# Narval Technical Manual

**Olivier Cayrol**
**Alexandre Fayolle**
**Sylvain Thénault**

# Narval Technical Manual

Olivier Cayrol

Alexandre Fayolle

Sylvain Thénault



## Legal Notice



## Abstract

Narval is both a language and an interpreter for this language. The language is well suited for writing intelligent personal assistants. This document presents its internal architecture and its coding philosophy. Reading this document is advised to all people wanting to study Narval source code, or just to understand how it works.

## Revisions

| Num. | Date | Author | Remarks |
|---|---|---|---|
| $Revision: 1.4 $ | $Date: 2001/10/16 10:18:25 $ | $Author: alf $ | |

# Table of Content

# Warning

This document presents with great details how Narval works. It is expected that the reader has understood the purpose of the application, and some global knowledge of the functionalities. It is strongly advised to read the User Manual first.

*Chapter I*

# Narval Operation

## 1. Elements, interfaces and adapters

Everything in narval is based on what we call *elements*. Those elements are used to hold any information but also to describe a desired behaviour of the agent, using elements such as *actions* and *recipes*, or to control it, using elements such as start-plan, *plans*, etc... All of them are manipulated by the interpreter and contained in it's *memory*.

Elements are actually python class instances, and so defined in python modules and (usually automatically) registered to the interpreter. Also, for a greater flexibility, the concepts of *interface* and *adapter* are used:

- Interfaces and adapters, as elements, are defined by classes in python modules and registered to the interpreter.

## 2. The memory: notion of context nesting

Narval has a *memory* in which it stores *elements* that it handles. In the memory, elements are grouped according to the *plans* that use them. It is thus possible to define a context for each plan (see Figure 1).

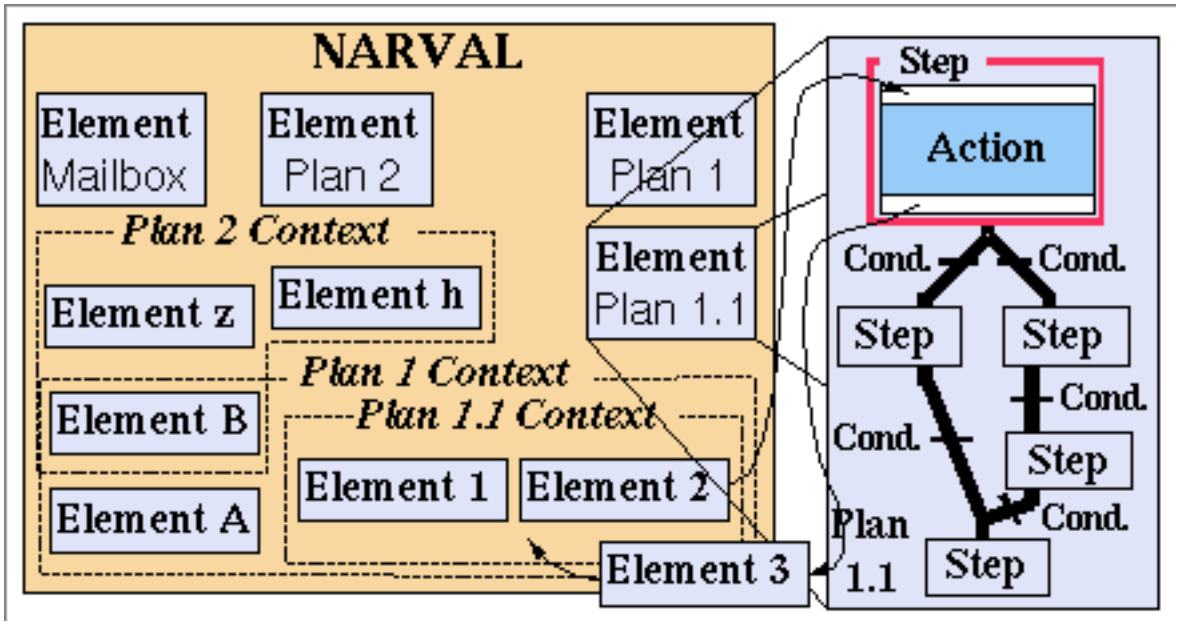


**Figure 1 -** Narval's memory

The *context* of a plan is the set of elements handled by the plan, i.e. all the elements created or used by the *steps* of the plan or that triggered the *transitions* of the plan. The memory holds everything handled by Narval, that is, all the contents of the contexts, as well as general interest elements, such as plans, recipes

or, as shown in the example (see Figure 1), a `mailbox` element which represents the mail box used by Narval. Since it is meant to be shared by several plans, this element is held in the memory, and in no specific context. When a plan is run from within another plan, its context is nested within its parent's context. This is the case in the example shown in Figure 1) with plan `Plan 1.1` having a context nested in plan `Plan 1`'s.

During plan execution, elements are dynamically added to the different contexts (see element `Element 3` in Figure 1). Elements in memory that have not been used after a given amount of time may be automatically removed so that the memory size will not grow too much.

# 3. Difference between recipes and plans

## 3.1. Introduction to recipes

A *recipe* is a specification of a sequence of *steps* and *transitions* necessary to complete a given task. Recipes represent everything a Narval can do. A recipe is described with steps and transitions but cannot be executed. This requires the instantiation of a *plan*.

*Recipes* are *elements* stored in Narval's *memory*.

When a plan is built from a recipe (see Figure 2), Narval creates a new `plan` element in memory, which has all the data required for the execution to the steps and transitions found in the recipe.

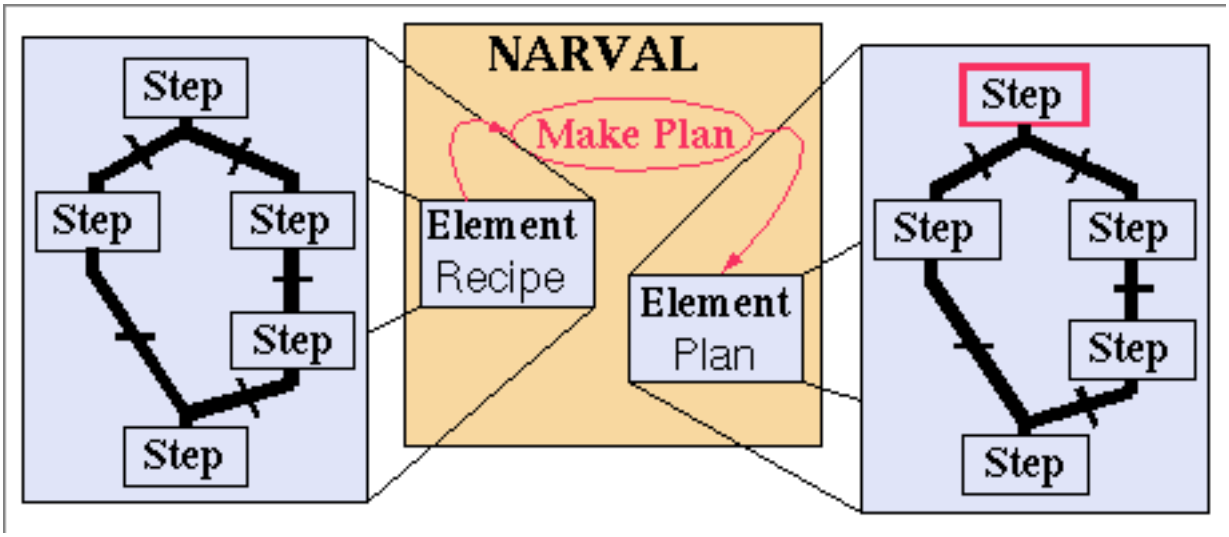


**Figure 2 -** Building a plan from a recipe

## 3.2. Plan execution

When executing a plan, Narval starts with the first step (see Section 4 "*Running a step*" - Chapter I). After that, the outgoing transitions of this step are evaluated, and if one can be fired, Narval runs the destination steps of the transition, and so on. Executing a plan is much like walking through a graph.

In the next example, Figure 3, after having run the *action* (label 1), Narval evaluates the transitions (label 2) and only fires one of them to select the next action (label 3).

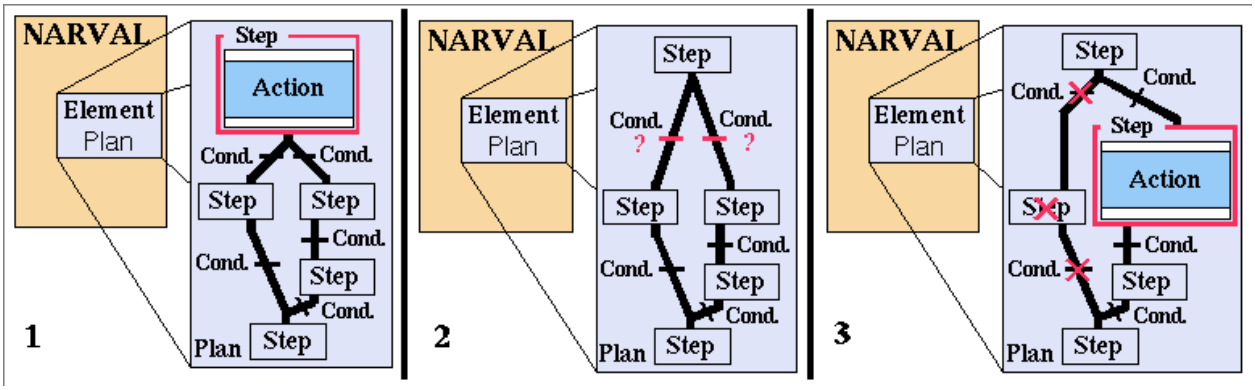


**Figure 3** - Plan execution

# 4. Running a step

Two kinds of *steps* can exist in a *recipe*, depending on the target, which can be a *recipe* or an *action*.

## 4.1. General behaviour

If the step's target is a recipe, Narval creates a new plan from this recipe and executes it within the *context* of the parent plan (the context of the new plan is consequently nested within the current plan's).

When the target is an action or transformation, Narval simply runs the target (see Section 5 "*Running an action or a transition*" - Chapter I).

## 4.2. Special behaviour: the *foreach* attribute

The *foreach* attribute can only be set on steps whose target is an action. It specifies that the action must be run for each element matching a given input (the *foreach* attribute gives the input identifier). In the example presented on Figure 4), the `Send_greetings` action takes an email address element as an input with Email_Ad as identifier (those addresses are stored in `Email_Ad` elements), and creates an new year greetings email (stored in an `Email` type element). The step representing this action has a foreach attribute set to `Email_Ad` as the desired input identifier. The action will thus be parallelized for each email found in memory (in the example, three times). In the end, the action will have created three `Email` elements.

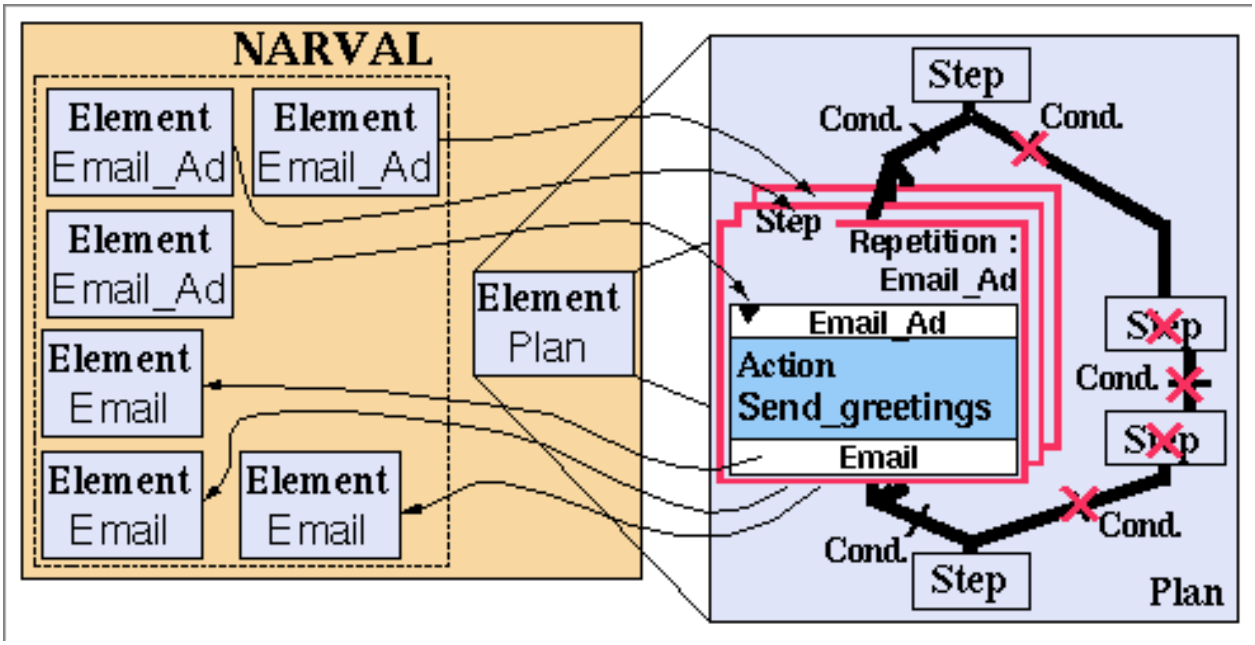


**Figure 4** - Foreach attribute in a step

# 5. Running an action or a transition

**Note**

**Remark**

Unless explicitely noted, all that is said about actions is also true for transitions in this section. In order to make the text more readable, we shall only speak of actions.

## 5.1. Prototype of an action

Actions used by Narval provide a prototype. In other words, they describe the elements they require in order to execute correctly and the elements that they produce when the execution was successful. For instance, the `Catch_new` action which gets new emails in a mail box, specifies that it needs a `mailbox` element as an input, and that it produces `email` elements as outputs.

Each action input describes an element type that is required by the action. For each input, it is possible to specify additional properties.

### 5.1.1. Used and consulted inputs

An input may be marked as used by an action using the *use* attribute. When an action uses an input, an element matching the input can only be passed once as an input to the action. This is generally the expected behaviour for an action that transforms an element into another element. On the opposite, if the element is not marked as used, it can be used over and over by plans instantiated from the same recipe. In this case the element is said to be consulted.

If we consider Figure 5, action `catch_new` has a `mailbox` element as an input, and this input is

consulted. When two plans issued from the same recipe are run (labels 1 and 2), the same `mailbox` element can be used both times. This is obviously the expected behaviour since we always want to fetch emails in the same mailbox.

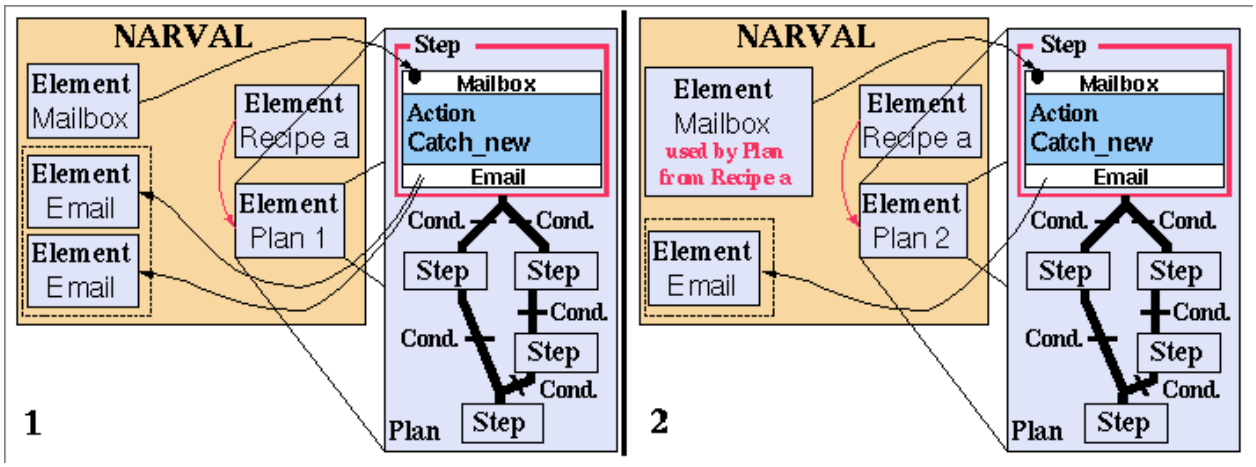


**Figure 5 -** Action with a consulted input

On the other hand, in Figure 6, action `Acknowl_mail` which acknowledges reception of mails takes `Email` elements as inputs, and uses them. As a consequence if, during the first execution (label 1), the `Email` element number `1` is processed, it is not processed again on the second execution (label 2), and element number `2` which was not there on the first time is processed. Once again, this is the expected behaviour, since we want to acknowledge each mail only once, but we also wish to keep them in memory so that they can be passed to other plans.

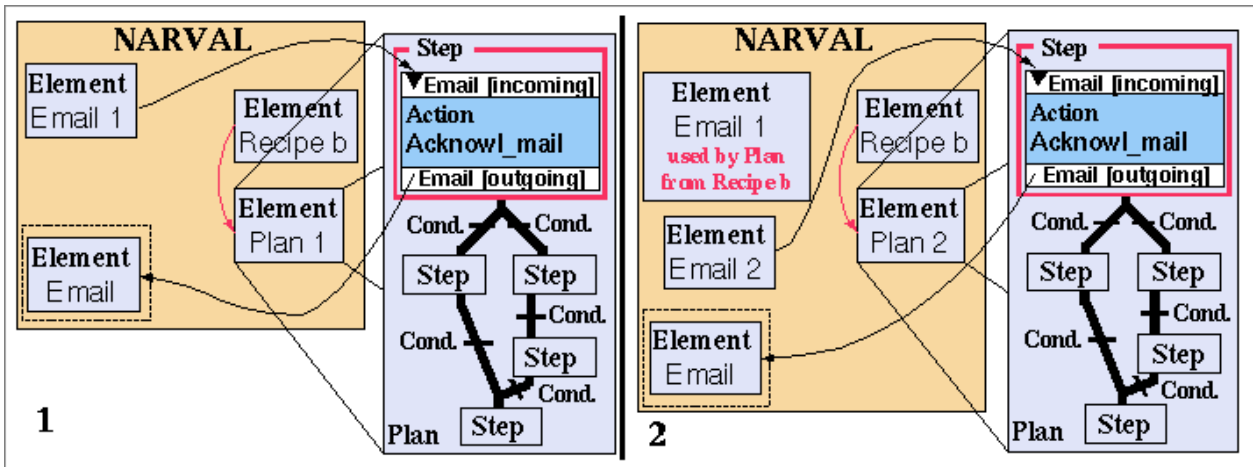


**Figure 6 -** Action with used inputs

### 5.1.2. Outdated inputs

There are times where it is necessary to make sure that an element will never be reused after having been passed by an action. This is called outdating the element, and is achieved by specifying an *outdates* attribute on the input with the value of 'yes'. This is like using the *use* attribute, except that no other action or transtion will ever be able to use the element.

A common use for this is emulating global variables in Narval: if an action outputs an element of the same type as the outdated input, the effect will as if the new element replaced the original one.

### 5.1.3. Single element or list of elements

In most cases, an action input is supposed to match only one element at run time. However, it is sometimes necessary to specify that an input can be matched by an unknown number of elements. For instance, in Figure 7, action `Send_greetings2` takes a list of email addresses (`Email_Ad` elements) as an input, in order to send to all the persons in the list a new year greetings email. Please note that this action creates a single `email` regardless of the number of `Email_Ad` elements. This is quite different from the foreach attribute in a step: in Figure 4, the `Send_greetings` action step had created one `Email` for each `Email_Ad`.

For each input, the list attribute specifies whether a single element or a list of elements must be used.

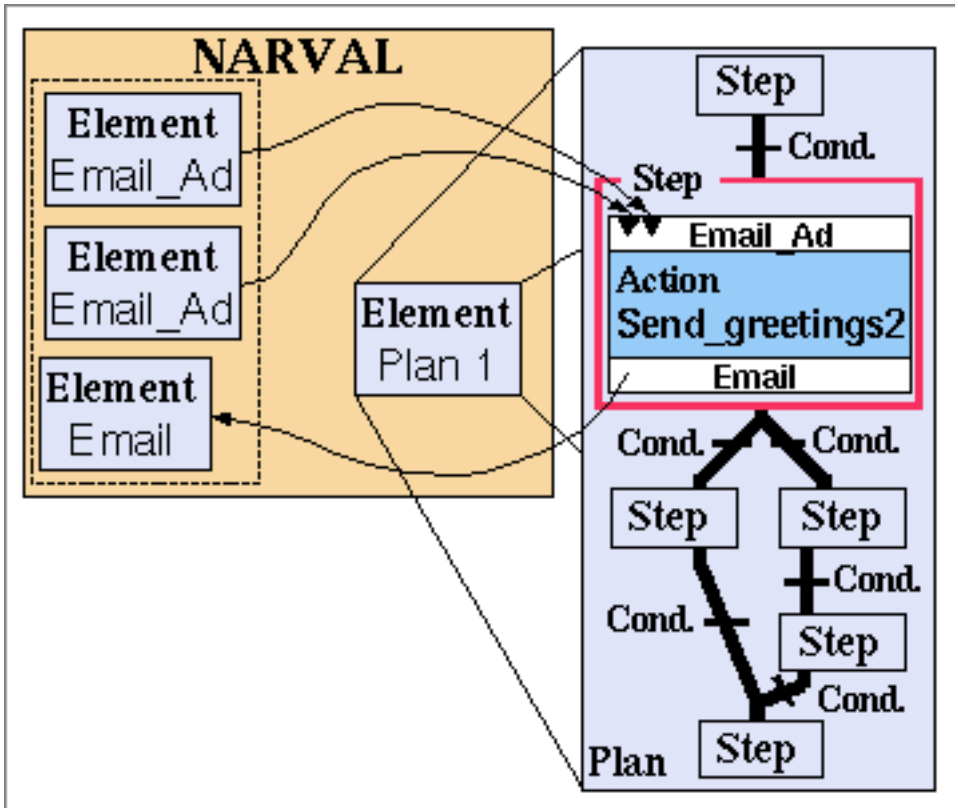


**Figure 7 -** Action with a list of elements input

### 5.1.4. Optional and mandatory inputs

It is also possible for an action to take optional inputs. Optional inputs are not required for the execution of the step, so if a matching element is not found for the step, this will not prevent the step from executing properly. This is not the case for mandatory inputs (by default if not specified otherwise), and if no element can be found that matches a mandatory input, the action will not be executed and the plan will be ended with an error.

In the example of Figure 8, action `Send_greetings3` has an optional `Signature` input. If no `Signature` element is present in memory, the action will be run anyway.

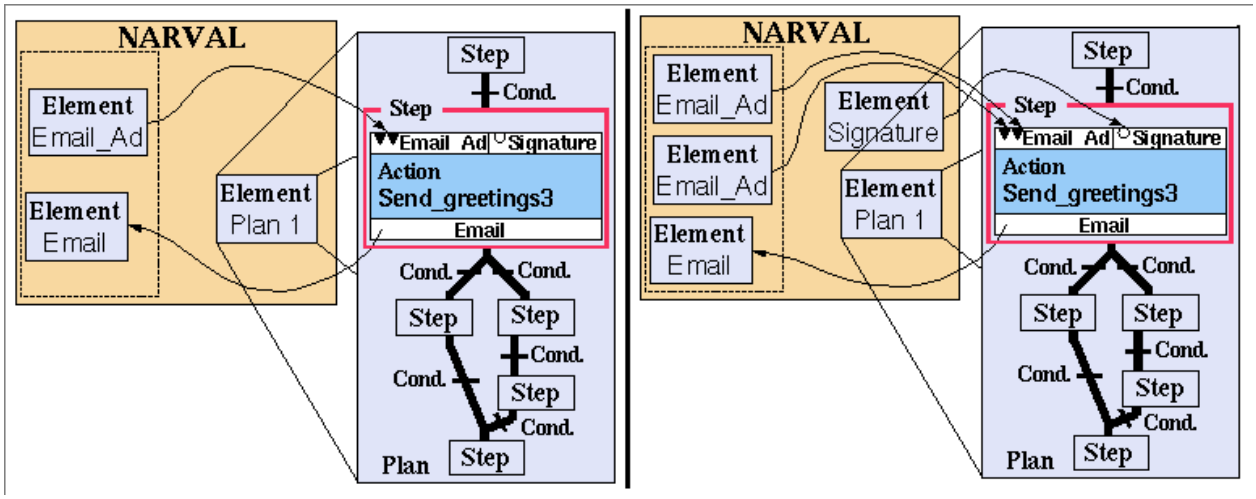


**Figure 8 -** Action optional input

### 5.1.5. Arguments of an action

It is also possible to specify arguments for an action explicitly in a recipe. As for elements in memory, an argument is an element that can be used by an action when it is run. The main difference is that the element is statically specified in the recipe instead of having been created dynamically at run time. If the argument does not match any input, it is not used.

In the example of Figure 9, action `Send_greetings4` accepts a `Text` element containing the greeting to be sent in the email. The step has an argument providing this element explicitly. This provides a facility for using a generic action that is specialized when the recipe is written. For instance, in this example, the same action can be used to send greetings in English or in French.

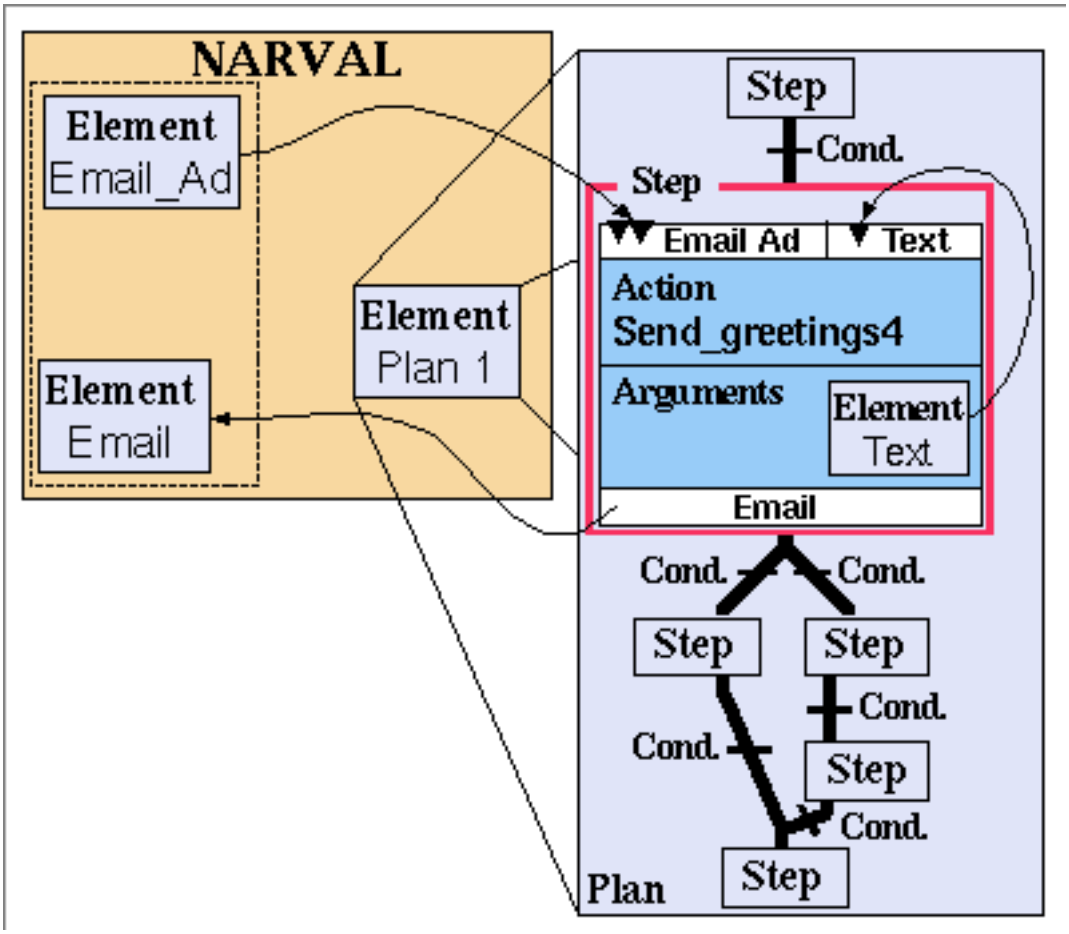


**Figure 9 -** Step with an argument

### 5.1.6. Constraining the prototype in the Step

It is possible to constrain the prototype of an action in the step, using the `id` attribute provided by the action's inputs prototype. The description of the elements can be precised by adding `match` statements, and some attributes of the input can be overriden, depending on the value of the attribute in the action prototype, as shown in the table below.

| attribute | value in action | overridable in step |
|---|---|---|
| optional | yes | yes |
| | no | no |
| list | yes | yes |
| | no | no |
| use | yes | no |
| | no | yes |
| outdates | yes | no |
| | no | yes |

**Table 1 -** attribute precedence

This is a very powerful feature, that can be used, for instance, to provide template actions, which can operate on different types of argument. Which kind of argument precisely is specified in the recipe as the action is embedded in a step.

## 5.2. Getting inputs before running an action

When Narval is about to run an action step, it first gathers all the elements potentially acceptable by the action's inputs. Then the elements are assigned to each input according to a number of priorities: explicit arguments to the step have higher priority than elements that have validated the incoming transition, which have a higher priority than elements produced by an input step of that transition, which have a higher priority than elements in the context of the plan and so on with all the nested contexts up to the memory itself. This ordering enables to privilege elements that are close to the step.

| Rank | Element origin |
|---|---|
| 1 | action argument and step context |
| 2 | elements having validated a condition of the transition that has lead to the step. |
| 3 | elements produced by an incoming step of that transition |
| 4 | context of the step's plan (elements in memory belonging to the context of that plan) |
| 5 | context of the caller plan (this can be iterated) |
| 6 | global memory |

**Table 2 -** Priorities given to elements according to their origin when assigning action inputs

The *from_context* and *to_context* attributes of an input can be used to control the memory area (from the parent step to the whole memory) from which elements will selected for the input. For instance, if you set from_context to "plan", the elements lookup will start from the step's plan, skipping step 1 to 3 of the above table. In the same way, if you set to_context to "plan", the elements lookup will stop to the step's plan, skipping step 5 and 6. Available values for those attributes are "step", "plan", "parent_plan" and "memory"

Keep in mind that a given element can be assigned to only one input.

If the input can be a list of elements, the list is built with elements having the same priority, for instance only elements having validated the transition leading to the step, or only elements coming from the context of the plan. It is therefore impossible to build a list with elements coming, for example, from the context of the action, from the transition and the context of the plan. The list with the higher priority is used.

If the step has a *foreach* attribute (see Section 4.2 “*Special behaviour: the foreach attribute*” - Chapter I) on a given element type, Narval tries to assign all the matching elements to a single input. If this is not possible, Narval stops the execution of the plan and produces an error element (see below).

If after input assignment one of the mandatory inputs is unmatched, Narval stops the plan execution and produces an `error` element in memory. This element describes the error (which plan was running, which action was being prepared and which input was missing). This is of course not the case for optional inputs.

## 5.3. Executing the action and fetching the outputs

After having checked that all inputs are available, Narval launches the action using the selected elements as an arguments. The execution of the action can, of course, use several external programs. When the execution is over, Narval checks that the elements returned match the expected outputs described in the action prototype. If an unexpected element is found or if one output is matched by several elements Narval terminates the plan and creates an `error` element in memory. This element describes the running plan and step, and which output caused the problem.

## 5.4. Handling action generated errors

Errors can occur during action execution, for instance a file that the action is supposed to read can be missing. However, we sometimes wish to handle such situations within the plan, and avoid aborting the execution. This is possible if the action produces an `error` element itself. In that case, Narval checks for the existence of specific transition dedicated to error handling (see Section 6.2 “*Behaviour of transition*” - Chapter I). If such a transition is found in the step's outgoing transitions, and its conditions are matched, it is fired immediately and the execution goes on. Otherwise, the plan is aborted as described above.

## 5.5. Further execution of the plan

The elements produced as outputs by the step are submitted to following transitions, so that their conditions can be evaluated.

# 6. Evaluating transitions

In a plan, the transitions control the execution flow. Figure 10 shows a recipe with several transitions. Some have only one incoming step, and only one outgoing step, others have several incoming steps or several outgoing steps, or both. In order for a transition with several incoming steps to be fired, all the steps must have been successfully executed. When a transition has several outgoing steps, the execution of all the steps is parallelized.

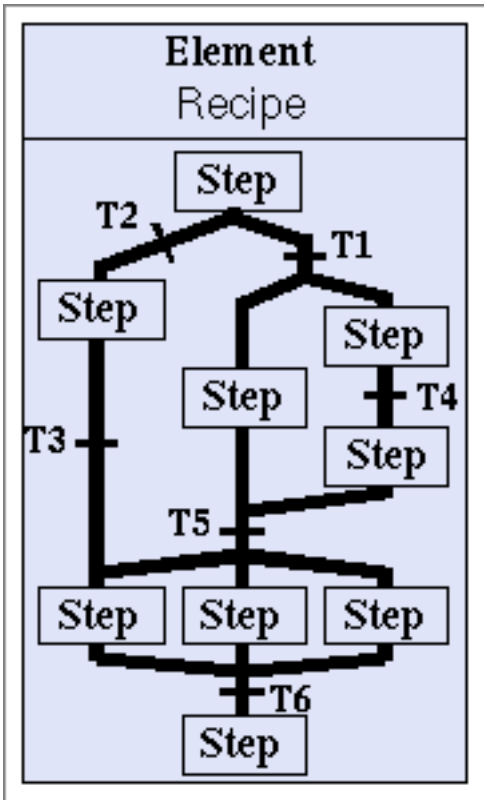


**Figure 10** - General presentation of transitions in a recipe

## 6.1. Conditions in a transition

Each transition can bear one or more conditions allowing to choose which transition will be fired, and thus what will be the execution path of the plan, according to the elements produced by the steps that where formerly executed, and the content of the global memory.

A condition checks the presence in memory of an element having a number of specificities. It becomes true, thus giving a chance to the transition to be fired, only if such an element exists. On Figure 11, one of the transitions checks the existence of an element of type `Elt1` and the other one checks for an element of type `Elt2`. Since there is only an `Elt1` element in memory, only the former transition can be fired, and its outgoing steps will be executed. If two transitions can be fired at the same time, the transition with the highest priority is fired (see Section 6.5 "*Priorities*" - Chapter I). In case of a tie, the behaviour is unspecified. Such a situation can be avoided by choosing carefully the conditions of the transitions and by assigning different priorities to transitions that may be simultaneously fireable.

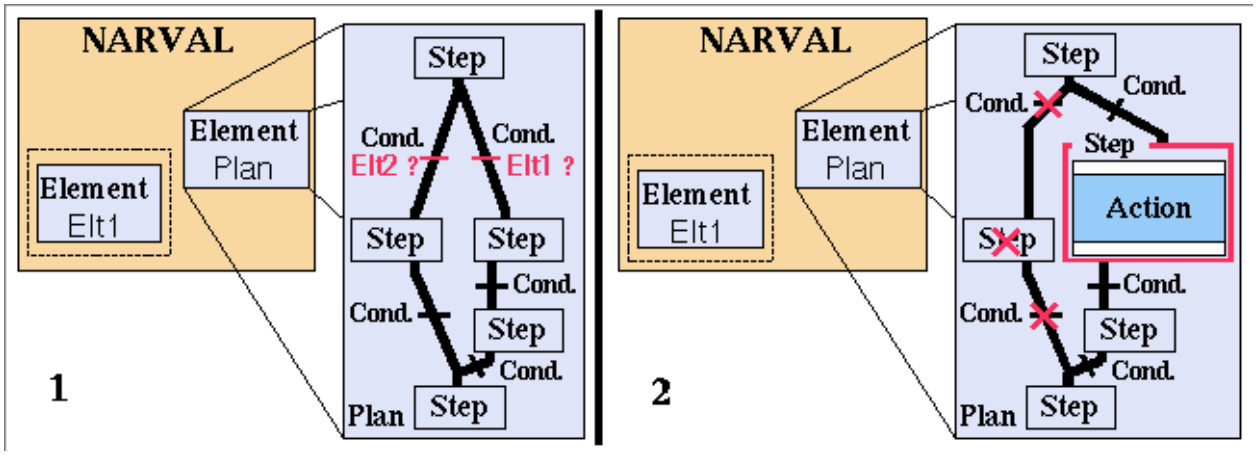


**Figure 11 -** Handling of a transition's conditions by Narval

A transition can have several conditions. Each condition checks for the existence of a different element. Therefore, a transition with three conditions can be fired only if its three conditions are matched by three different elements.

## 6.2. Behaviour of transition

The incoming links of a transition can optionally be flagged as error handling links.

Normally this flag is not set. In this case, the transition can be fired only if the step connected to this link has been successfully executed and all the conditions of the transition are satisfied.

If one of the input links is flagged as error handling, the transition can be fired only if the step connected to this link has produced an `error` element during its execution, and, of course, if all the conditions of the transition are matched.

In Figure 12, on the left panel, an action has generated an `Error` element (see Section 5.4 *"Handling action generated errors"* - Chapter I). This causes the transition on the error handling link to be activated, regardless of other transitions. On the left panel, since no error has occurred in the step, the normal link is activated, and the other transition is evaluated.

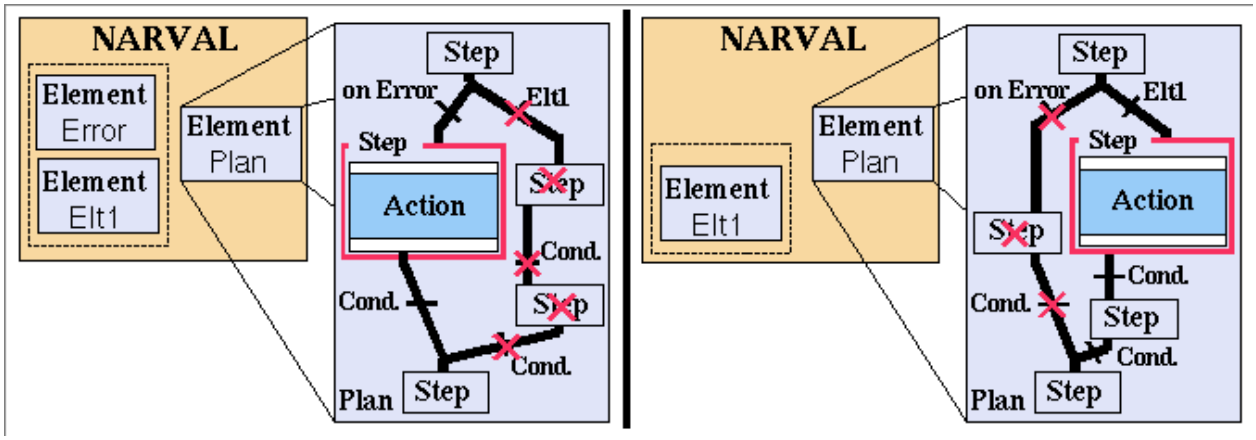


**Figure 12 -** Transition behaviour (standard or error handling)

## 6.3. Used and consulted elements

Just as for action inputs (see Section 5.1.1 "*Used and consulted inputs*" - Chapter I), it is possible to specify that a condition will 'use' its matching element, so that the same element will not be able to match a condition of the transition more than once. For instance, if the first transition of a plan can be fired if an `email` element is available, and this element is used by the condition, the plan will be run only once for each email.

## 6.4. Context of a transition

Just as for steps (see Section 5.2 "*Getting inputs before running an action*" - Chapter I), it is possible to specify a context for a transition condition. This forces Narval to look for elements in a given memory area, either the global memory or the context of the running plan, and to use these elements first to match the condition. (see Section 6.6 "*Using elements to evaluate transitions*" - Chapter I).

## 6.5. Priorities

When several conditions are available after a step, a priority can be set on these transitions, so that Narval can break ties when selecting which transition to fire (the one with the highest priority is used). This enables having a default condition that will be fired if no other transition can be fired, or to solve situations where several transitions could be fired simultaneously.

## 6.6. Using elements to evaluate transitions

When Narval must evaluate a transition, it first gathers all the elements that match each condition. Then it assigns an element to each condition, using the same algorithm as for action input assignment (see Section 5.2 "*Getting inputs before running an action*" - Chapter I). This privileges the elements that are closer to the transition.

| Assignment order | Element location |
|---|---|
| 1 | transition context |
| 2 | elements produced by incoming steps |
| 3 | current plan memory context |
| 4 | context of caller plans |
| 5 | global memory |

**Table 3 -** Selection order for element condition assignment in transitions

An element can be assigned to at most one condition. If after condition assignment, a condition is still not matched, the transition is not fireable. If no transition in the plan is fireable, execution of the plan is suspended until another plan adds a new element in the memory that makes a transition fireable, and thus enables the plan execution to be resumed.

## 6.7. Further execution of the plan

When a transition is fired, Narval prepares the outgoing steps for execution and sends them the ele-

ments having matched the transition conditions.

## 7. Element selection and condition evaluation

When an action is executed, Narval checks that the input elements given to the action and the output elements produced by the action match the action prototype. To evaluate a transition, Narval looks in memory for elements matching the description given in the condition. Similarly, to build the context of a step or of a transition, or to implement the foreach evaluation of a step, Narval selects elements in memory. In all cases, the same matching algorithm is used: only elements having at least the required patterns are eligible. For instance, the transition leading to an email forwarding action step lets through only mails sent by M. Dupont, without considering the email subject or body. This is done by writing a condition saying that we want an `email` element having the `sender="M. Dupont"` property. Each condition can specify several properties of an element and several conditions specify different elements. This is a very powerful tool to specify constraints pertaining to the execution flow of the steps, by restraining the prototype of the actions.

*Chapter II*

# Chosen representations and used techniques

Narval uses numerous techniques lying on the XML data description language and the Python programming language. The following sections will describe Narval majors implementation choices.

## 1. Description of the various elements

### 1.1. Description of a recipe

A recipe is described using an XML tree. This tree contains a root node (the `cookbook` node) and below several child nodes. Each node can have attributes. The recipe XML tree is described below. A real recipe might, of course, have several `step` nodes and several `transition` nodes. Node ordering is unimportant.

| **recipe** |
|---|
| Recipes can appear as a top level element in Narval's memory. When serialized, they are usually stored in `cookbook` Nodes (this is only a convention though). |

| **Content model** |
|---|
| step+,transition* |
| A recipe may consist of a single `step` element, which will be both the start and end step. In most cases, more than one steps will be used, and these steps will be connected by `transition` elements. The order of the child nodes is not important. |

| **Attributes** | |
|---|---|
| group | Mandatory. The name of the recipe group. This relates the recipe to a cookbook, and thus provides a namespace for the recipe, in which we are sure that no other recipe will have the same name. |
| name | Mandatory. The name of the recipe itself. When used as the target of a step or a start plan command, the recipe will be refered to as 'group.name'. |
| restart | Optional, defaults to 'no'. Should be 'yes' if plans instanciated from this recipe should restart after the end step has been completed. |
| decay | Optional. Gives the number of seconds after which plans instantiated from this recipe will be forgotten |

**Table 4 -** Description of recipe nodes

| **step** |
|---|
| Steps encapsulate the behavioural units in a recipe, which can be `actions`, `transformations` or other `recipes`. |

| **Content model** |
|---|
| arguments?,input*,output* |
| Comments: a basic step is empty. If the target is an action or a transformation, explicit arguments can be provided in an `argument` node; the |

| Content model |
| --- |
| `context` child node can be used to specify the context in which the arguments will be fetched; `input` and `output` nodes can be used to provide restrictions to the prototype of the action or transformation. The order of the child nodes is not important. |

| Attributes | |
| --- | --- |
| id | Mandatory. The identifier of the step. Must be unique within a recipe |
| type | Mandatory. The type of the target of the step. Must be one of 'recipe', 'action' |
| target | Mandatory. The name of the target of the step, generally of the form 'group.name'. |
| foreach | Optional. See Section 4.2 *"Special behaviour: the foreach attribute"* - Chapter I for more information |

**Table 5 -** Description of step nodes

| transition |
| --- |
| Transitions are used to control the execution flow in the recipe. They can express conditional execution, or synchronization |

| Content model |
| --- |
| in+,out+,condition,time?,context? |
| |

| Attributes | |
| --- | --- |
| id | Mandatory. The identifier of the transition. Must be unique within a recipe |
| priority | Optional, defaults to 0. Used to break ties when determining which transition should be fired: the one with the highest priority is used. |
| state | This attribute is only valid for transitions in `plan` elements. It specifies the current state of the transition. Possible values are wait-step, wait-time, wait-condition, fireable, fired, fail, impossible |

**Table 6 -** Description of transition nodes

| condition |
| --- |
| A condition is a set of `match` nodes, that should all be matched by the same element in order for the condition to be true. |

| Content model |
| --- |
| match+ |

| Attributes | |
| --- | --- |
| use | optional, defaults to no. Possible value are yes, no. If use is yes, then an element will only be able to match the condition once. |
| from_context | Optional. The context from which elements lookup should start. |
| to_context | Optional. The context where elements lookup should stop. |

**Table 7 -** Description of condition nodes

| match |
| --- |
| a `match` describes a number of elements using an python expression |

| Content model |
| --- |
| #PCDATA |
| The python expression describing the matched elements. Elements for which the expresion evaluates to true, a non empty list of nodes, a non empty string or a non zero numeric value are matching elements. The expression should use the special *elmt* identifier, which represent the element currently being tested (actually the expression is ran against avery matchable element). |

**Table 8 -** Description of match nodes

| time |
| --- |
| |

| Content model |
| --- |
| EMPTY |

| Attributes | |
| --- | --- |
| seconds | min = 0 max = 59 |
| minutes | min = 0 max = 59 |
| hours | min = 0 max = 23 |
| monthdays | min = 1 max = 31 |
| months | min = 1 max = 12 |
| years | min = -10000 max = 10000 |
| yeardays | min = 1 max = 366 |
| weekdays | min = 0 max = 6 (0 is monday) |

**Table 9 -** Description of time nodes

| in |
| --- |
| an `in` node has a reference to a step in a plan or recipe. |

| Content model |
| --- |
| EMPTY |

| Attributes | |
| --- | --- |
| idref | Mandatory. The identifier of the step. |

**Table 10 -** Description of in nodes

| out |
| --- |
| an `out` node has a reference to a step in a plan or recipe. |

| Content model |
| --- |
| EMPTY |

| Attributes | |
| --- | --- |
| idref | Mandatory. The identifier of the step. |

**Table 11 -** Description of out nodes

Recipes are specified in XML files stored on the disk. When starting Narval, these files are read and recipes are stored in the memory.

## 1.2. Description of a plan

A plan looks like a recipe. It is also described as an XML tree. Plans have additional attributes allowing plan execution control. The plan tree is described below.

| plan |
| --- |
| Plans can appear as a top level element in Narval's memory. Plans are instances of recipes, that Narval is able to run. |

| Content model |
| --- |
| step+,transition*,elements |
| The steps and transitions are initially copied from the recipe from which the plan was instanciated. The `elements` node is used to store references to the elements used by the plan, which make up the context of the plan. Only Narval can create plans. |

| Attributes | |
| --- | --- |
| recipename | Mandatory. The name of the recipe from which the plan was instanciated. |
| start_step | Mandatory. The value of this attribute should be the id of the start step of the plan |
| end_step | Mandatory. The value of this attribute should be the id of the final step of the plan |
| restart | Optional, defaults to 'no'. Should be 'yes' if the plan should restart after the end step has been completed. |
| decay | Optional. Gives the amount of time after which the will be forgotten |
| eid | Optional. Element identifier in Narval's memory. |
| parent_plan | Optional. The eid of the plan of in which the current plan is embedded as a step. |
| parent_step | Optional. The id of the step of which the current plan is the target. |
| state | Specifies the current state of the plan. Possible values are ready, running, failed, end, failed-end, done |

**Table 12 -** Description of plan nodes

Plans don't have to be specified in XML files as Narval builds them in memory from the recipes. However, Narval can save the content of its memory in a file. Therefore, plans might appear in this file.

## 1.3. Notion of module. Description of the actions.

Actions are grouped in modules. A module is a Python file containing the actions code. In this file, there is also an XML tree describing all the actions of the module and their prototype (i.e. their inputs and their outputs). This tree is described below.

| module |
| --- |
| |

| Content model |
| --- |
| action* |

| Attributes | |
| --- | --- |
| name | Optional. The name of the module. If provided, must be the name of the python file (without the .py extention) |
| version | Optional. |

**Table 13 -** Description of module nodes

| action |
| --- |
| |

| Content model |
| --- |
| description*,input*,output* |
| The input and output nodes are the prototype of the action. |

| Attributes | |
| --- | --- |
| name | Mandatory. The name of the action. |
| group | Optional. If provided, must be the name of module |
| func | Mandatory. The name of the python function implementing the action |
| eid | Optional. Element identifier in Narval's memory. This is an internal attribute that should not be set when writing a module |

**Table 14 -** Description of action nodes

| description |
| --- |
| |

| Content model |
| --- |
| #PCDATA |

| Attributes | |
|---|---|
| lang | Mandatory. The iso notation for the language of the description |

**Table 15 -** Description of description nodes

When Narval executes a plan and has to run an action, it firstly gets the action name in the plan XML tree. This name is compound of the module name followed by the action name (for example, `Email.catch_new` refers to the `catch_new` action of the `Email` module). Narval loads the corresponding module and runs the Python function associated with the action (in previous example, Narval loads `Email.py` Python module).

The behaviour of the modules is further described in the modules programmer handbook.

## 1.4. Description of transform elements

Transforms in Narval conform to the XSLT specification. In order to be processed by Narval, tough, they must include some information about their prototype in a prototype node, which has to be a child of the root node of the transformation. Since this node is not in the XSLT namespace, it will be ignored by the tranformation engine.

| prototype |
|---|
| |

| Content model |
|---|
| description*,input*,output* |

**Table 16 -** Description of prototype nodes

More information is available in the Module Programmer Handbook.

# 2. Memory structure

## 2.1. Internal structure of the memory

Narval memory is a simple set of elements, that should be exportable as an XML tree. The various elements are attached to the root node (`memory`). Each element in the memory has got an id number called `eid`.

## 2.2. Memory initialization

When Narval starts, it fills its memory using the `$NARVAL_HOME/data/memory.xml`. This XML file contains a tree representing the initial memory. It might contain general data such as the user's name, the user's email address, his electronic mailbox, etc.

This file may also contains `start-plan` elements permitting recipes instantiation and plans starting when initializing Narval.

During its initialization, Narval loads the recipes in its memory. The recipes are described in XML files located in the `$NARVAL_HOME/recipes/` or in the `$NARVAL_SHARE/recipes/` directories.

## 3. Conditions expression and elements selection

As described above, the elements selection and the conditions evaluation are computed thanks to an unique matching algorithm. Such as algorithm searches the memory for elements corresponding to a pattern. Each pattern is described in a `match` node (in the actions prototypes). Inserting several nodes allows the description of several patterns and, thus, the selection of elements of different kinds. Inside the `match` node, the pattern definition is expressed using the Python language.

For instance, a condition waiting for an `email` element whose subject is `Hi there` is expressed as follow: `IEmail(elmt).subject == "Hi there"`. As Python syntax is very powerful, conditions might be much more complicated.

## 4. Evaluation of the fireability of a transition

When Narval evaluates the transitions of a plan in order to know which ones are fireable and to decide which one will be fired, it first classifies the transitions in three groups: the transitions that are impossible to fire, the undetermined transitions and the potentially fireable transitions.

The impossible-to-fire transitions are the ones with an input step that has failed or has already been used by a previous evaluation (step in the `failed` state or the `history` state). A transition flagged as error handling that has a correctly executed input step or a transition not flagged as error handling that has a failed input step are also impossible to fired.

The undetermined transitions are the ones with an input step being still executed (which does not allow knowing the result of the step).

The potentially fireable transitions are the other ones. The conditions of each of these transitions must be checked in order to know if it can be fired.

If all the transitions of a currently executed plan are impossible to fire, the plan fails and an `error` element is set in the memory. If a transition flagged as error handling is potentially fireable but can't be fired because of unsatisfied conditions, the plan also fails. In the other cases, the plan execution continues. Further execution can be immediate thanks to a potentially-fireable transition having all its conditions satisfied, or postponed as the plan waits for its undetermined transition to become impossible or potentially fireable, or for its potentially fireable transitions to have their conditions satisfied.

Whatever could be the state of the other transitions, as a transition is fireable, it is fired.

*Chapter III*

# Known Bugs

A web page covering the known bugs of Narval should be available very soon now, if it is not already there.

*Chapter IV*

# Conclusion

This documentation provides a skipping through of Narval code, exposing the main notions, describing the software and explaining major design choices. The interested reader should read now the source code of Narval, that is carefully commented, allowing the understanding of its exact detailed behaviour.

# Glossary

## Narval

Action

Conceptually elementary transformation that Narval can do with *elements*.

Arguments

Set of fixed *elements* used by an *action*.

Element

Generic representation of an object: all entities manipulated by Narval are Elements. It may be an *action*, an email, a Web page, a *plan* or a *recipe*. Technicaly, elements are applicative computing entity found in *memory*.

Error handling

See *transition*

Memory

Storage place of the *elements* in which they can be accessed by the *actions*.

Context

The context of a *plan* is the set of *elements* handled by the plan, i.e. all the elements created or used by the *steps* of the plan or that triggered the *transitions* of the plan.

Plan

Instance of a *recipe* allowing its execution.

Recipe

Sequence of *steps* linked by *transitions*, describing a functionality of Narval.

Repetition

See *step*

Step

Basic brick of a *recipe* that can be either an *action* or an other *recipe*.

A *step* can have a *repetition* behaviour: it is then executed in parallel as much as necessary in order to compute the *element* set on which the *repetition* is done (the *step* is said to have a *repetition* behaviour on these *elements* type).

Transition

Link between a set of origin *steps* and a set of destination *steps*, that can have a condition on *elements* found in *memory*. Each of the input can be flagged as *error handling*. If not set, the *step* linked with this input must be correctly executed to have the *transition* fireable. If set, the *step* linked with this input must generate an `error` *element* to have the *transition* fireable.

## Computing Languages used in Narval

Python

Programming language. See http://www.python.org/ [http://www.python.org/].

eXtensible Markup Language (XML)

Data tagging language defined by the W3C. See http://www.w3c.org/XML/ [http://www.w3c.org/XML/].