

# Processes and the Process Engine

Joachim De Beule  
joachim@arti.vub.ac.be

May 14, 2004

## **Note**

This document is part of the documentation of a software system designed to investigate the prerequisites, the origins and the evolution of grammatical language in a population of autonomous artificial agents. As the system is still under development this documentation is preliminary and reflects (a part of) the system as it was at the date of publication of this document. The author of this document is Joachim De Beule, although it should be noted that many people have contributed and are still contributing to the system (Luc Steels, Joris Van Looveren, Nicolas Neubauer.) Comments regarding this document are welcome.

## **Abstract**

This document describes the process engine of the system. The first part gives an overview and illustrates some aspects with two executable examples. The second part gives a more detailed description which should enable you to write processes and understand how they will be executed by the process engine.

# Contents

<b>1</b>	<b>Overview</b>	<b>3</b>
<b>2</b>	<b>Examples</b>	<b>4</b>
2.1	First Example . . . . .	4
2.2	A More Elaborate Example . . . . .	5
2.3	The Simple Process Macro . . . . .	9
<b>3</b>	<b>The Mprocess-Definition structure</b>	<b>11</b>
3.1	The Name Slot . . . . .	11
3.2	The Active-p slot . . . . .	11
3.3	The Required Processes . . . . .	11
3.4	The Init-Function . . . . .	12
3.5	The Run Function . . . . .	12
3.6	The Fix Proposal Function . . . . .	13
3.7	The Execute Fix Function . . . . .	13
3.8	The Feedback Function . . . . .	14
3.9	The Parameter- and Data-Copier slots . . . . .	14
3.10	The Specialist-p Slot . . . . .	14
3.11	The Parameter-Spec Slot . . . . .	14
<b>4</b>	<b>Tasks and the Process Engine</b>	<b>15</b>

# 1 Overview

The process engine is responsible for running processes, collecting their status and output data, registering problems, asking for fix proposals and asking to execute a proposal. It is also responsible for creating new tasks and start a search when multiple hypotheses are returned by a process.

When an agent has to achieve a goal, a task for that goal has to be created and run. A task consists of a set of processes that all try to contribute to the goal. This is done in a blackboard-system like way but with some optimizations. During a run cycle a process-state is kept for every process in the task:

```
(defstruct process-state
  confidence
  name
  active
  data
  parameters
  status
  required-processes
  depending-processes
  problems ;; list of problems the process received
  reported ;; list of problem id's the process has reported
  recalculate-p) ;;; boolean: set to nil after the process has run,
```

Among other things such a process-state holds the status and output data of the process as the process returned it after it was last run. The state also contains a reference to the problems the process reported or received. Some processes might require the data of another process. in this case it is said that the process requires or depends upon the other process. Whenever a process is run it should also specify if it changed something that depending processes should know about. The first time a task is run every process is given a chance to run and return initial status and data information and to report problems. After that a process will not run as long as one of the problems that the process reported is not solved. If this is not the case (there is no unsolved problem) the process will only be run when one of the required processes reported a change or when a problem that was reported to the process is solved. Note that the solving of a problem that the process reported itself does not trigger the process unless the problem was reported to itself. When a process is run it might return several hypotheses. For each of them a new task is created and a search is started. Whenever a task fails it is added to the agents' list of failed tasks. When there are no tasks left or when a task achieves the goal the search stops. The task can then receive feedback which is passed on to its processes.

Before giving any details we will first give two examples. Although not everything will be clear at first, going over the examples is probably a good idea and will help to understand the rest of the document.

## 2 Examples

To define a process you have to do at least three things:

1. define an `mprocess-definition` for the process and add it to the `agent-process-definitions` slot.
2. add the process name to the global variable `*task-process-names*` which is an association list with cells of the form `(goal . process-names)`.
3. add a `goal-test-function` to the global variable `*task-goal-tests*` which is an association list with cells of the form `(goal . goal-test-function)`. `Goal-test-function` should be a function on a task argument returning a non-nil value if the tasks goal is achieved.

An `mprocess-definition` is defined as follows:

```
(defstruct mprocess-definition
  name
  required-processes
  (active-p t)
  init-function
  run-function
  propose-fixes-function
  do-fix-function
  feedback-function

  parameter-copier
  data-copier

  specialist-p
  parameter-spec))
```

### 2.1 First Example

As a simple example, suppose you want to build an agent that is capable of perceiving the world by looking at the output data of some event detection system. Assume the data of the even detection system can be retrieved by calling the function `perceive-world`:

```
(setf *task-process-names* '((example . (perceive-world))))
(setf *task-goal-tests*
  (list (cons 'example
             #'(lambda (task)
                 (get-process-data task 'perceive-world)))))

(setf agent (create-agent))
(setf (agent-process-definitions agent)
  (list (make-mprocess-definition
        :name 'perceive-world
        :run-function
        #'(lambda (agent task)
            (format t "% This is process perceive-world")
            (add-to-process-return
             'succeed
             (perceive-world (make-world-model))
```

```

1.0
nil)
*process-return*)))))
(create-task agent 'example)
(init-world)
(new-game)
(set-trace '(process-engine process-engine-verbose))
(run-task agent (agent-current-task agent))

```

This example actually works (hence the game and world initialization calls) and produces the following output:

```

Adding process PERCEIVE-WORLD (1)
----- Running task 25 (confidence 0) ----- (2)
running process PERCEIVE-WORLD (3)
  (process trigger and/or problems solved: (INITIAL)) (4)
  This is process perceive-world (5)
  status SUCCEED, (6)
  output #S(WORLD-MODEL ...) (7)
new-tasks: 25 (T), (8)
Task 25 succeeded (at confidence value 1.0) (9)
----- (10)

```

Line (1) is printed during the call to `create-task` stating that the process `perceive-world` is added to the task. From line (2) to (7) the task is actually run. In line (3) one can see that the `perceive-world` process is run, this is because this is the first time the task is run as one can see in line (4). Line (5) is due to the format in the `perceive-world` run function. In line (8) all processes of the task have run and since the `perceive-world` process returned only one hypotheses no new task is created. the (T) in line (8) means that the task changed in some way so should run again, but as can be seen in line (9) this is not needed since the goal is achieved.

## 2.2 A More Elaborate Example

In this example two additional processes are added to illustrate problems and process-requirements. The first additional process will select a topic from the world model. This requires the `perceive-world` data. The second additional process will try to do a lexicon lookup for the topic but will report a problem to itself whenever it encounters an unknown meaning. The example code is shown below:

```

(setf *task-process-names* (1)
      '((example . (perceive-world select-topic lexicon-production))))
(setf *task-goal-tests*
      (list (cons 'example
                  #'(lambda (task)
                      (eq (get-process-status task 'lexicon-production)
                          'succeed))))))

(setf agent (create-agent))
(setf (agent-process-definitions agent) (10)
      (list (make-mprocess-definition
              :NAME 'perceive-world
              :RUN-function

```

```

#' (lambda (agent task)
  (let ((previous (get-process-data task 'perceive-world)))
    (add-to-process-return
      (if previous 'achieved 'succeed)
      (or previous (perceive-world (make-world-model)))
      1.0 NIL (not previous)))
    *process-return*)) (20)

(make-mprocess-definition
 :NAME 'select-topic
 :REQUIRED-PROCESSES '(perceive-world)
 :RUN-FUNCTION
 #' (lambda (agent task)
  (let ((world (get-process-data task 'perceive-world)) fact)
    (when world
      (setq fact
        (second
          (thing-facts
            (first (world-model-objects world))))))
      (add-to-process-return
        'succeed
        '((unit (referent (,(second (fact-content fact))))
          (meaning (,(fact-content fact))))))
      1.0 NIL T)))
    *process-return*)) (40)

(make-mprocess-definition (40)
 :NAME 'lexicon-production
 :REQUIRED-PROCESSES '(select-topic)
 :RUN-FUNCTION
 #' (lambda (agent task)
  (let ((topic (get-process-data task 'select-topic)))
    (when topic
      (dolist (hypotheses (lsplit-structure (47)
        (agent-lexrules agent task)
        (copy-tree topic)))
        (setf (fourth hypotheses) (50)
          (delete-if-not
            #' (lambda (unit)
              (unit-feature-value unit 'meaning))
            (fourth hypotheses)))
          (add-to-process-return (55)
            (if (fourth hypotheses) 'fail 'succeed)
            (make-lexicon-data (57)
              :sem-data (second hypotheses)
              :syn-data (if (fourth hypotheses) nil
                (third hypotheses))
              :rules-used (fifth hypotheses)
              :rules-not-used (sixth hypotheses))
            (first hypotheses) (63)
            (loop for unit in (fourth hypotheses) collect
              (make-problem
                :from '(lexicon-production )
                :to 'lexicon-production
                :message '(missing-word-meaning ,unit)))
            (fifth hypotheses))))))
    *process-return*)) (69)

```

```

:PROPOSE-FIXES-FUNCTION (71)
  #'(lambda (agent task problem)
    (when (and (consp problem)
              (eq (first problem) 'missing-word-meaning))
      '(((invent-new-word ,(second problem))))))
:DO-FIX-FUNCTION (76)
  #'(lambda (agent task proposal)
    (when (and (consp proposal)
              (eq 'invent-new-word (first proposal)))
      (introduce-new-word (80)
        (agent-lexrules agent task)
        (retrieve-meaning (second proposal))
        :referent (determine-word-referent (second proposal))
        (set-process-return 'succeed nil t))
      *process-return*))))
(create-task agent 'example) (87)
(init-world)
(new-game)
*context*
(set-trace '(process-engine process-engine-verbose))
(run-task agent (agent-current-task agent))

```

In lines (1) to (7) the three processes are added to the example goal and the goal is defined to be achieved whenever the lexicon-production process succeeds. From line (10) to (85) the three processes are defined and added to the agent. The first process is identical to the perceive-world process in the first example, except that care is taken now that if the process already produced output once it will not recalculate its output again but return an achieved status (line 17) and only report a change the first time (last expression in line 19.) The select-topic process (lines 21-39) requires the perceive-world process. The most interesting process is lexicon-production. The `lsplit-structure` function does a lexicon lookup for a semantic structure and returns a list of hypotheses. Every hypothesis is specified by a list containing the following elements:

1. a score,
2. a semantic structure (this could be different from the original semantic structure because it is split up according to the lexicon),
3. a corresponding syntactic structure,
4. the part of the semantic structure that is not covered by the lexicon,
5. a list of the rules used
6. a list of the rules that matched but were not used because they conflict with other rules that are used.

In lines (55) to (69) one can see that this information is used to determine a process return for every hypotheses. In particular, in lines (64) to (68) you can see that the process reports a `missing-word-meaning` problem to itself when there is uncovered meaning. The `propose-fixes-function` (line 71) will be called to get an `invent-new-word` fix proposal to solve such a problem. And

the do-fix-function (line 76) will be called to execute the proposal by calling the invent-new-word function. See some other document for more information on these functions.

This example will produce the following output:

```

Adding process PERCEIVE-WORLD (1)
Adding process SELECT-TOPIC
Adding process LEXICON-PRODUCTION
----- Running task 46 (confidence 0) -----
running process PERCEIVE-WORLD (5)
  (process trigger and/or problems solved: (INITIAL))
  status SUCCEED,
  output #S(WORLD-MODEL
            :OBJECTS
            (#S(THING :NAME OBJECT-676 (10)
                  :FACTS
                  (#S(FACT :CONTENT (GREEN OBJECT-676)) ...))
            ...))
            ...))
running process LEXICON-PRODUCTION (15)
  (process trigger and/or problems solved: (INITIAL))
running process SELECT-TOPIC
  (process trigger and/or problems solved: (PERCEIVE-WORLD INITIAL))
  status SUCCEED,
  output ((UNIT (REFERENT (OBJECT-676) (20)
                (MEANING ((GREEN OBJECT-676))))))
new-tasks: 46 (T),
----- Running task 46 (confidence 0.6666667) -----
running process PERCEIVE-WORLD (25)
  (process trigger and/or problems solved: NIL)
running process LEXICON-PRODUCTION
  (process trigger and/or problems solved: (SELECT-TOPIC))
  status FAIL,
  output #(Lexicon Data
          Semantic structure: (30)
            ((UNIT
              (REFERENT (OBJECT-676))
              (MEANING ((GREEN OBJECT-676))))))
          Syntactic structure:
            ((NIL)) (35)
          Rules used:
          Rules not used:)
  problem 153=(MISSING-WORD-MEANING
              (UNIT (REFERENT (OBJECT-676))
                    (MEANING ((GREEN OBJECT-676)))) (40)
              reported to LEXICON-PRODUCTION by LEXICON-PRODUCTION
new-tasks: 46 (T),
problems & fixes: (#S(FIX-PROPOSAL
                      :PROBLEM-ID 153
                      :FIX (INVENT-NEW-WORD (45)
                                (UNIT (REFERENT (OBJECT-676))
                                      (MEANING ((GREEN OBJECT-676))))))
                      :CONFIDENCE NIL))
Fix for problem 153: SUCCEED
----- Running task 46 (confidence 0.6666667) ----- (50)
running process PERCEIVE-WORLD
  (process trigger and/or problems solved: NIL)

```

```

running process LEXICON-PRODUCTION
(process trigger and/or problems solved: (153))
status SUCCEED, (55)
output #(Lexicon Data
  Semantic structure:
    ((UNIT
      (MEANING ((GREEN OBJECT-676)))
      (REFERENT (OBJECT-676)))) (60)
  Syntactic structure:
    ((UNIT
      (FORM ((STRING UNIT LONETI))))))
  Rules used:
    Rule ENTRY-LONETI-11950 (use 0, score 0.5): (65)
      ((?UNIT
        (MEANING (== (GREEN ?OBJECT-6765)))
        (REFERENT (?OBJECT-6765))))
      <-->
      ((?UNIT (70)
        (FORM (== (STRING ?UNIT LONETI))))))
  Rules not used:)
new-tasks: 46 (T),
Task 46 succeeded (at confidence value 0.8333334)
----- (75)

```

Line (15) shows that the lexicon-production is run because this is the first time the task is run. But since the lexicon-production requires the topic-selection process data it does not return anything of interest. Line (18) states that the topic-selection process is triggered not only because it is the first time it is run but also because the perceive-world process reported a change. The process succeeds and returns a semantic structure describing the topic. As line (27) shows this triggers the lexicon-production process again. The process fails and reports a missing-word-meaning problem (with id 153) to itself (line 38). A fix is proposed for it (line 42) and executed (line 49). This triggers the lexicon-production process again (line 54) which succeeds this time. The new lexrul can be seen in lines (65) to (71). Line (74) shows that the task succeeds at a confidence value of 0.8333. This is because of the way the confidence value of a task is calculated: it is the sum of confidence values of every process in the task divided by the number of processes in the task. Since the perceive-world and the select-topic processes both return a value of 1.0 and the lexicon-production process returns a value of 0.5 (the score of the new lexrul that was created) you get  $(1+1+0.5)/3=0.8333$ .

### 2.3 The Simple Process Macro

There is a macro `simple-process` that can be used to define a process. This macro accepts a set of keywords to set the different slots of an `mprocess` definition. In addition it provides a default framework to define a process that supports skill and challenge parameters. This is explained in some other document. For now, the macro is illustrated by using it to define the lexicon-production process as follows:

```

(simple-process lexicon-production
:REQUIRED-PROCESSES ((select-topic topic (not topic) nil))

```

```

:SKILLS ((use-lexicon 1 0 1 1))
:DO #'(lambda (agent task parameters data)
      ... ;; lines (47) to (69)
      )
:PROPOSE-FIX-FN ... ;; lines (72) to (75)
:DO-FIX-FN ... ;; lines (77) to (85)
)

```

which expands into

```

(PROGN
  (DEFCONSTANT *LEXICON-PRODUCTION-PROCESS-NAME*
    (QUOTE LEXICON-PRODUCTION))
  (DEFVAR *TRACE-LEXICON-PRODUCTION* T)
  (DEFVAR *LEXICON-PRODUCTION-DEFINITION*)
  (DEFUN LEXICON-PRODUCTION-DATA-INIT (TASK)
    (SET-PROCESS-DATA TASK *LEXICON-PRODUCTION-PROCESS-NAME*
      (FUNCALL #<Function LIST>)))
  (DEFUN LEXICON-PRODUCTION-RUN (AGENT TASK)
    (RELEASE-PROCESS-RETURN)
    (LET* ((#:PARAMETERS (GET-PROCESS-PARAMETERS TASK 'LEXICON-PRODUCTION))
           (TOPIC (GET-PROCESS-DATA TASK 'SELECT-TOPIC))
           (USE-LEXICON (GET-SIMPLE-SKILL USE-LEXICON #:PARAMETERS))
           (:#DATA (GET-PROCESS-DATA TASK 'LEXICON-PRODUCTION)))
      (COND ((NOT TOPIC)
        (ADD-TO-PROCESS-RETURN 'IRRELEVANT #:DATA 1.0 NIL))
        ((<< USE-LEXICON 1)
        (ADD-TO-PROCESS-RETURN 'LOW-SKILL #:DATA 1.0 NIL))
        (T (LET ((#:RESULT
          (FUNCALL #'(LAMBDA (AGENT TASK PARAMETERS DATA)
            ... ;; lines (47) to (69)
            AGENT TASK #:PARAMETERS #:DATA)))
          (UNLESS *PROCESS-RETURN*
            (ADD-TO-PROCESS-RETURN 'UNKNOWN #:RESULT 0.0 NIL))
          (WHEN *TRACE-LEXICON-PRODUCTION*
            (FORMAT T "~%Process ~A" 'LEXICON-PRODUCTION)
            (FUNCALL #<Function (:INTERNAL SIMPLE-PROCESS 0)>
              *PROCESS-RETURN*))))))
        *PROCESS-RETURN*)
    (SETF (SYMBOL-FUNCTION 'LEXICON-PRODUCTION-PROPOSE-FIXES)
      ... ) ;; lines (72) to (75)
    (SETF (SYMBOL-FUNCTION 'LEXICON-PRODUCTION-DO-FIX)
      ... ) ;; lines (77) to (85)
  (DEFUN LEXICON-PRODUCTION-FEEDBACK (AGENT TASK RESULT TOPIC UTTERANCE)
    (DECLARE (IGNORE AGENT TASK RESULT TOPIC UTTERANCE)))
  (SETF *LEXICON-PRODUCTION-DEFINITION*
    (MAKE-MPROCESS-DEFINITION
      :NAME *LEXICON-PRODUCTION-PROCESS-NAME*
      :ACTIVE-P T
      :SPECIALIST-P NIL
      :REQUIRED-PROCESSES '(SELECT-TOPIC)
      :INIT-FUNCTION #'LEXICON-PRODUCTION-DATA-INIT
      :PARAMETER-COPIER #<Function COPY-SIMPLE-PARAMETER-SPEC>
      :DATA-COPIER NIL
      :RUN-FUNCTION #'LEXICON-PRODUCTION-RUN
      :PROPOSE-FIXES-FUNCTION #'LEXICON-PRODUCTION-PROPOSE-FIXES
      :DO-FIX-FUNCTION #'LEXICON-PRODUCTION-DO-FIX
      :FEEDBACK-FUNCTION #'LEXICON-PRODUCTION-FEEDBACK
    )
  )

```

```

:PARAMETER-SPEC (SIMPLE-PARAMETER-SPEC
                 :CHALLENGES NIL
                 :SKILLS ((USE-LEXICON 1 0 1))))
*LEXICON-PRODUCTION-DEFINITION*)

```

As can be seen some additional things are defined and taken care of, for example the support of challenge and skill parameters.

### 3 The Mprocess-Definition structure

In the following sections the different slots of the `mprocess-definition` are explained in more detail.

#### 3.1 The Name Slot

This should be a symbol and *has* to be provided. Take care to be consistent in referring to the process by using this name.

##### Using the Simple Process Macro

The first argument to the macro should be the name (not quoted).

#### 3.2 The Active-p slot

When this slot is NIL the process' run function will *never* be called. It is still possible to report problems to this process and its propose-fixes-function and do-fix-function will be called. This allows for a separation of the run functionality and problem handling of a process and to manipulate the process flow. For example, when a problem is reported to a process the process will be triggered when the problem is solved. Thus, when a problem is reported to an inactive process instead of to the problem reporting process itself, no process will run by default after the problem is solved.

#### 3.3 The Required Processes

This should be a list of process names. A process is run when one of its required returned a `process-return` with a non nil `changed` value (see section 3.5.)

##### Using the Simple Process Macro

The `:REQUIRED-PROCESSES` keyword lets you specify a set of required processes in the following way. The value to the keyword should be a list of elements of the form

```

(process-name | (process-name*)
 process-data-name
 run-fail-condition
 fail-condition-block)

```

Within the body of the run-function the symbol `process-data-name` will be bound to the process data of the process with name `process-name` (or the first

process that exists and has non-nil data in the case that multiple process names are specified.)

The part of the run function specified by the `:DO` keyword will not be run if any of the `run-fail-conditions` evaluates to true. In this case the `fail-condition-block` will be executed and a process-return with status `irrelevant`, data the previous data, confidence 1.0 and changed flag nil is returned.

### 3.4 The Init-Function

This should be a function on two arguments: an agent and a task. This function will be called when the process is added to a task or when the task is reset.

#### Using the Simple Process Macro

Not supported.

### 3.5 The Run Function

When the process engine decides to run a process its run function is called. This should be a function of two arguments: an agent and a task. It should return a list of process-return structures:

```
(defstruct process-return
  status
  data
  confidence
  problems
  changed)
```

The status slot should be one of `{irrelevant, succeed, achieved, fail, unknown}`.

The confidence slot should be a number between 0 and 1. It will be used to calculate the confidence value of the task the process is part of.

The data slot may contain whatever the process wants to remember the following time it is called, or it wants to pass to other depending processes. The value of this slot can be retrieved by calling `(get-process-data task process-name)`.

The problems slot should be a list of problem structures:

```
(defstruct problem
  from
  to
  message)
```

`From` and `to` should be process-names (symbols). The process will be suspended as long as there is a problem it reported that is not solved or explicitly ignored.

The `changed` slot is used to force the processes that depend on this process to recalculate. Setting this slot to a non-nil value triggers the depending processes even if the process data did not actually change. It is a good idea to *always* specify this slot.

The function `add-to-process-return` can be used to add an element to the process return.

```
(defun add-to-process-return (status data confidence problems
                             &optional (changed-p nil))
  ...)
```

If this function is used you have to make sure that the last line that is executed in the run function is always `*process-return*`.

### Using the Simple Process Macro

If the `simple-process` macro is used the body of the run function is specified by the `:DO` keyword argument. This value for this keyword should be a function on four arguments: an agent, a task, parameters and data. The parameters and data arguments will, be bound to the current process parameters and the previously returned data respectively.

You *have* to specify the process return value by using the `add-to-process-return` function but you do not have to take care of returning `*process-return*` yourself. In addition you can make use of the data of required processes simply by referring to the name that was assigned to it in the `:REQUIRED-PROCESSES` slot.

## 3.6 The Fix Proposal Function

The process engine may ask a process to propose a fix for a problem by calling the process' `propose-fixes-function`. This should be a function on three arguments: an agent, a task and a message (the contents of the message-slot of a problem.) This function should return nil (no fix proposed) or else a list of two elements: a fix proposal and a confidence value.

### Using the Simple Process Macro

The entire `propose-fix-function` is specified with the `:PROPOSE-FIX-FN` keyword. If not specified, a default `propose fix` function is defined that takes care of proposing fixes for challenge and skill parameters.

## 3.7 The Execute Fix Function

This function should execute a fix proposed by the fix proposal function. It should be a function on three arguments: an agent, a task and a proposal (the first element of the return value of the fix proposal function. This function should return one of `succeed`, `at-max`, `ignore`, `fail`.

### Using the Simple Process Macro

The entire `do-fix-function` is specified with the `:DO-FIX-FN` keyword. If not specified, a default handler function is defined that handles parameter update requests.

### 3.8 The Feedback Function

This should be a function of five arguments: an agent, a task a result a topic and an utterance. The result is normally one of `success`, `failure`, `problem`, the topic the name of a thing (a symbol) and the utterance a list of symbols.

#### Using the Simple Process Macro

The entire feedback function is specified with the `:FEEDBACK-FN` keyword.

### 3.9 The Parameter- and Data-Copier slots

These slots let you provide copy functions for the process' parameters and data. This is important because when some process returns multiple hypotheses a new task will be created for every hypotheses. For this the parameters and data of every process have to be copied into the new task in such a way that they become independent, i.e. when the data is modified in one task it will not be modified in another. The parameter-copier function should accept and return a parameters data structure. The data-copier should accept and return a data argument.

#### Using the Simple Process Macro

With the `:DATA-COPIER` keyword the data-copier can be specified. With the `:PARAMETER-COPIER` keyword the data-copier can be specified. A default parameter-copier function `COPY-SIMPLE-PARAMETER-SPEC` is defined and used since a simple-process uses simple-parameter structures.

### 3.10 The Specialist-p Slot

A specialist is a special kind of process that requires some additional initialization and handling by the process-engine. This is not further explained in this document.

#### Using the Simple Process Macro

The value of the slot is set to the value of the `SPECIALIST-P` keyword which is by default equal to `NIL`.

### 3.11 The Parameter-Spec Slot

This slot lets you specify the parameters of the process.

#### Using the Simple Process Macro

The slot is set to a `simple-parameter-spec` structure according to the specification provided by the `:REQUIRED-PROCESSES` keyword.

## 4 Tasks and the Process Engine

As explained in the examples the variable `*task-process-names*` is an association list specifying which processes should be added to a task to solve a certain goal. The variable `*task-goal-tests*` should contain the associated goal tests on a task. A task for a goal is created with the function `create-task` which takes as an argument an agent and a goal (a symbol). The `process-definitions` slot of the agent should contain the `mprocess-definitions` of the processes associated with the goal.

The function `create-task` will make and return a fresh task and initialize it such that for example process-dependencies are calculated and stored. It will also create and store a process-state for every process added and set the `recalculate-p` slot to true. Finally it will set the `current-task` slot of the agent to the new task.

A task is run by calling the function `run-task` which takes an agent and a task. Next to a `current-task` slot an agent also has `failed-tasks` and `task-queue` slots. Running a task goes as follows:

1. the task queue is cleared and the current task is added to the queue with its `changed` flag set to true.
2. if the queue is empty return fail.
3. else the task with the highest confidence value is removed from the queue and becomes the current task.
4. if the current task contains problems then try to solve one of them in the following way: Loop over every problem in chronological order (i.e. the oldest problem first). Ask the different processes for fix proposals for the problem. Select the fix with the highest confidence value and try to execute it. If it succeeds goto step (5), else try the next problem.
5. if for the current task the goal is achieved, return the current task.
6. if the current task has its `changed` flag set to nil add the task to the `failed-tasks` list and return to step (2)
7. For every process in the current task the following happens:
  - (a) first it is decided if it should run. A process should always run when its `recalculate-p` slot is true or when it does not depend on any other process unless one of the following things hold: (i) it is inactive (slot `active` in its state is NIL) or (ii) it has reported a problem that is not handled yet. If it should run goto step (b), else consider the next process in (a) or if there are no processes left goto (2).
  - (b) Before running a process its `recalculate-p` flag is set to nil. Then its `run-function` is called. This function returns a list of hypotheses, every hypothesis being a process-return structure. If multiple hypotheses are returned a new task is created for each of them. For

every hypotheses the problems in the process return are added to the corresponding task and the new state and process data are stored (replacing the old ones). If the changed-p flag of the process-return is true the recalculate-p slot of all depending processes is set to true and the changed flag of the task is set to true. The confidence value of the hypotheses is added to the score of the task.

- (c) When multiple hypotheses were returned add every (new) task to the task queue and go to step (2) (even when not every process got a chance to run.) Else continue with the next process or go to step (2) if no processes are left after adding the current task to the task-queue again.