# The Wanda-Server

(Very First Pre-Alpha) Version 0.1

Christian Veenhuis

January 31, 2003

**Christian Veenhuis:**
The Wanda-Server

# Contents

**1 Introduction** **5**
1.1 Structure of the Wanda-Server . . . . . 5

**2 Requirements** **7**

**3 Procedure of a Plug-In Call** **9**

**4 Examples** **11**
4.1 Script-based Plug-in . . . . . 11
4.2 Binary-based Plug-in . . . . . 13
4.2.1 Static Parameter . . . . . 13
4.2.2 Variable Parameter . . . . . 15

**A WPML Reference of all Elements** **19**
A.1 Root Element and Meta Data . . . . . 20
A.1.1 WPML . . . . . 20
A.1.2 description . . . . . 21
A.2 Variables . . . . . 22
A.2.1 vars . . . . . 22
A.2.2 var . . . . . 22
A.3 Invocation of Plug-in . . . . . 22
A.3.1 call . . . . . 22
A.3.2 location . . . . . 23
A.3.3 invocation . . . . . 23
A.3.4 parameter . . . . . 24
A.3.5 use_var . . . . . 24
A.3.6 use_input . . . . . 24
A.3.7 use_output . . . . . 25
A.4 Communication . . . . . 25
A.4.1 protocol . . . . . 25

A.4.2 to_server . . . . . 25
A.4.3 to_client . . . . . 26
A.4.4 transfer_var . . . . . 26
A.4.5 transfer_input . . . . . 27
A.4.6 transfer_output . . . . . 27
A.4.7 call_plugin . . . . . 27
A.5 Error Handling . . . . . 28
A.5.1 errors . . . . . 28
A.5.2 error . . . . . 28

**B WPML Document Type Definition (DTD) 29**

# Chapter 1

# Introduction

The Wanda-Server ('WS') provides the functionality of processing modules ("plug-ins") over a network. Wanda-Clients ('WC') can call these plug-ins by using the WS. A single plug-in provides algorithms or other processing steps of the field of image processing. A single plug-in can be used by a WC by connecting to it over the WS. This way, meta-data like e.g. name, description or the version can be obtained and the plug-in can also be executed. The communication between a WC and the WS as well as the transfer of data between a WC and an appropriate plug-in will be realized by XML documents, as depicted in figure 1.1.

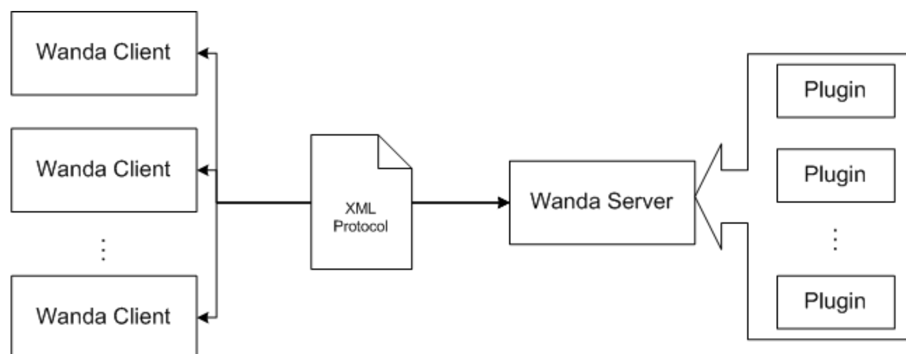


Figure 1.1: Wanda-Server and Wanda-Client

A WC can be any software program which understand the communication protocol using these XML documents. The WS is a software system running on every server within a network (internet/intranet). A WC is a software system which can be used on any computer system e.g. desktop PC's, Mac's, Linux-systems, Laptops, etc.. Thereby also a single computer can be used as client and server, i.e. a WC and a WS can be installed on the same system (e.g. a laptop).

## 1.1 Structure of the Wanda-Server

The WS is a multi-threaded server which can operate on any number of requests from WCs concurrently. Each WC connecting to the WS gets its own thread. By using this thread the whole communication using XML documents can be realized. Thereby, each WC can do any number of request concurrently. This

isn't depicted in figure 1.2 for simplicity. Each thread gets the whole access to all installed plug-ins. Each plug-in is an executable program which can be written in any programming language. Communication with the WS is realized by using arguments as well as input and output files.

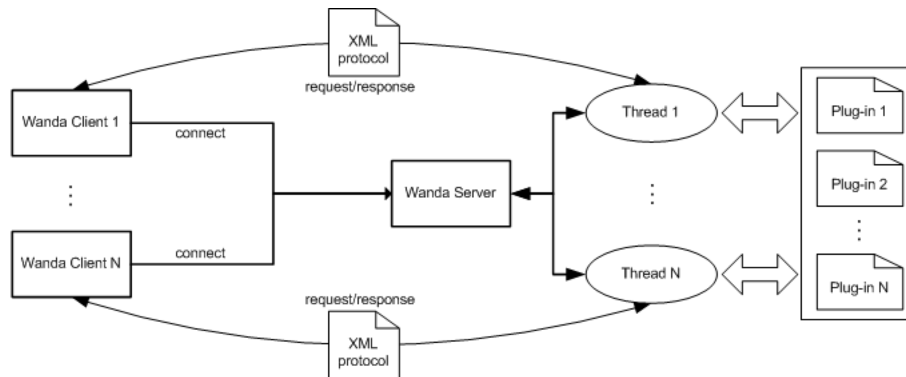


Figure 1.2: Structure of the Server

The advantages of this concept are:

1. The whole functionality of the WS is situated central on a server.
2. The extension of the functionality can be done by merely copying the new module into a given plug-in directory. This way it extends the functionality for all users simultaneously.
3. Improvements and debugging of failures of the plug-ins can be done centralized. This way, changes take effect to all WCs at once.
4. Platform independent, because XML documents are purely text-based.
5. Every executable program (even scripts, batch-files etc.) can be used as plug-in.
6. By using ASCII documents the usage of the HTTP protocol combined with one of the well-known web-servers can be possible.

Some disadvantages shall not be hided:

1. For every call of a plug-in the whole input-data must be transmitted to the WS and the output-data must be transferred back. This takes more time then using the plug-in modules directly within the program which needs the functionality.
2. A WC must either be written in Java or developed with a system/language which is able to process XML.
3. Because a plug-in is a normal executable program it has the same disadvantages of the CGI programs used within web-servers: Resources, e.g. connections to databases must be established every time the plug-in is called.

# Chapter 2

# Requirements

The main concept behind the Wanda-Server ('WS') is the ability to use every kind of executable program, script or whatever as plug-in. To realize this, one has to fulfill the following requirements while developing a WS plug-in:

1. The plug-in has to be an executable program (or script) which can be called within a shell.
2. The program is not allowed to wait for user interaction. No interaction will be given. Thus the program should be command line oriented.
3. The program gets an argument as filename for each input and output file. Such an argument consists of an absolute path of the appropriate file. But no extensions can be expected,i.e. the filename could consist only of a number like `/bin/wanda/temp/4324` even if this file is e.g. a `.jpg` or `.xml` file.
4. The sequence of arguments can be freely defined by the plug-in developer. Also arguments (e.g. own static parameters or variables) can be used, i.e. arguments not used for the input and output files. This sequence can be defined by using WPML.
5. The program must return an exit code after execution. In C/C++ this would be the return value of the main function or the value of the exit() function.
6. The program must return a 0 as exit code if the execution was successful.
7. For every case of an error, the program must return an exit code unequal to 0 defined by the plug-in developer. Additionally, the plug-in developer can specify his own error codes within the WPML document. This is not a must, but every not specified error results in an 'unspecified error'.
8. For the program, a WPML document must be written. This WPML document represents the plug-in and is analyzed by the Wanda-Server. By the way, one could write even multiple WPML documents for the same program, whereby each WPML document uses e.g. a different call and arguments for the program.

The main advantages of this concept are given in the following:

1. No restrictions are made to the used programming language. Even scripts could be used.
2. No APIs or libraries needed.

3. No one has to incorporate into a library or API or a new programming language.

4. Because the arguments can be defined freely within the WPML document, every command line oriented program able to get an input and output filename can be used as plug-in.

The following points could be understood as disadvantages of this concept:

1. Every plug-in has to be able to read the different image file formats for itself. No support is given by the WS. However, the libraries *SIC Image* and *SIC Sequi* are provided within the Wanda environment to support plug-in development.

2. Because the communication between the WS and the plug-in will be done by using files, this could lead to a worse performance than by using e.g. shared objects or DLLs as plug-ins. That means, the development and integration of plug-ins as libraries would enable a faster calling of these plug-ins.

# Chapter 3

# Procedure of a Plug-In Call

This chapter describes the procedure of the Wanda-Server, if a plug-in is called. If the Wanda-Client sends a request to call a plug-in, the Wanda-Server creates a context $C$ which contains among other things functions for getting unique filenames for all input ($U_{input}$) and output ($U_{output}$) files with the appropriate numbers $N_{input}$ and $N_{output}$:

$$U_{input} : \{0, \ldots, N_{input} - 1\} \rightarrow Z \quad (3.1)$$

$$U_{output} : \{0, \ldots, N_{output} - 1\} \rightarrow Z \quad (3.2)$$

$$C = (U_{input} = \{n \mapsto in_n \mid 0 \leq n < N_{input}\}, U_{output} = \{n \mapsto out_n \mid 0 \leq n < N_{output}\}, \ldots) \quad (3.3)$$

The context will be created by the Wanda-Server and gets unique filenames. For this, the Wanda-Server manages a counter $c_{names}$ which can be accessed only synchronized (because the Wanda-Server uses concurrent threads). This counter is initialized ($c_{names} = 0$) if the Wanda-Server starts to run. After each call to this counter, it will be incremented. For this the function $U_{names} :\rightarrow Z$ will be used: $U_{names}() = c_{names}, c_{names} = c_{names} + 1$.

The unique filenames are created as follows:

1. Create input filenames: $U_{input} = \emptyset$
   $\forall n \in \{0, \ldots, N_{input} - 1\} : U_{input} = U_{input} \cup \{n \mapsto U_{names}()\}$
2. Create output filenames: $U_{output} = \emptyset$
   $\forall n \in \{0, \ldots, N_{output} - 1\} : U_{output} = U_{output} \cup \{n \mapsto U_{names}()\}$

With the context $C$ (containing these unique filenames) the plug-in can be executed. This invocation of the plug-in is realized by the following steps:

1. Create invocation command **CMD** as string:
   String **DIR** = absolute path to the location of the input and output files.
   String **CMD** = content of WPML element *location*.
   For each element $E$ nested in WPML element *invocation* in top down order, do:

- $E$ = *parameter*: CMD = CMD + content of this WPML element *parameter*
- $E$ = *use_input*: CMD = CMD + DIR + $U_{input}$( use_input.number )
- $E$ = *use_output*: CMD = CMD + DIR + $U_{output}$( use_output.number )
- $E$ = *use_var*: CMD = CMD + content of this WPML element *use_var*

2. Execute plug-in by using the built invocation command. For this, use a system-call to invoke plug-in: `Runtime.getRuntime().exec( CMD )`.

3. Get exit code $e$ of plug-in. Every Wanda-Server plug-in has to return an exit code to indicate the state of success. If this exit code is unequal to 0, an error has occurred. In this case, the appropriate error message has to be determined and returned. For this, a list of *error* elements can be specified within WPML documents nested in an *errors* element as list: $EL = \{ERROR_1, \ldots, ERROR_n\}$, whereby each error has a code and a message: $ERROR(ERROR_{code} \in N, ERROR_{message})$. The error message is determined as follows:
   **if** $e \neq 0$:
   **if** $\exists ERROR \in EL.ERROR_{code} = e$
   **then** return $ERROR_{message}$
   **else** return "CallPlugin: unspecified error"
   If no error occurs the exit code 0 is returned:
   **if** $e = 0$
   **then** return "OK".

# Chapter 4

# Examples

This chapter reveals some examples of WPML documents describing plug-ins. Especially in section 4.2 WPML documents using all features of WPML are presented. A reference of all WPML elements can be found in chapter A.

## 4.1 Script-based Plug-in

Every executable program or script can be used as plug-in for the Wanda-Server. Imagine one has the following batch file called `version.bat` on a DOS operating system:

```
ver > %1
```

This batchfile writes the version number of the DOS system into a file given as first argument. If one wants to use this batch file as Wanda-Server plug-in to get the version number of the operating system the Wanda-Server is working on, one could use the following WPML document:

```
< WPML
  name    = "dos-ver"
  version = "0.1"
  author  = "Veenhuis, Christian"
  inputs  = "0"
  outputs = "1"
 >

 < description> This plugin returns the version of the DOS </ description>

 < call>
   < location> plugins/version.bat </ location>
   < invocation>
     < use_output number="0"/>
   </ invocation>
 </ call>
```

```
< protocol>

   < call_plugin />

   < to_client>
      < transfer_output number="0"/>
   </ to_client>

</ protocol>

</ WPML>
```

With the elements *WPML* and *description* some meta data are defined for this plug-in. Additionally this WPML document consists of two areas defined by the elements *call* and *protocol*.

The first area represents the call of the plug-in described with the following part:

```
< call>
   < location> plugins/version.bat </ location>
   < invocation>
      < use_output number="0"/>
   </ invocation>
</ call>
```

The *location* element contains the name of the script relative to the Wanda-Server (or as absolute path). The arguments used to call the plug-in are specified within the *invocation* element. The batch file only uses one output and no input files. This fact is declared by the attributes `inputs` and `outputs` of the *WPML* element. The only argument is the name of an output file. This name is represented by the *use_output* element.

The second area defines the communication sequence between the Wanda-Client and this plug-in. This is defined by the following part:

```
< protocol>

   < call_plugin />

   < to_client>
      < transfer_output number="0"/>
   </ to_client>

</ protocol>
```

Because this example uses no input files, nothing needs to be transferred from the Wanda-Client to the Wanda-Server. Thus, the plug-in can be called immediately. After this, the generated output file can be transferred from the Wanda-Server to the Wanda-Client. This is described by using the *to_client* and *transfer_output* elements.

If a Wanda-Client calls this plug-in it will get back a file containing the version number of the DOS system.

## 4.2 Binary-based Plug-in

The example shown in section 4.1 uses only one output file. The following example uses an executable program (.exe) on a DOS/Windows system. Of course, this example works on every platform, if the filename of the executable is adapted to the platform-specific form. Unlike the example given in section 4.1 this example uses additionally input files, parameters and variables. Thereby two slightly different versions of plug-ins are described, althoug both plug-ins use the same binary executable. This shows that a plug-in means a more abstract thing and is not a synonym for one executable file.

The two plug-ins are described in the following sections. For this we assume the existance of a program called `binarize.exe`. Calling this program in a shell produce the following output:

```
> binarize
binarize <input> <output> <threshold>

>
```

This program needs one input and one output file as well as a threshold for the binarizing operator. It returns the following exit codes:

| Exit Code | Meaning |
|---|---|
| 0 | Success (no error) |
| -1 | Wrong number of arguments |
| -2 | Unknown image format |
| -3 | Couldn't read image |
| -4 | Couldn't write image |
| -5 | Threshold invalid ([0;255]) |

### 4.2.1 Static Parameter

Assumed, one wants to use the program `binarize.exe` as described above to create a plug-in which binarizes the images with a static threshold of 127. This can be realized with the following WPML document:

```
< WPML
  name    = "bin"
  version = "0.1"
  author  = "Veenhuis, Christian"
  inputs  = "1"
  outputs = "1"
 >

 < description> Binarizes the given image with threshold 127 </ description>


 < call>
   < location> plugins/binarize/binarize.exe </ location>
   < invocation>
     < use_input number="0"/>
```

```
      < use_output number="0"/>
      < parameter> 127 </ parameter>
    </ invocation>
  </ call>


  < protocol>

    < to_server>
      < transfer_input number="0"/>
    </ to_server>

    < call_plugin />

    < to_client>
      < transfer_output number="0"/>
    </ to_client>

  </ protocol>


  < errors>
    < error code="-1" msg="Wrong number of arguments"/>
    < error code="-2" msg="Unknown image format"/>
    < error code="-3" msg="Couldn't read image"/>
    < error code="-4" msg="Couldn't write image"/>
    < error code="-5" msg="Threshold invalid ([0;255])"/>
  </ errors>

</ WPML>
```

With the elements *WPML* and *description* some meta data are defined for this plug-in. Additionally this WPML document consists of three areas defined by the elements *call*, *protocol* and *errors*.

The first area represents the call of the plug-in described with the following part:

```
  < call>
    < location> plugins/binarize/binarize.exe </ location>
    < invocation>
      < use_input number="0"/>
      < use_output number="0"/>
      < parameter> 127 </ parameter>
    </ invocation>
  </ call>
```

The *location* element contains the name of the program relative to the Wanda-Server (or as absolute path). The arguments used to call the plug-in are specified within the *invocation* element. This example uses one input and one output file (as declared in the *WPML* element). The invocation is build in top-down order of the elements contained within the *invocation* element. That means, the first argument is the name for the input file, the second is the name of the output file produced by the program. The third argument is a static

parameter which is used for each call of this plug-in. Such parameters can be declared by using *parameter* elements.

The second area defines the communication sequence between the Wanda-Client and this plug-in. This is defined by the following part:

```
< protocol>

  < to_server>
    < transfer_input number="0"/>
  </ to_server>

  < call_plugin />

  < to_client>
    < transfer_output number="0"/>
  </ to_client>

</ protocol>
```

In this example one input file is used. This input file should be transferred from the Wanda-Client to the Wanda-Server as first action to be existance while calling the program. For this the *to_server* and *transfer_input* elements are used. After this, the program for this plug-in is called. The generated output file can be transferred from the Wanda-Server to the Wanda-Client. This is described by using the *to_client* and *transfer_output* elements.

In some cases the program for this plug-in could return an error. This can happen, if, e.g., the input file from the Wanda-Client doesn't fulfill the expectations of the program. For this, all possible errors returned by the program can be declared within the WPML document:

```
< errors>
  < error code="-1" msg="Wrong number of arguments"/>
  < error code="-2" msg="Unknown image format"/>
  < error code="-3" msg="Couldn't read image"/>
  < error code="-4" msg="Couldn't write image"/>
  < error code="-5" msg="Threshold invalid ([0;255])"/>
</ errors>
```

If one of these errors occur, the Wanda-Server returns the appropriate error-code and message back to the Wanda-Client.

### 4.2.2 Variable Parameter

Assumed, one wants to use the program `binarize.exe` as described above to create a plug-in which binarizes the images with any threshold. The threshold shall be given over from the Wanda-Client. This can be realized with the following WPML document:

```
< WPML
  name    = "bint"
  version = "0.1"
```

```
 author  = "Veenhuis, Christian"
 inputs  = "1"
 outputs = "1"
>

< description> Binarizes the given image with the given threshold </ description>


< vars>
  < var name="threshold" />
</ vars>


< call>
  < location> plugins/binarize/binarize.exe </ location>
  < invocation>
    < use_input number="0"/>
    < use_output number="0"/>
    < use_var name="threshold" />
  </ invocation>
</ call>


< protocol>

  < to_server>
    < transfer_input number="0"/>
    < transfer_var  name="threshold" />
  </ to_server>

  < call_plugin  />

  < to_client>
    < transfer_output number="0"/>
  </ to_client>

</ protocol>


< errors>
  < error code="-1" msg="Wrong number of arguments"/>
  < error code="-2" msg="Unknown image format"/>
  < error code="-3" msg="Couldn't read image"/>
  < error code="-4" msg="Couldn't write image"/>
  < error code="-5" msg="Threshold invalid ([0;255])"/>
</ errors>

</ WPML>
```

This example resembles the above example in section 4.2.1. Therefore, only the differences are explained in the following.

To use variable arguments (i.e. arguments transferred from the Wanda-Client) one has to declare them like the following:

```
< vars>
  < var name="threshold" />
</ vars>
```

The *vars* element can hold any number of *var* elements. Every variable which shall be used is to declare by using a *var* element. This way, the Wanda-Server can handle the usage of variables within the other WPML elements. For this example a variable called `threshold` is declared.

Instead of using the *parameter* element, the *use_var* element is used to create the third argument for the program:

```
< call>
  < location> plugins/binarize/binarize.exe </ location>
  < invocation>
    < use_input number="0"/>
    < use_output number="0"/>
    < use_var name="threshold" />
  </ invocation>
</ call>
```

That means that the content of this variable is used as argument. This content must be transferred from the Wanda-Client to the Wanda-Server:

```
< protocol>
  < to_server>
    < transfer_input number="0"/>
    < transfer_var  name="threshold" />
  </ to_server>
        ⋮
</ protocol>
```

In this example first the input file and afterwards the variable are transferred from the Wanda-Client to the Wanda-Server. But this order can be defined freely by the plug-in developer.

# Appendix A

# WPML Reference of all Elements

This chapter reveals all elements with their definitions in form of a language reference. The single elements are presented with the following notation:

< **NameOfElement**

`NameOfAttribute = "` **Default**`"`

>

`[frequ] <ContentElement>` **Default content** `</ContentElement>`

</ **NameOfElement**>

**NameOfElement**
The name of the element according to its task e.g. *var*, *invocation* or *error*.

**NameOfAttribute**
An element can possess arbitrarily many attributes. Beside its name (`NameOfAttribute`) a single attribute possesses a default value (`Default`). The default value is taken, whenever this attribute is not defined explicitly.

**ContentElement**
Some elements can have as many as desired elements as subordinated contents. Everyone of these subordinated elements is an independent element thus possessing its own `NameOfElement`. Similar to the attributes it possesses also a content as default (`Default content`). The default content is used, if no other content was explicitly defined. The specification `[frequ]` indicates how much of the appropriate elements are allowed to occur. A number for `frequ` means exactly this number of occurences. A `+` means at least one, a `?` zero or one, and an `*` any number of occurences (including 0). Sometimes the content is optionally presented by a simple explaination.

To clarify the notation a sample element with the name `Person` (not existent in WPML) is described:

< **Person**

`name REQUIRED`

```
  age = " 19"
  weight = " 70.0"

>

  [*] <Hobby> </Hobby>
  [+] <Residence> Berlin </Residence>
  [1] <ID> </ID>

</ Person>
```

In the description of the element *Person* all valid description elements were used for the definition of the WPML elements. There are the name of element (*Person*), some attributes and two content elements. The attributes possess default values (up to the `name` attribute). To assign a name a default value is not meaningful; instead one marks such attributes with the note: REQUIRED. Thus this attribute is marked as mandatory necessary. The content elements are the elements for describing the hobbies if in existance (with no default value), a single ID number and at least one residence (with `Berlin` as default). A valid *Person* element in accordance with the above description would be:

```
< Person name = " Meier" weight = " 82.4" >

  <Hobby> Parachuting </Hobby>
  <Hobby> Sailing </Hobby>

  <Residence> Munich </Residence>

  <ID> IDP31415926 </ID>

</ Person>
```

In the following section all elements are presented using the above notation.

## A.1 Root Element and Meta Data

### A.1.1 WPML

The root element is situated on the highest level and encloses all other WPML elements. This element represents the whole plug-in.

```
< WPML
  name    REQUIRED
  author  = ""
  version = " 0"
  inputs  = " 1"
  outputs = " 1"
>

 [*] <description> </description>
```

```
[*] <vars> </vars>
[*] <call> </call>
[*] <protocol> </protocol>
[*] <errors> </errors>
```

**</ WPML>**

**Attribute: name**
The name of the plug-in represented by this *WPML* element.

**Attribute: author**
The author(s) of this WPML document.

**Attribute: version**
The version number of this WPML document (not the version of WPML).

**Attribute: inputs**
The number of input files transferred from Wanda Client to Wanda Server. The numbers in interval $[0; inputs - 1]$ are used to identify the input files while invoking the plug-in or within the communication protocol.

**Attribute: outputs**
The number of output files transferred from Wanda Server to Wanda Client. The numbers in interval $[0; outputs - 1]$ are used to identify the output files while invoking the plug-in or within the communication protocol.

**Content: description**
This element can contain any number of *description* elements as described in section A.1.2. But only the last defined *description* element is used. All previous ones are ignored (or overwritten by the last one).

**Content: vars**
This element can contain any number of *vars* elements as described in section A.2.1. The content of all *vars* elements are collected.

**Content: call**
This element can contain any number of *call* elements as described in section A.3.1. But only the last defined *call* element is used. All previous ones are ignored (or overwritten by the last one).

**Content: protocol**
This element can contain any number of *protocol* elements as described in section A.4.1. The content of all *protocol* elements are collected and executed in their order (top-down).

**Content: errors**
This element can contain any number of *errors* elements as described in section A.5.1. The content of all *errors* elements are collected.

### A.1.2 description

This element contains a text simply describing this plug-in.

**< description>**

```
... any text ...
```

**</ description>**

**Content**
Any text can be used to describe the plug-in.

## A.2 Variables

### A.2.1 vars

Plug-ins can be invoked by using variables as arguments. Every variable transferred and used has to be declared previously within a *vars* element.

**< vars>**

```
[+] <var />
```

**</ vars>**

**Content: var**
This element contains at least one *var* element as described in section A.2.2.

### A.2.2 var

A single variable can be defined by using this element. Only the name of the variable is neccessary so that the WPML interpreter is able to create and use this variable.

**< var**
```
  name    REQUIRED
```
>

**</ var>**

**Attribute: name**
The name of the variable. This name must be unique. Otherwise the WPML interpreter rejects the WPML document.

## A.3 Invocation of Plug-in

### A.3.1 call

This element contains all information needed to create the call of the plug-in program. The contents of all *invocation* children are collected from all *call* elements.

**< call>**

```
[*] <location> </location>
[*] <invocation> </invocation>
```

**</ call>**

**Content: location**
This element can contain any number of *location* elements as described in section A.3.2. But only the last defined *location* element is used. All previous ones are ignored (or overwritten by the last one).

**Content: invocation**
This element can contain any number of *invocation* elements as described in section A.3.3. The content of all *invocation* elements are collected and executed in their order (top-down).

### A.3.2 location

This element contains the path for invocing the plug-in program.

**< location>**

```
... the path ...
```

**</ location>**

**Content**
Any valid text can be used as path for invoking the plug-in program. The separators '/' and '\' are automatically adapted to the appropriate operating system. This path is supposed to be relative to the Wanda-Server or absolute.

### A.3.3 invocation

This elements collects the arguments for calling the plug-in.

**< invocation>**

```
[*] <parameter> </parameter>
[*] <use_var />
[*] <use_input />
[*] <use_output />
```

**</ invocation>**

**Content: parameter**
This element can contain any number of *parameter* elements as described in section A.3.4. All *parameter* elements are collected and executed in their order (top-down).

**Content: use_var**
This element can contain any number of *use_var* elements as described in section A.3.5. All *use_var* elements are collected and executed in their order (top-down).

**Content: use_input**
This element can contain any number of *use_input* elements as described in section A.3.6. All *use_input* elements are collected and executed in their order (top-down).

**Content: use_output**
This element can contain any number of *use_output* elements as described in section A.3.7. All *use_output* elements are collected and executed in their order (top-down).

### A.3.4 parameter

This element enables the definition of static arguments. A static argument is an argument which is the same for every call of the plug-in.

**< parameter>**

```
... the argument ...
```

**</ parameter>**

**Content**
Any text can be used as static argument for invoking the plug-in program.

### A.3.5 use_var

To use the content of a variable as argument, this element can be used. For this, the variable must be defined within a *vars* element. The content of the variable is transferred between client and server according to the communication protocol defined within the *protocol* element.

**< use_var**
```
name    REQUIRED
```
>

**</ use_var>**

**Attribute: name**
The name of the variable. There must be a variable defined with this name within a *vars* element. Otherwise the WPML interpreter rejects the WPML document.

### A.3.6 use_input

To use the unique name generated for the input file of a given number as argument, this element can be used. For this, the given number must be in $[0; WPML.inputs - 1]$, i.e. it must be in the interval defined by the *WPML* element.

**< use_input**
```
number    REQUIRED
```
>

**</ use_input>**

**Attribute: number**
The number of input file. This number must be in $[0; WPML.inputs - 1]$. Otherwise the WPML interpreter rejects the WPML document.

### A.3.7 use_output

To use the unique name generated for the output file of a given number as argument, this element can be used. For this, the given number must be in $[0;WPML.outputs-1]$, i.e. it must be in the interval defined by the *WPML* element.

**< use_output**
```
  number    REQUIRED
>
```

**</ use_output>**

**Attribute: number**
The number of output file. This number must be in $[0;WPML.outputs-1]$. Otherwise the WPML interpreter rejects the WPML document.

## A.4 Communication

### A.4.1 protocol

This element contains all information needed to realize the bi-directional communication between the Wanda-Client and Wanda-Server. Thereby all contained elements are executed in their order, i.e. you can combine any number of those elements to realize a mixed communication (transfer things to the server, then to client, maybe to the server again, call the plugin, etc.).

**< protocol>**

```
 [*] <to_server> </to_server>
 [*] <to_client> </to_client>
 [*] <call_plugin />
```

**</ protocol>**

**Content: to_server**
This element can contain any number of *to_server* elements as described in section A.4.2. Multiple *to_server* elements are executed in their order (top-down).

**Content: to_client**
This element can contain any number of *to_client* elements as described in section A.4.3. Multiple *to_client* elements are executed in their order (top-down).

**Content: call_plugin**
This element can contain any number of *call_plugin* elements as described in section A.4.7. Multiple *call_plugin* elements are executed in their order (top-down).

### A.4.2 to_server

This element collects a set of transfer commands. These transfers are done from the Wanda-Client to the Wanda-Server in the defined order (top-down).

**< to_server>**

```
[*] <transfer_var />
[*] <transfer_input />
[*] <transfer_output />
```

**</ to_server>**

**Content: transfer_var**
This element can contain any number of *transfer_var* elements as described in section A.4.4. All *transfer_var* elements are collected and executed in their order (top-down).

**Content: transfer_input**
This element can contain any number of *transfer_input* elements as described in section A.4.5. All *transfer_input* elements are collected and executed in their order (top-down).

**Content: transfer_output**
This element can contain any number of *transfer_output* elements as described in section A.4.6. All *transfer_output* elements are collected and executed in their order (top-down).

### A.4.3 to_client

This element collects a set of transfer commands. These transfers are done from the Wanda-Server to the Wanda-Client in the defined order (top-down).

**< to_client>**

```
[*] <transfer_var />
[*] <transfer_input />
[*] <transfer_output />
```

**</ to_client>**

**Content: transfer_var**
This element can contain any number of *transfer_var* elements as described in section A.4.4. All *transfer_var* elements are collected and executed in their order (top-down).

**Content: transfer_input**
This element can contain any number of *transfer_input* elements as described in section A.4.5. All *transfer_input* elements are collected and executed in their order (top-down).

**Content: transfer_output**
This element can contain any number of *transfer_output* elements as described in section A.4.6. All *transfer_output* elements are collected and executed in their order (top-down).

### A.4.4 transfer_var

To transfer the content of a variable, this element can be used. For this, the variable must be defined within a *vars* element.

**< transfer_var**
```
  name    REQUIRED
>
```

**</ transfer_var>**

**Attribute: name**
The name of the variable. There must be a variable defined with this name within a *vars* element. Otherwise the WPML interpreter rejects the WPML document.

### A.4.5 transfer_input

To transfer an input file of a given number, this element can be used. For this, the given number must be in $[0; WPML.inputs - 1]$, i.e. it must be in the interval defined by the *WPML* element.

**< transfer_input**
```
  number    REQUIRED
>
```

**</ transfer_input>**

**Attribute: number**
The number of input file. This number must be in $[0; WPML.inputs - 1]$. Otherwise the WPML interpreter rejects the WPML document.

### A.4.6 transfer_output

To transfer an output file of a given number, this element can be used. For this, the given number must be in $[0; WPML.outputs - 1]$, i.e. it must be in the interval defined by the *WPML* element.

**< transfer_output**
```
  number    REQUIRED
>
```

**</ transfer_output>**

**Attribute: number**
The number of output file. This number must be in $[0; WPML.outputs - 1]$. Otherwise the WPML interpreter rejects the WPML document.

### A.4.7 call_plugin

Whenever this element occurs within the *protocol* element, the plug-in program is executed according to the *call* element. This element assumes that all needed input files and variables etc. are transferred by previously defined *to_server* and *to_client* elements. The *call_plugin* element can be called as often as needed. Thus, a calling sequence in combination with the other elements of the *protocol* element can be realized.

**< call_plugin>**

**</ call_plugin>**

## A.5 Error Handling

### A.5.1 errors

Plug-in programs are able to return an exit-code as error code. E.g. if a called plug-in cannot handle the given input files or arguments are wrong, an error code is returned. To inform the Wanda-Client about the success of a call, an error code is returned by the Wanda-Client. If the plug-in returns an error code unequal to 0, the appropriate error is searched within the list of *error* elements. The found error is returned to the Wanda-Client. Could no appropriate *error* element be found, an error with the message 'unspecified error' is returned.

**< errors>**

```
 [*] <error />
```

**</ errors>**

**Content: error**
This element can contain any number of *error* elements as described in section A.5.2. All *error* elements are collected.

### A.5.2 error

A single error can be defined by using this element. For this, you can use the exit code of your plug-in program as error code. Additionally, a message in form of a string (of one line text) can be specified. Both, the error code and the message are returned to the Wanda-Client, if this error occurs.

**< error**

```
  code    REQUIRED
  msg     REQUIRED
```

>

**</ error>**

**Attribute: code**
The error code as signed integer. At this place, the exit code of the plug-in program can be specified. It should be different to 0, because an exit code of 0 is interpreted as 'no error'. If you set this attribute to 0, this *error* elementt will be ignored.

**Attribute: msg**
The message of this error as string. Please use a senseful description so that a Wanda-Client can understand the specific problem of your plug-in program.

# Appendix B

# WPML Document Type Definition (DTD)

For the development of valid WPML descriptions with any (validating) XML editor, a document type definition (DTD) was created for WPML. It describes the current state of WPML and is presented in the following:

```
<!--

  WPML (Wanda Plugin Modeling Language)

  Version 0.1

  07.10.2002

  Christian Veenhuis

-->



<!-- ================= -->
<!--        WPML       -->
<!-- ================= -->

<!ELEMENT  WPML (description | vars | call | protocol | errors )*  >

<!ATTLIST  WPML name    CDATA    #REQUIRED >
<!ATTLIST  WPML author  CDATA    "" >
<!ATTLIST  WPML version CDATA    "0" >
<!ATTLIST  WPML inputs  CDATA    "1" >
<!ATTLIST  WPML outputs CDATA    "1" >
```

```
<!-- ============================================================================ -->
<!-- META DATA                                                                    -->
<!-- ============================================================================ -->


<!-- ================= -->
<!--   description     -->
<!-- ================= -->

<!ELEMENT  description (#PCDATA) >


<!-- ============================================================================ -->
<!-- VARIABLES                                                                    -->
<!-- ============================================================================ -->


<!-- ================= -->
<!--       vars        -->
<!-- ================= -->

<!ELEMENT  vars  (var)+ >


<!-- ================= -->
<!--       var         -->
<!-- ================= -->

<!ELEMENT  var  EMPTY >

<!ATTLIST  var name  CDATA   #REQUIRED >


<!-- ============================================================================ -->
<!-- CALL                                                                         -->
<!-- ============================================================================ -->


<!-- ================= -->
<!--      call         -->
<!-- ================= -->

<!ELEMENT  call (location|invocation)* >


<!-- ================= -->
<!--     location      -->
<!-- ================= -->
```

```
<!ELEMENT  location (#PCDATA) >


<!-- ================= -->
<!--     invocation    -->
<!-- ================= -->

<!ELEMENT  invocation (parameter|use_var|use_input|use_output)* >


<!ELEMENT  parameter (#PCDATA) >

<!ELEMENT  use_var  EMPTY >
<!ATTLIST  use_var name  CDATA   #REQUIRED >

<!ELEMENT  use_input EMPTY >
<!ATTLIST  use_input number  CDATA   #REQUIRED >

<!ELEMENT  use_output EMPTY >
<!ATTLIST  use_output number  CDATA   #REQUIRED >


<!-- =========================================================================== -->
<!-- PROTOCOL                                                                    -->
<!-- =========================================================================== -->


<!-- ================= -->
<!--      protocol     -->
<!-- ================= -->

<!ELEMENT  protocol (to_server|to_client|call_plugin)* >

<!ELEMENT  to_server (transfer_var|transfer_input|transfer_output)* >

<!ELEMENT  transfer_var  EMPTY >
<!ATTLIST  transfer_var name  CDATA   #REQUIRED >

<!ELEMENT  transfer_input EMPTY >
<!ATTLIST  transfer_input number  CDATA   #REQUIRED >

<!ELEMENT  transfer_output EMPTY >
<!ATTLIST  transfer_output number  CDATA   #REQUIRED >


<!ELEMENT  to_client (transfer_var|transfer_input|transfer_output)* >
<!-- <transfer_var> already defined -->
<!-- <transfer_input> already defined -->
<!-- <transfer_output> already defined -->
```

```
<!ELEMENT  call_plugin EMPTY >


<!-- ============================================================================ -->
<!-- ERRORS                                                                       -->
<!-- ============================================================================ -->


<!-- ================= -->
<!--       errors      -->
<!-- ================= -->

<!ELEMENT  errors  (error)* >


<!-- ================= -->
<!--        error      -->
<!-- ================= -->

<!ELEMENT  error  EMPTY >

<!ATTLIST  error code  CDATA   #REQUIRED >
<!ATTLIST  error msg   CDATA   #REQUIRED >
```