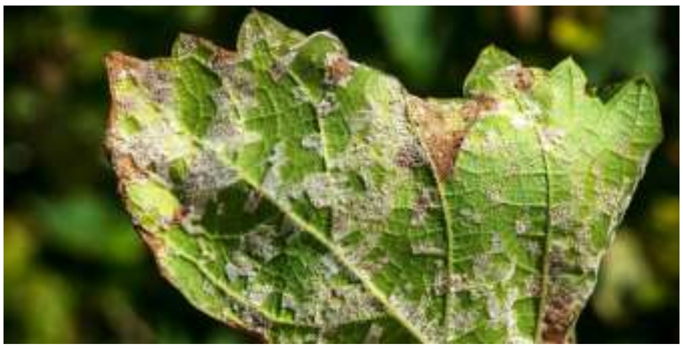


Powdery mildew is a fungal disease that affects many of our landscape plants, flowers, vegetables and fruits. Powdery mildew is an easy one to identify. Infected plants will display a white powdery substance that is most visible on upper leaf surfaces, but it can appear anywhere on the plant including stems, flower buds, and even the fruit of the plant. This fungus thrives during low soil moisture conditions combined with high humidity levels on the upper parts of the plant surface. It tends to affect plants kept in shady areas more than those in direct sun.

Tips for Controlling Powdery Mildew

- Inspect plants that you buy from a greenhouse before purchasing for mildew (and insects).
- Wiping off the leaves is not an effective powdery mildew treatment as it will return within days of cleaning.
- Because spores overwinter in debris all infected debris should be removed. Trim and remove infected plant parts.
- Do not till the debris into the soil or use in the compost pile.
- Space plants far enough apart to increase air circulation and reduce humidity.

Downy Mildew

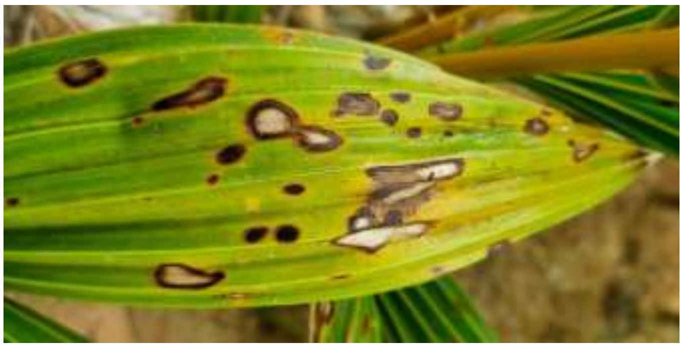


Downy mildews produce grayish fuzzy looking spores on the lower surfaces of leaves. To identify downy mildew, look for pale green or yellow spots on the upper surfaces of older leaves. On the lower surfaces, the fungus will display a white to grayish, cotton-like downy substance. Downy mildew occurs during cool, moist weather such as in early spring or late fall. Spore production is favored by temperatures below 65°F and with a high relative humidity.

Tips for Downy Mildew Treatment

- Downy mildew needs water to survive and spread. It there is no water on your leaves, the disease cannot spread. Keep water off leaves as much as possible.
- Because the disease overwinters on dead plant debris, be sure to clean around your plants in the fall to help prevent the disease in the following spring.

Blight

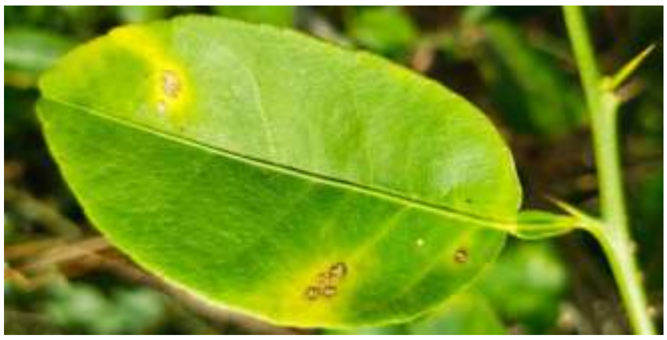


Blight is a fungal disease that spreads through spores that are windborne. For this reason, spores can cover large areas and rapidly spread the infection. Blight can only spread under warm humid conditions, especially with two consecutive days of temps above 50°F, and humidity above 90% for eleven hours or more. No cure exists. Prevention is the only option.

Tips for Preventing Blight

- If growing potatoes, grow early varieties because blight occurs during mid-summer and you can harvest your crop before the blight.
- Plant resistant varieties: Sarpo Mira and Sarpo Axona are two varieties that show good resistance. Practice good garden hygiene.
- Destroy any blight-infected plant parts. Keep the area clean of fallen debris from your diseased plants and discard in the trash. Do not add to your compost pile.

Canker



Canker is often identified by an open wound that has been infected by fungal or bacterial pathogens. Some cankers are not serious while others can be lethal. Canker occurs primarily on woody landscape plants. Symptoms may include sunken, swollen, cracked or dead areas found on stems, limbs or trunk. Cankers can girdle branches and kill foliage. Cankers are most common on stressed plants that have been weakened by cold, insects, drought conditions, nutritional imbalances or root rot. Rodents can also spread the pathogens.

Tips for Controlling Canker in Plants

- Remove diseased parts in dry weather.
- Grow resistant varieties whenever possible.
- Avoid overwatering and overcrowding; avoid mechanical wounds such as damage from lawn mowers.
- Wrap young, newly planted trees to prevent sunscald. Sunscald creates dead patches that form on trunk and limbs of young trees if the trunks have been shaded, then transplanted to sunny areas.
- Keep plants healthy by planting in healthy soils and maintaining nutritional requirements.

Shot Hole

This disease of peach, apricot, plum and cherry spreads in warm wet weather infecting buds, blossoms, leaves, fruit and twigs (not large branches). Leaves develop numerous small, tan to purplish spots about 6 mm in diameter that drop out causing a shot hole appearance. Red to purplish spots also form on the fruit and can be accompanied by a clear, gummy substance. Gummy twig and small branch cankers also occur.

Shot Hole Control

plant resistant varieties. Rake up and destroy fallen leaves and prune out and destroy infected twigs and branches. To prevent twig and bud infections spray with Copper Spray: Peaches after harvest and all other trees in September before fall rains start.

Late Blight, Early Blight

Late Blight and Early Blight these are fungal diseases of tomatoes, potatoes and other related plants. Early blight appears as dark brown to black leaf spots with concentric rings. Black spots develop on stems and large, black, leathery, sunken spots on the fruit. Infections often occur in May or June in wet years. Late blight forms irregular greenish black, water-soaked blotches first on older leaves or stems quickly spreading to the fruit. This disease usually doesn't appear until August in wet years, but it can destroy entire plants overnight.

Late Blight and Early Blight Control

Space and prune plans for good air circulation. Avoid overhead watering. If Early blight starts to appear, pick off and destroy the infected leaves. If chemical control is required apply a copper spray at 7 to 10 day intervals. If late blight starts to appear remove diseased leaves or entire plants immediately, seal in a plastic bag and send to the landfill. Do not compost late blight infected plants. Apply a copper spray at every 5 to 10 days till allowed days before harvest.

Botrytis Blight or Grey Mold

Botrytis Blight or Grey Mold is a grey fuzzy mold develops on dead and dying plant tissue spreading to healthy tissue when conditions are wet. Infections first appears as water-soaked spots or areas on soft or senescent foliage, flower parts and young stems. On flowering plants, woody ornamentals and small fruit this disease can cause flower, leaf and shoot blights as well as stem and fruit rots. Very susceptible plants include: peonies, roses, hostas, strawberries and raspberries.

Botrytis Blight or Grey Mold Control

Plant resistant cultivars. Thoroughly clean and discard garden debris and refuse in the fall to reduce the level of grey mold in your garden. Susceptible plants (that are sun loving) should be grown in sunny areas with good air circulation. If practical water at the base of plants not over the foliage. If botrytis appears, remove infected leaves and fruit. It is rarely worth applying fungicides to control this disease.

Verticilium Wilt

Verticilium Wilt is a serious fungal disease of many deciduous trees, herbaceous perennials, berries and vegetables. It is of particular concern for flowering cherries. It enters roots from the soil moving upwards in the plant, plugging up the plants transportation system. Visible indication that there is a problem starts with yellowing, wilting and dying back of young twigs and branches often on one side of plant or tree. Many other problems look the same, however Verticillium wilt gets worse from year to year. Cutting into a woody stem with a knife reveal black or brown streaks in the wood are vascular cambium just under the bark.

Verticilium Wilt Control

Control is all preventative as there is no cure once a plant is infected. Avoid drought stress or flooding on mature landscape trees. Remove dead and dying plants including the infested roots and the soil and replant with tolerant or resistant species. When pruning trees that may have this disease, sterilize your pruning tools between trees to prevent spreading it to an and noninfected tree. Rubbing alcohol, Lysol or a 10% household bleach solution (corrosive) can be used to disinfect pruning tools. Once an area is infected with Verticilium Wilt, we generally suggest not planting the same species in that area for several years.

Conclusions

Future developments