



Narval Technical Manual

Narval Technical Manual

by Olivier Cayrol and Alexandre Fayolle

Copyright © 2000-2001 by Logilab

Legal Notice

Copyright © 2000-2001 by Logilab.

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/> [<http://www.opencontent.org/openpub/>]).

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.

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.

Table of Contents

Warning.....	1
1. Narval Operation.....	2
1.1. The memory: notion of context nesting.....	2
1.2. Difference between recipes and plans.....	3
1.2.1. Introduction to recipes.....	3
1.2.2. Plan execution.....	3
1.3. Running a step.....	4
1.3.1. General behaviour.....	4
1.3.2. Special behaviour: the <i>foreach</i> attribute.....	4
1.3.3. Context of a step.....	5
1.4. Running an action or a transition.....	5
1.4.1. Prototype of an action.....	5
1.4.1.1. Used and consulted inputs.....	6
1.4.1.2. Outdated inputs.....	7
1.4.1.3. Single element or list of elements.....	7
1.4.1.4. Optional and mandatory inputs.....	8
1.4.1.5. Arguments of an action.....	8
1.4.1.6. Constraining the prototype in the Step.....	9
1.4.2. Getting inputs before running an action.....	10
1.4.3. Executing the action and fetching the outputs.....	11
1.4.4. Handling action generated errors.....	11
1.4.5. Further execution of the plan.....	11
1.5. Evaluating transitions.....	11
1.5.1. Conditions in a transition.....	12
1.5.2. Behaviour of transition.....	13
1.5.3. Used and consulted elements.....	14
1.5.4. Context of a transition.....	14
1.5.5. Priorities.....	14
1.5.6. Using elements to evaluate transitions.....	14
1.5.7. Further execution of the plan.....	15
1.6. Element selection and condition evaluation.....	15
2. Chosen representations and used techniques.....	16
2.1. Description of the various elements.....	16
2.1.1. Description of a recipe.....	16
2.1.2. Description of a plan.....	21
2.1.3. Notion of module. Description of the actions.....	23
2.1.4. Description of transform elements.....	25
2.1.5. Description of the other elements.....	25
2.2. Memory structure.....	26
2.2.1. Internal structure of the memory.....	26
2.2.2. Memory initialization.....	27
2.2.3. Building the XML trees in memory.....	27

2.3. Conditions expression and elements selection.....	27
2.4. Evaluation of the fireability of a transition.....	27
3. Known Bugs.....	29
4. Conclusion.....	30
Glossary.....	31

List of Figures

1.1. Narval's memory.....	2
1.2. Building a plan from a recipe.....	3
1.3. Plan execution.....	4
1.4. Foreach attribute in a step.....	5
1.5. Action with a consulted input.....	6
1.6. Action with used inputs.....	7
1.7. Action with a list of elements input.....	7
1.8. Action optional input.....	8
1.9. Step with an argument.....	9
1.10. General presentation of transitions in a recipe.....	12
1.11. Handling of a transition's conditions by Narval.....	13
1.12. Transition behaviour (standard or error handling).....	13

List of Tables

1.1. attribute precedence.....	10
1.2. Priorities given to elements according to their origin when assigning action inputs.....	10
1.3. Selection order for element condition assignment in transitions.....	14
2.3. Description of recipe nodes.....	16
2.4. Description of step nodes.....	17
2.5. Description of transition nodes.....	18
2.6. Description of condition nodes.....	18
2.7. Description of context nodes.....	19
2.8. Description of match nodes.....	19
2.9. Description of time nodes.....	20
2.10. Description of in nodes.....	20
2.11. Description of out nodes.....	21
2.12. Description of plan nodes.....	21
2.13. Description of elements nodes.....	22
2.14. Description of element nodes.....	23
2.15. Description of module nodes.....	23
2.16. Description of action nodes.....	24
2.17. Description of description nodes.....	24
2.18. Description of prototype nodes.....	25

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.

1.2. Difference between recipes and plans

1.2.1. 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 1.2.), 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.

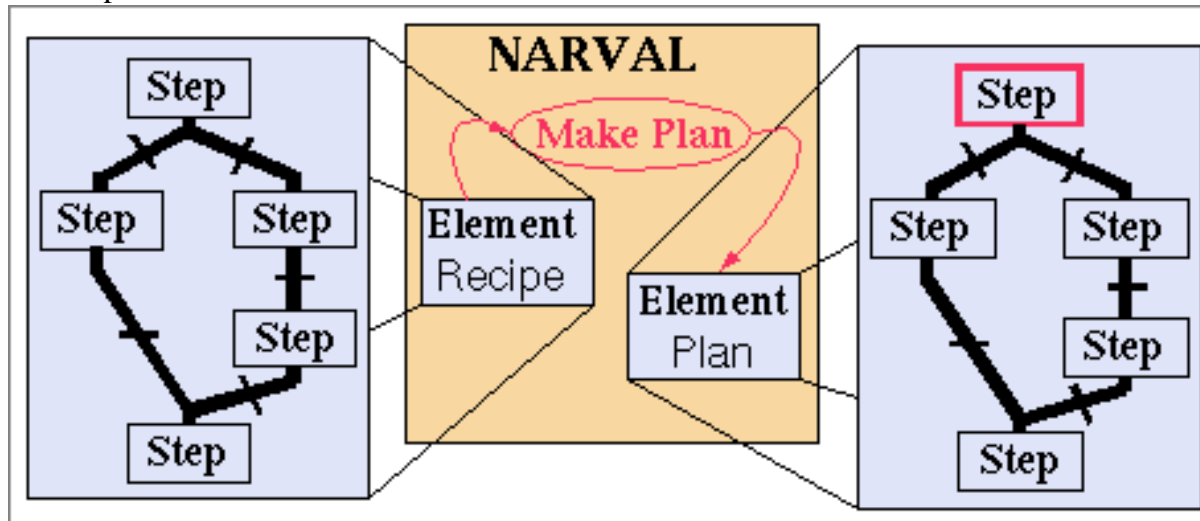


Figure 1.2. Building a plan from a recipe

1.2.2. Plan execution

When executing a plan, Narval starts with the first step (see Section 1.3.). 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 1.3., 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).

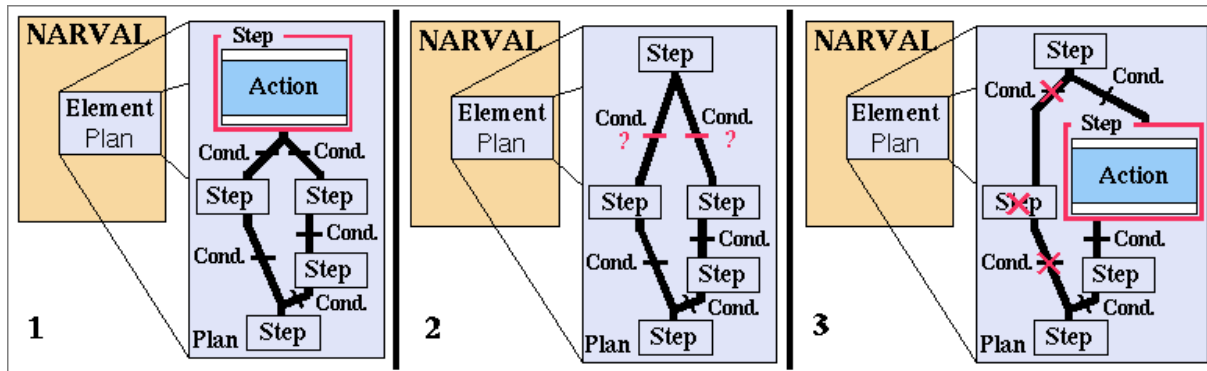


Figure 1.3. Plan execution

1.3. Running a step

Three kinds of steps can exist in a recipe, depending on the target, which can be a recipe, an action or a transformation. However, actions and transformations are handled in a very similar way

1.3.1. 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 1.4.).

1.3.2. Special behaviour: the *foreach* attribute

The *foreach* attribute can only be set on steps whose target is an action or a transformation. It specifies that the action must be run for each element matching a given pattern (given by the attribute value). In the example presented on Figure 1.4.), the `Send_greetings` action takes an email address element as an input (this is stored in an `Email_Ad` type element) and creates a new year greetings email (stored in an `Email` type element). The step representing this action has a *foreach* attribute matching `Email_Ad` elements. 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.

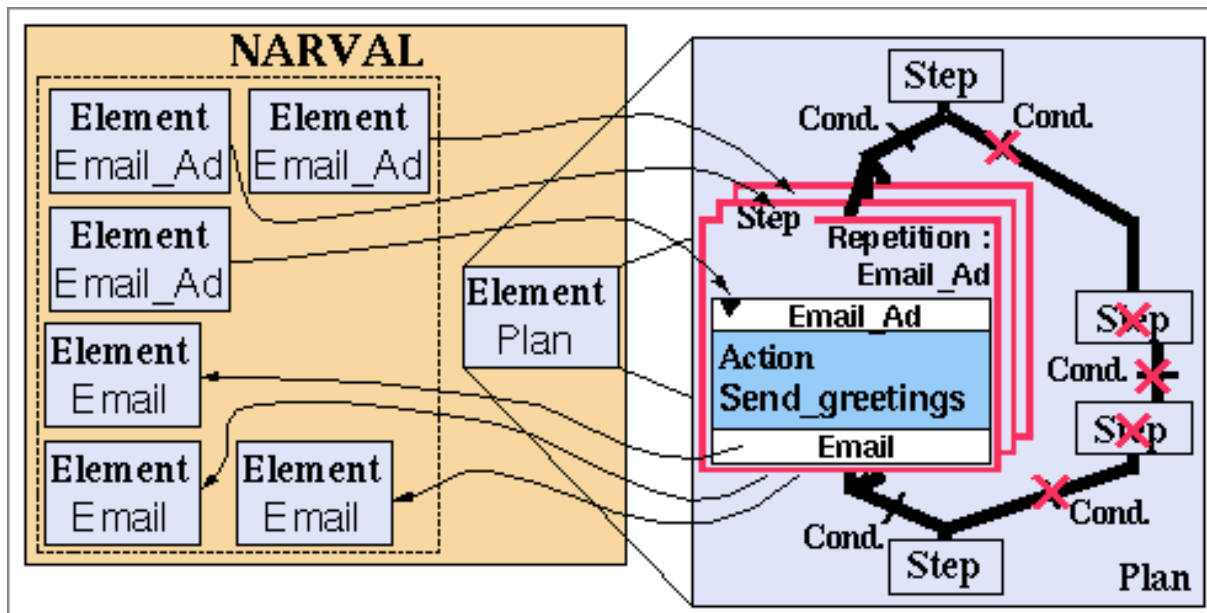


Figure 1.4. Foreach attribute in a step

Important

All the elements matching the foreach attribute value must also match one (and only one) of the action's input (see Section 1.4.1. and Section 1.4.2.).

1.3.3. Context of a step

The context attribute can only be set on action or transformation steps. This attribute specifies a memory area (either the whole memory or the context of the current plan) from which elements will be selected for the action (see Section 1.4.2.).

1.4. Running an action or a transition

Remark

Unless explicitly 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.

1.4.1. 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.

1.4.1.1. 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 1.5., 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.

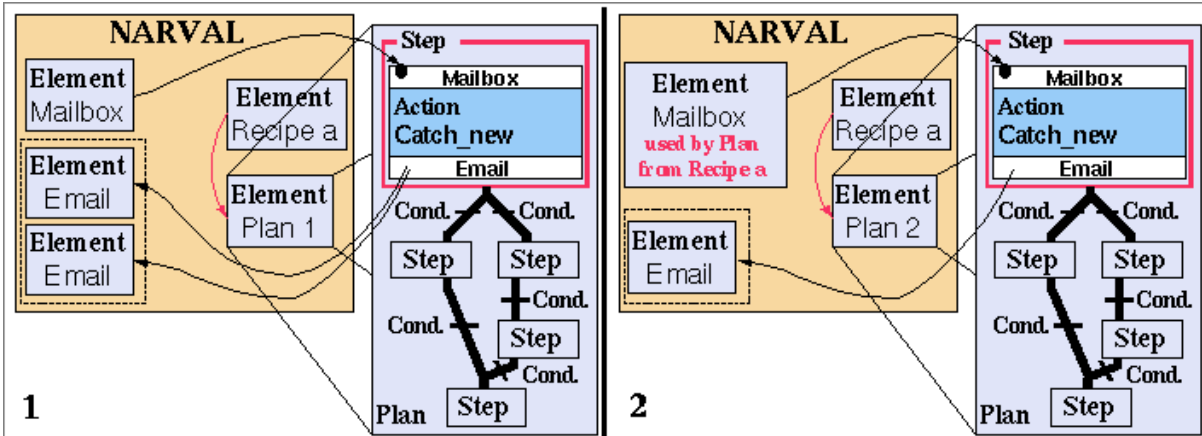


Figure 1.5. Action with a consulted input

On the other hand, in Figure 1.6., 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.

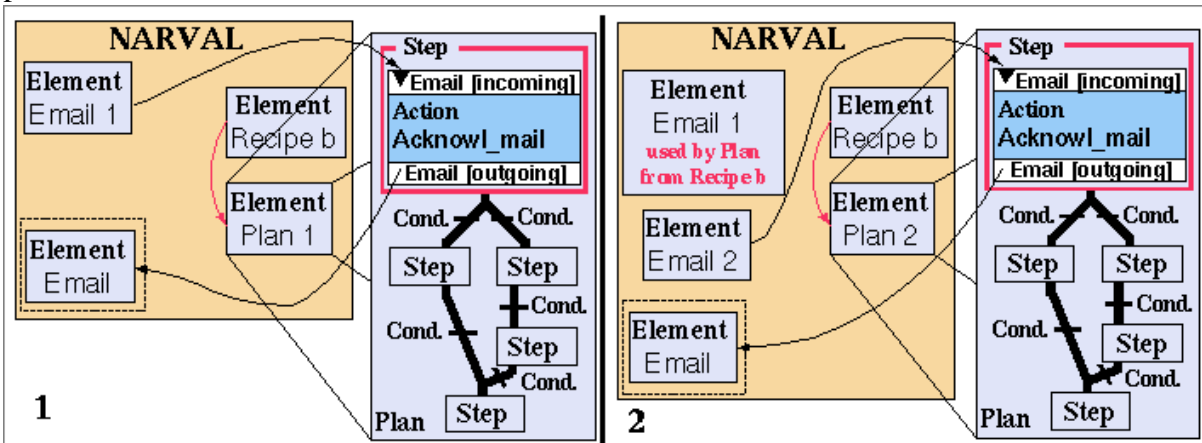


Figure 1.6. Action with used inputs**1.4.1.2. Outdated inputs**

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 outdateding 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 transition 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 be as if the new element replaced the original one.

1.4.1.3. 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 1.7., 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 1.4., 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.

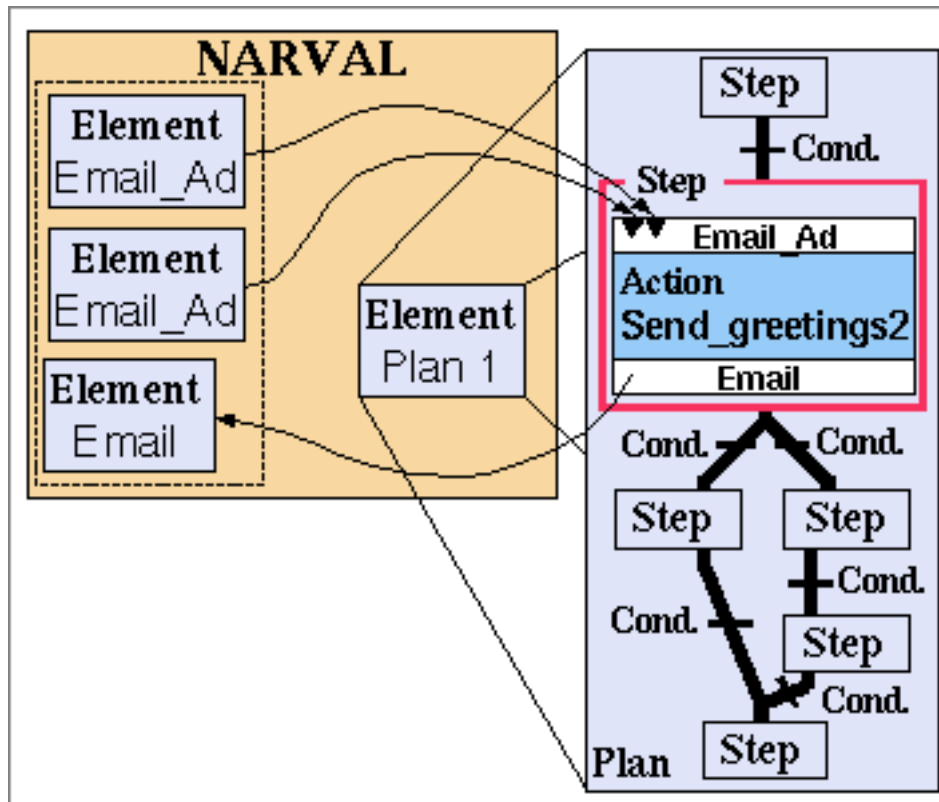


Figure 1.7. Action with a list of elements input

1.4.1.4. 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, 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 1.8., action `Send_greetings3` has an optional `Signature` input. If no `Signature` is present in memory, the action will be run anyway.

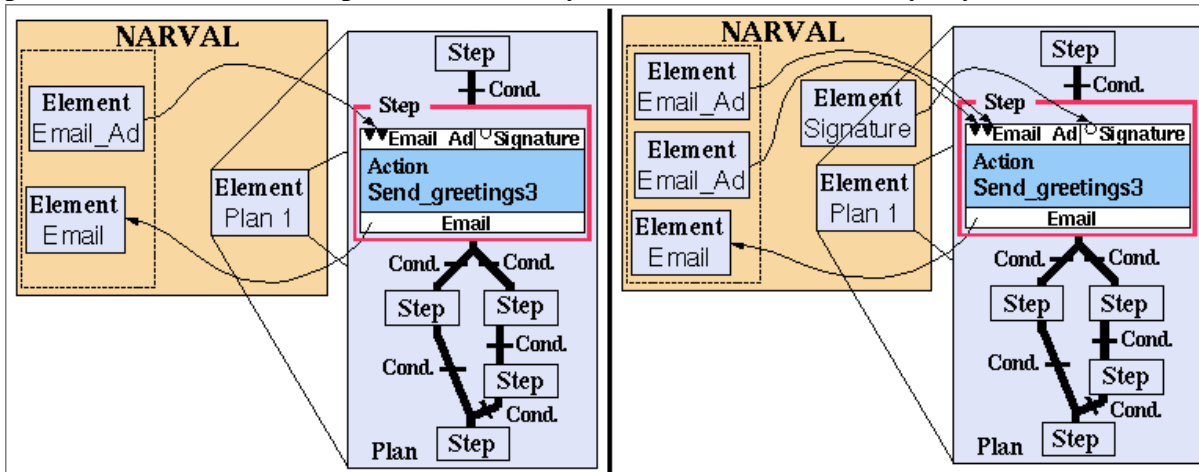


Figure 1.8. Action optional input

1.4.1.5. 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 1.9., 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.

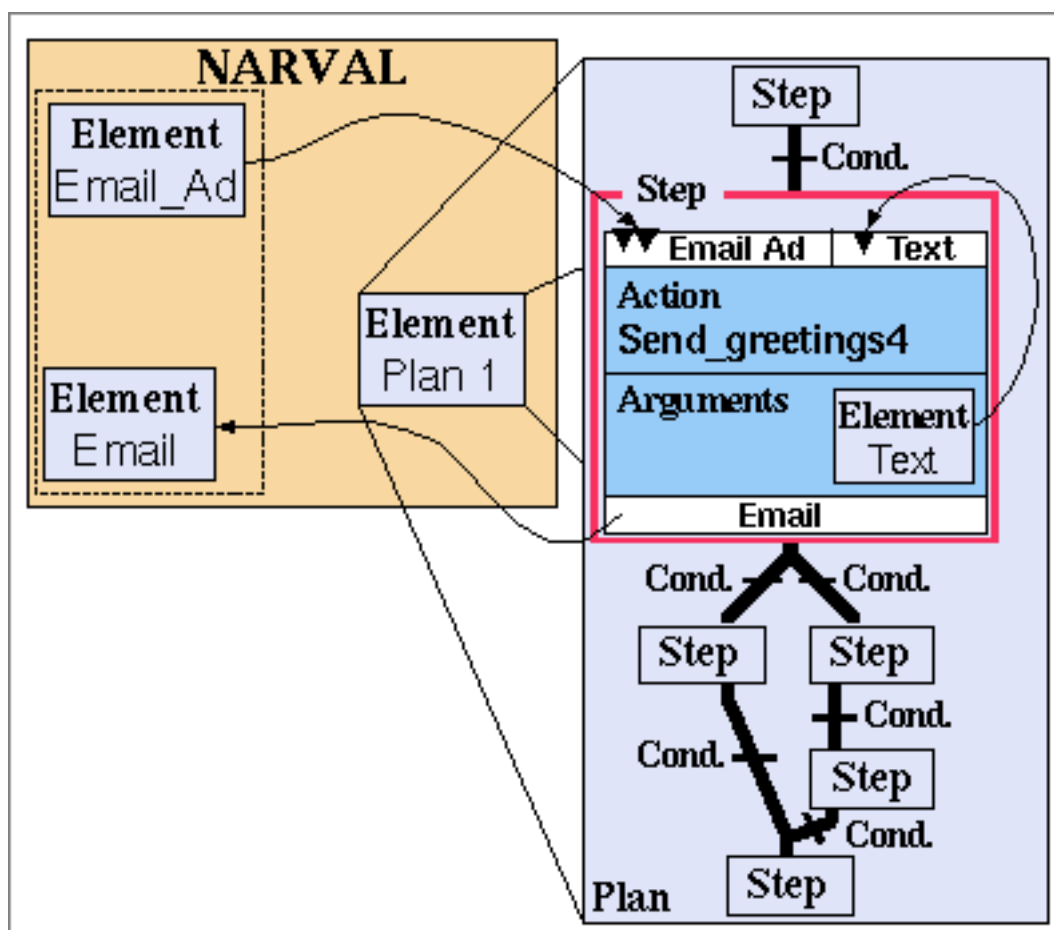


Figure 1.9. Step with an argument

1.4.1.6. Constraining the prototype in the Step

It is possible to constrain the prototype of an action in the step, if the action prototype provided an `id` attribute for its inputs. The description of the elements can be precised by adding match statements, and some attributes of the input can be overridden, 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

attribute	value in action	overridable in step
	no	yes

Table 1.1. 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.

1.4.2. 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 1.2. Priorities given to elements according to their origin when assigning action inputs

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 1.3.2.) 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). *Foreach* attributes must therefore specify a subset of elements potentially acceptable by one of the inputs of the action.

If after input assignment one of the 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.

1.4.3. 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 argument tree. 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.

1.4.4. 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 1.5.2.). 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.

1.4.5. 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.

1.5. Evaluating transitions

In a plan, the transitions control the execution flow. Figure 1.10. 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 in-

coming 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.

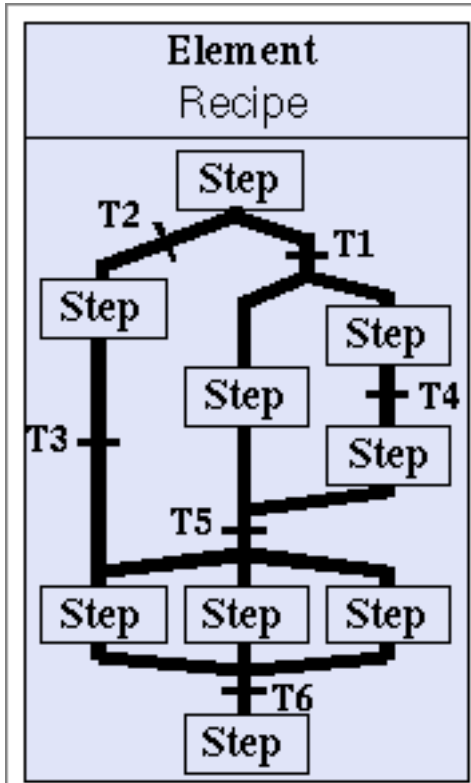


Figure 1.10. General presentation of transitions in a recipe

1.5.1. 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 were 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 1.11., 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 1.5.5.). 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.

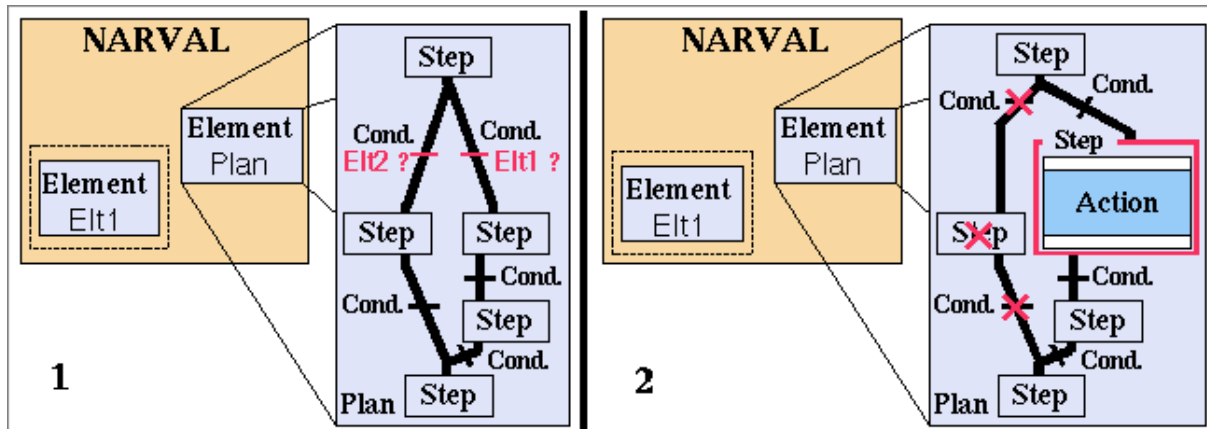


Figure 1.11. Handling of a transition's conditions by Narval

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.

1.5.2. 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 1.12., on the left panel, an action has generated an `Error` element (see Section 1.4.4.). 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.

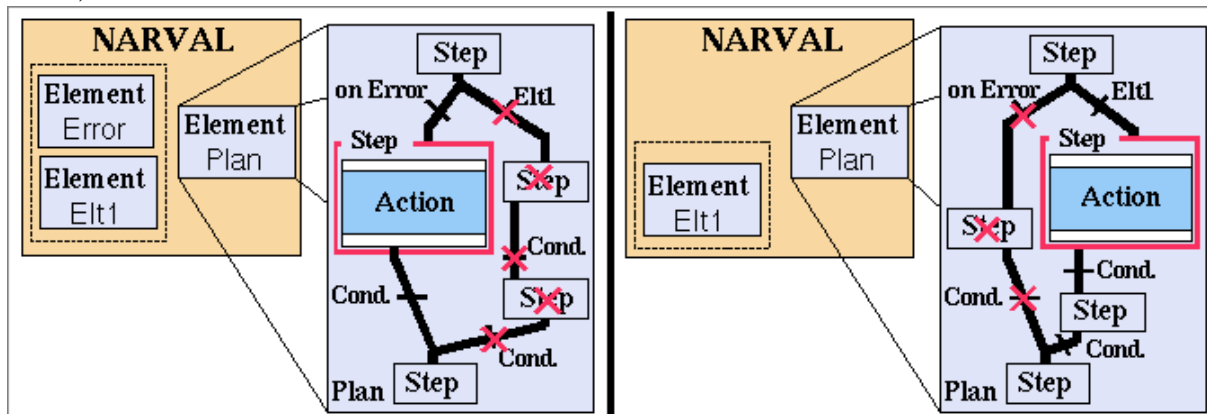


Figure 1.12. Transition behaviour (standard or error handling)

1.5.3. Used and consulted elements

Just as for action inputs (see Section 1.4.1.1.), 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.

1.5.4. Context of a transition

Just as for steps (see Section 1.3.3.), 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 1.5.6.).

1.5.5. 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.

1.5.6. 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 1.4.2.). 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 1.3. 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.

1.5.7. Further execution of the plan

When a transition is fired, Narval prepares the outgoing steps for execution and sends them the elements having matched the transition conditions.

1.6. 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 2. 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.

2.1. Description of the various elements

2.1.1. 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 <code>cookbook</code> Nodes.	
Content model	
step+,transition*	
A recipe may consist of a single <code>step</code> 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 <code>transition</code> 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 referred to as 'group.name'.

Attributes	
start_step	Mandatory. The value of this attribute should be the id of the start step of the recipe
end_step	Mandatory. The value of this attribute should be the id of the final step of the recipe
restart	Optional, defaults to 'no'. Should be 'yes' if plans instantiated 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
eid	Optional. Element identifier in Narval's memory. This is an internal attribute that should not be set when writing a recipe.

Table 2.3. Description of recipe nodes

step	
Steps encapsulate the behavioural units in a recipe, which can be actions, transformations or other recipes.	
Content model	
arguments?,context?,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 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', 'transform'
target	Mandatory. The name of the target of the step, generally of the form 'group.name'.

Attributes	
foreach	Optional. See Section 1.3.2. for more information
state	This internal attribute is only valid for steps in <code>plan</code> elements. It specifies the current state of the step. Possible values are <code>todo</code> , <code>ready</code> , <code>running</code> , <code>end</code> , <code>done</code> , <code>history</code> , <code>impossible</code> , <code>failed</code> , <code>error</code>

Table 2.4. 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?	
in and out elements point to steps in the recipe, respectively the incoming and outgoing steps of the transition. The condition node specifies the elements that will trigger the transition. time provides transitions that are time-triggered, and context gives information about where to look for elements that could match the conditions.	
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 <code>plan</code> elements. It specifies the current state of the transition. Possible values are <code>wait-step</code> , <code>wait-time</code> , <code>wait-condition</code> , <code>fireable</code> , <code>fired</code> , <code>fail</code> , <code>impossible</code>

Table 2.5. Description of transition nodes

condition
A condition is a set of <code>match</code> 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.

Table 2.6. Description of condition nodes

context	
The context of a step or a transition specifies elements from which Narval should select arguments to a step or triggers to a transition.	
Content model	
match+	
The descriptions of the elements belonging to the context.	
Attributes	
from	Mandatory. Specifies where to look for the elements described by the match nodes. Possible values are memory and plan.

Table 2.7. Description of context nodes

match	
a match describes a number of elements using an XPath expression	
Content model	
#PCDATA	
The XPath expression describing the matched elements. Elements for which the XPath evaluates to true, a non empty list of nodes, a non empty string or a non zero numeric value are matching elements.	

Table 2.8. Description of match nodes

time	
time nodes are used to describe a date, for instance the date when a condition can be fired. The syntax is inspired from crontab entries. Each attribute is a comma separated list of values or '*' (default value). Each value can be an integer between min and max (given below) or a range, such as a-b.	
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 2.9. 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 2.10. 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 2.11. 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.

2.1.2. 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 in-	

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

Table 2.12. Description of plan nodes

elements
an <code>elements</code> node is a container for a list of element nodes.

Content model
element*

Table 2.13. Description of elements nodes

element
an element node has a reference to an element in Narval's memory.
Content model
EMPTY
Attributes
eid
Mandatory. The identifier of the element in memory.

Table 2.14. Description of element 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.

2.1.3. 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
module nodes hold lists of action nodes
Content model
action*
Attributes
name
Optional. The name of the module. If provided, must be the name of the

Attributes	
	python file (without the .py extention)
version	Optional.

Table 2.15. Description of module nodes

action	
action nodes describe a way for Narval to do something, generally using python code.	
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 2.16. Description of action nodes

description
description nodes hold a short description of what an action does. All actions and transformations should provide at least an english description. Beware that if your description uses non-ascii characters, the encoding of the XML document must be set accordingly.
Content model
#PCDATA

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

Table 2.17. 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.

2.1.4. Description of transform elements

Transforms in Narval conform to the XSLT specification. In order to be processed by Narval, though, 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 transformation engine.

prototype
prototype nodes hold the prototype of a transformation.
Content model
description*,input*,output*

Table 2.18. Description of prototype nodes

More information is available in the Module Programmer Handbook.

2.1.5. Description of the other elements

Each element in the memory is described using an XML tree. For example, an email element can be represented as below.

```
<email>
  <to>
    <name>...</name>
    <email_addr>...</email_addr>
  </to>
  <from>
    <name>...</name>
    <email_addr>...</email_addr>
  </from>
  <subject>...</subject>
```

```

<signature>...</signature>
<body
  multipart
>
  <part
    type
  >
  </part>
</body>
</email>

```

- (1) Description of an email element.
- (2) Recipient data.
- (5) Sender data.
- (3) Name of the recipient or the sender.
- (4) Email address of the recipient or the sender.
- (6) Email subject.
- (7) Email signature.
- (8) Email body.
- (9) Specifies the email has several parts (i.e. attached files).
- (10) Email part.
- (11) Type of the part (text or file).

Each new type of elements Narval handles must have an XML description included in the correct DTD¹.

2.2. Memory structure

2.2.1. Internal structure of the memory

Narval memory is organized 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*.

2.2.2. Memory initialization

[1] **Document Type Definition (DTD):** Description of the content of an XML file, in particular the elements and the attributes that might appear in this file.

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 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/recipes/` directory.

2.2.3. Building the XML trees in memory

The various XML trees described above are built in memory either from an XML file during initialization, or dynamically during execution. For building these trees, Narval uses the DOM level 2 programming interface.

2.3. 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 recipes and actions XML trees). 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 XPath language.

For instance, a condition waiting for an `email` element whose subject is *Hi there* is expressed in XPath as follow: `email[@subject="Hi there"]`. As XPath syntax is very powerful, conditions might be much more complicated.

2.4. 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 3. Known Bugs

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

Chapter 4. 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	Applicative computing entity found in memory that can be handled by an action. An email, a Web page, a plan or a recipe are elements.
Error handling	See Transition.
Memory	Storage place of the elements in which they can be accessed by the actions.
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 <code>error</code> element to have the transition fireable.

Computing Languages used in Narval

Document Object Model (DOM)	Interface for accessing XML trees, defined by the W3C. See http://www.w3c.org/DOM/ [http://www.w3c.org/DOM/].
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/].

XML Path Language (XPath)	Nodes selection language used in XML trees, defined by the W3C. See http://www.w3c.org/TR/xpath.html [http://www.w3c.org/TR/xpath.html].
eXtensible Stylesheet Language Transformation (XSLT)	XML trees transformation language, defined by the W3C. See http://www.w3c.org/TR/xslt.html [http://www.w3c.org/TR/xslt.html].