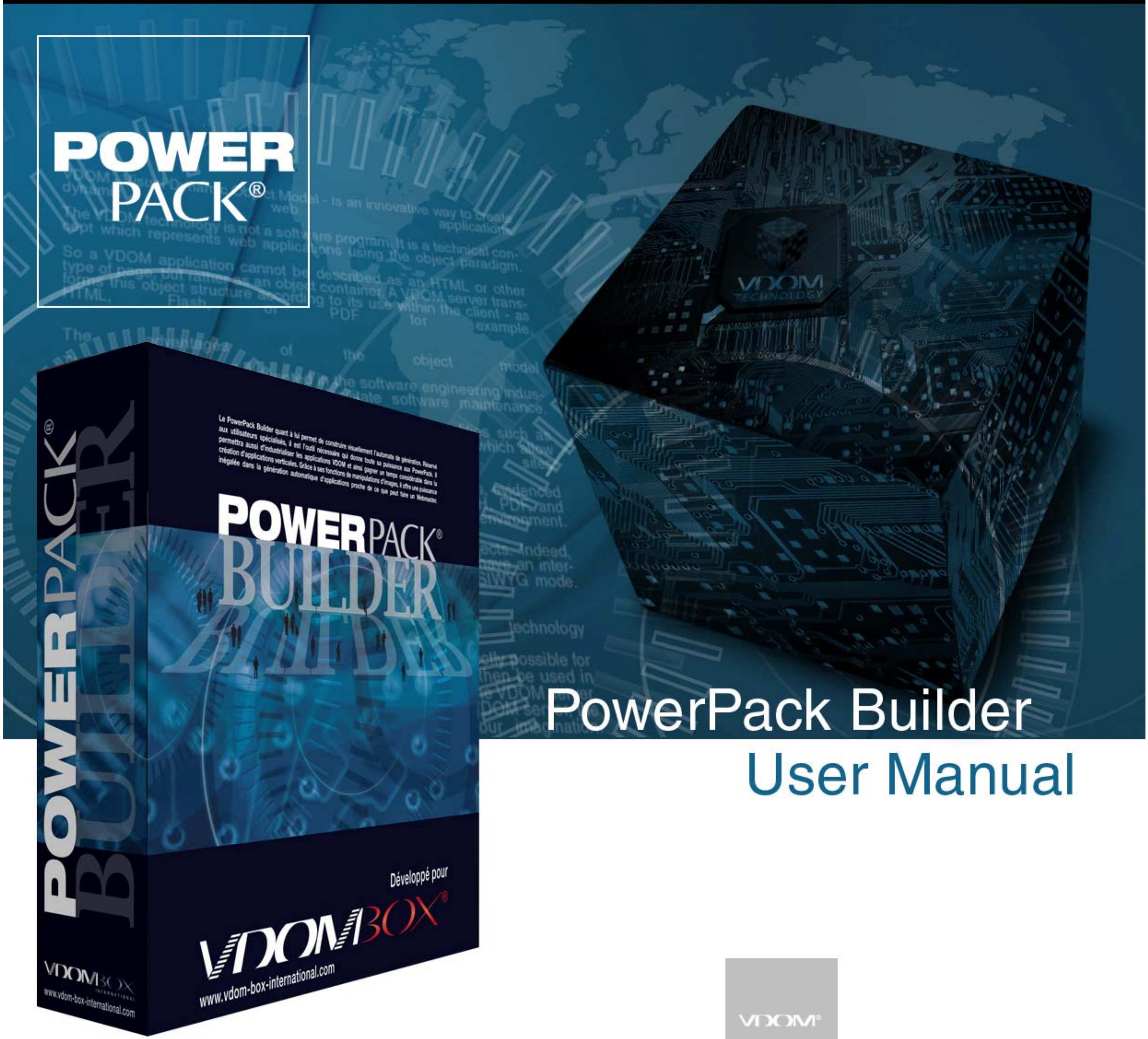
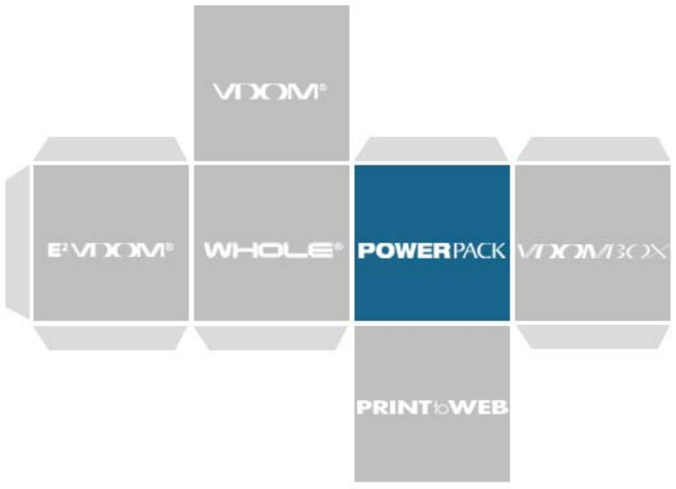
VDOMBOX INTERNATIONAL


POWER PACK®
Le PowerPack Builder quant à lui permet de construire visuellement l'automate de génération. Réservé aux utilisateurs spécialisés, il est l'outil nécessaire qui donne toute sa puissance aux PowerPack. Il permettra aussi d'industrialiser les applications VDOM et ainsi gagner un temps considérable dans la création d'applications verticales. Grâce à ses fonctions de manipulations d'images, il offre une puissance inégalée dans la génération automatique d'applications proche de ce que peut faire un Webmaster.
POWERPACK® BUILDER
Développé pour
VDOMBOX®
www.vdom-box-international.com
VDOM TECHNOLOGY

PowerPack Builder
User Manual



Copyright © 2008 VDOM Box International OOD, All rights reserved.

The information published in this document represents the opinion of VDOM Box International as for the subjects tackled at the date of publication. VDOM Box International having to answer fluctuating conditions of market, the information published in this document should not be interpreted like an engagement of its share, VDOM Box International which cannot guarantee the precision of this information after the date of publication.

This guide of evaluation is published only at ends of information. VDOM Box International does not offer any guarantee, express or implicit, as for the contents of this document.

The subject tackled in this document can make the object of patents, patent applications, marks, copyright or other rights of ownership intellectual belonging to VDOM International Box and/or Nicolas Korboulewsky. This document does not give you any right to these patents, marks, copyright or another intellectual property, except in the case of a specific contract drawn up with VDOM International Box and/or Nicolas Korboulewsky.

V.D.O.M., logo V.D.O.M., software suite, V.D.O.M., W.H.O.L.E. logo W.H.O.L.E. E²V.D.O.M., Logo E²V.D.O.M. are marks of trade or trademarks under license by VDOM International Box in France and/or in other countries. The other names or marks belong to their respective owners.

VDOM Box International, Sofia 1220, Nadezkda bl 122 Ent. A

Table of Content

Introduction

VDOM Server use XML file to describe and store a VDOM application, the Power Pack are designed to generate this file, this section will introduce why and how we do it.

1. PowerPack Objectives

1.1 VDOM Application & VDOM Box

The VDOM Box Appliance is designed to run VDOM Applications, a VDOM application is made of a XML file. An XML file is a text oriented file with an internal structure based on Tags. This file can be easily generate by any software witch can write ASCII or any other characters encoding system.

To create a VDOM Application we can use the VDOM IDE connected to a VDOM Box, in this case it's the server software that create and update the VDOM Application XML File. This way is the most common way to create a VDOM Application.

But it appears that Web Application can also be categorized in several generics applications, to save developing time it could be useful to pre-generate a basic (or even complex) template structure of this application, we can even imagine to completely generate a full functional application.

To generate a VDOM Application we need in fact to create the XML file of this application to be able next to install it on the VDOM Box. We can also imagine to directly install it from the Power Pack reader by a direct connection to the VDOM Box.

This documentation will describe the use of the Power Pack Builder and the concepts linked to it.

1.2 Graph oriented XML Generator

The Power Pack software is in fact a XML generator as you have understand, but what we would like is to be able to generate not only a strict copy of a template but to be able to generate some variations of it. To do it we can use a non determinist oriented graph. The idea is to follow a graph where each node contain a part of the XML text to generate, each node are connected to an other one, but many link can start from a node to others one. If this situation happen the choice to follow a special link is done randomly.

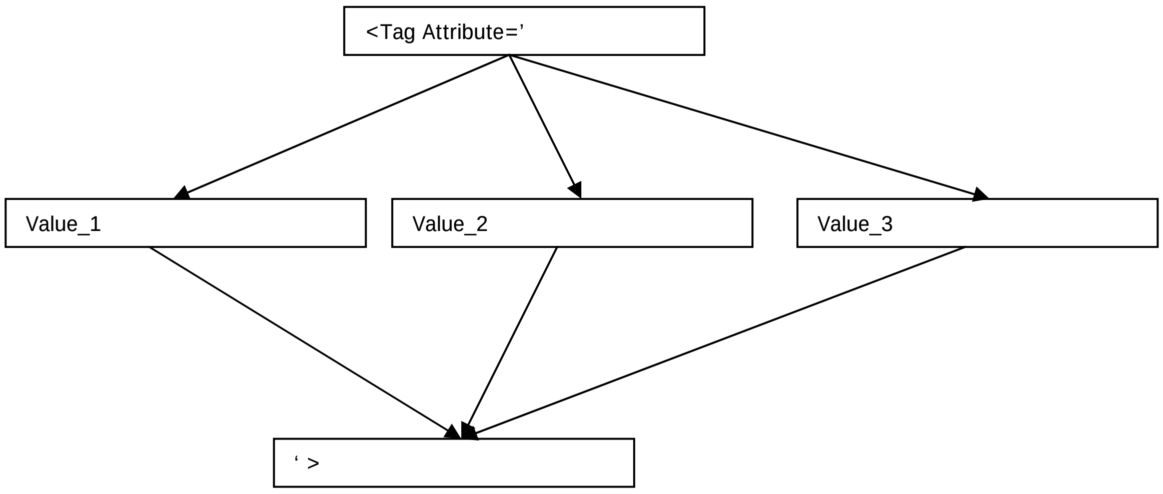


Figure 1: Graph example.

So depending of the link followed the result can be <Tag Attribute='Value_1'>; <Tag Attribute='Value_2'>; <Tag Attribute='Value_3'>, this demonstrate how to introduce variation into a template. The Power Pack Builder will be software than enable the user to create this graph and test it, those graphs will be store as file. The Power Pack Reader will use this file to generate VDOM Applications; it will create XML File or directly install it on a VDOM Box.

Power Pack Standard Design

This section will first introduce you the graph oriented design and then describe the VDOM XML file specification to allow you to generate it with the Power Pack Builder tool.

1. Graph oriented modeling, functional description

To generate the XML file we are using graph oriented model, it consist in a set of nodes and arrows (called transition) connecting the nodes, each arrow indicate a possible transition to the next node.

One traditionally distinguishes two types of nodes: initial nodes and final nodes. Each graph must have an initial node, which indicates a valid entrance point, and one or more final nodes which indicate the states on which the course can be completed. When no transition starts from a state, this last is implicitly regarded as terminal by the program.

In addition of the two types evoked above, the nodes are divided into three categories:

- **Normal:** The node of graph represents a character string. Certain sequences of letters can have a particular **significance**.
- **Subgraph:** This state causes a jump in another graph. The character string of the state represents the exact name of the graph. When the generator meets such a state, it jumps towards the specified graph, the crosses completely, and then begins again starting from this state.
- **Commands:** A command makes it possible to assign a value to a variable, do computations, make various tests to condition the process of generation, and execute embedded functions. The commands are the most complex part to apprehend, but remain necessary to create the most advanced generators.

1.1 Add node to the graph

You will click on the bottom of the graph, then on the right button of the mouse to reveal a contextual menu, and will select "Add a state". A node containing the default value "Node" is then added on the place where you clicked.

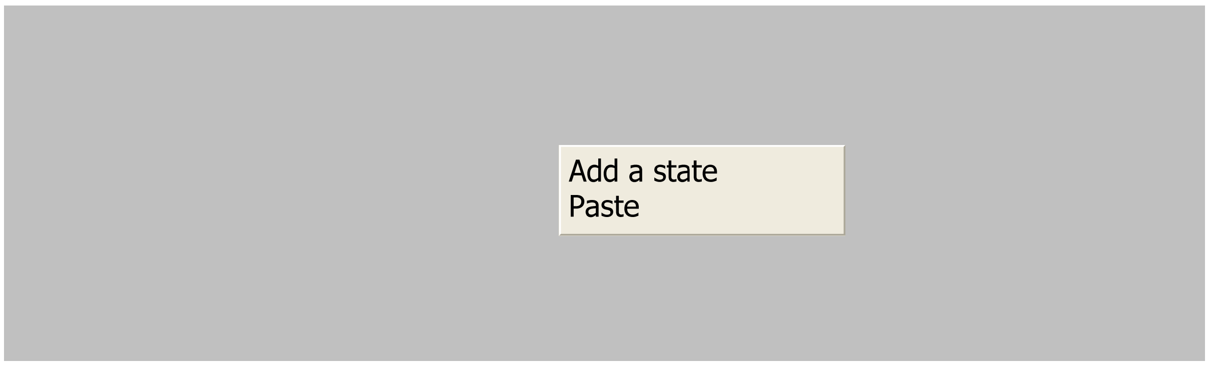


Figure 2: Add a state to a graph

1.2 Modify a node value

You will double click on the node. The node is then replaced by a zone of editable text. You will be able to then modify its contents, then it will validate while pressing on the "Enter" key, or while clicking elsewhere on the graph. It is possible to cancel the modification in progress while pressing on the key of exhaust ("ESC" or "ESCAPE").

1.3 Modify a node type

The user will position the pointer of the mouse on a node, then click-right. A contextual menu will appear then. The user will choose the suitable type then.

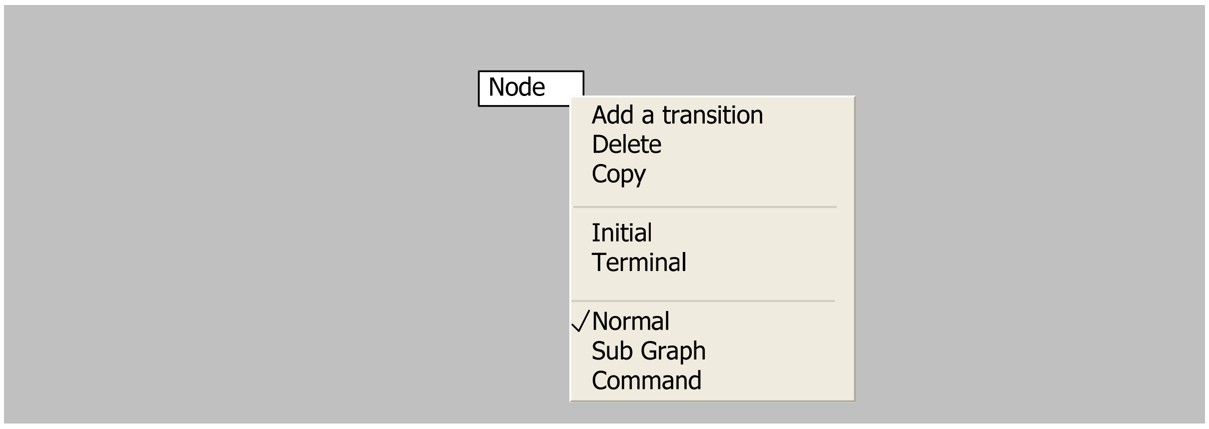


Figure 3: Change the node Type : last part of the menu

1.4 Add a transition

As previously, the user will click right on a node, then it will choose the option "Add a transition". An arrow appears then, and follows the pointer of mouse. The user will choose a target node then, and will click above to add a transition which goes from the first node to this one. To cancel, he will click on the bottom of the graph, at a place without node.

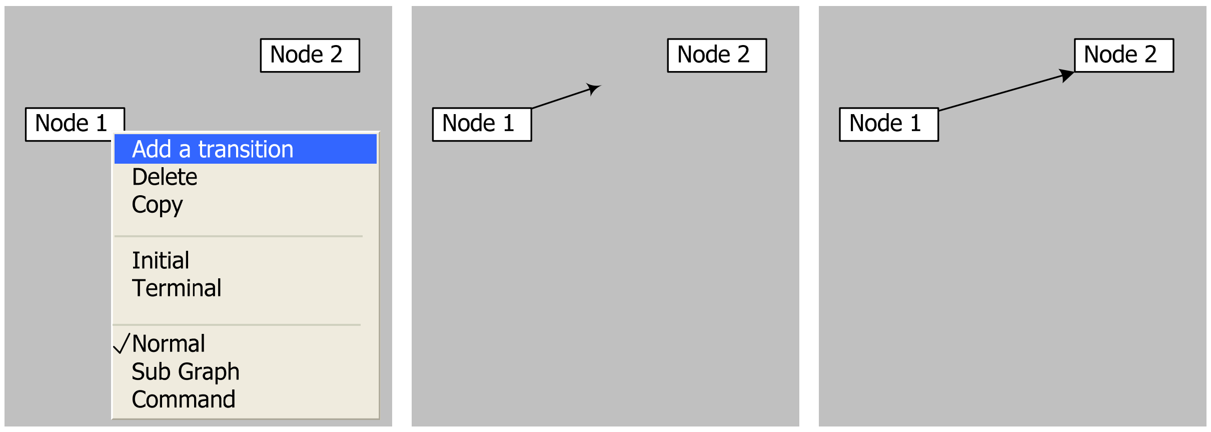


Figure 4: Process to create a transition

1.5 Manage graphs

The user will be able to add and erase graphs while respectively clicking on the buttons "Add a graph" and "Erase" located at the top of the list of the graphs available. The user will click on the elements of this list to select and publish a particular graph.

Description of a node known as "normal"

A normal node of graph (represented graphically in black letters on white zone) contains a simple character string. This character string is taken again just as it is during the process generation time, and a space is added automatically between each state during the generation.

Some special letters sequences (begin with ' \ ' or ' $ ') have a particular significance, and will be interpreted during the generation:

Particular sequences:

- **\n**: This sequence will be replaced by a line jump.
- **\t** : This one by tabulation.
- **\r** : This one by carry return.
- **\-** : When this sequence appears, it will be erased and any space located before or after will also be erased. That in particular makes it possible to stick two states together, when for example one ends in an apostrophe, or starts with a comma or a point.

VARIABLES REPLACEMENT:

It’s also possible to insert the value of a variable by specifying it like this: $Variable. For example in the chain "The $Objet", "$Objet" will be replaced by its value. The replacement is carried out generation progressively, and the variable is replaced by its value at the time when the state is traversed.

1.6 Commands

The command makes it possible to execute complex operations. They are divided into two types:

- **Test commands:** they make it possible to prevent the generator from passing by a transition if the test is not checked.
- **Other operations:** any command which is not a test is simply executed when the generator passes by the state which contains it. These operations can assign a value to a variable, execute a calculation, or jump to another graph.

DATA TYPE:

The commands handle data which can be literal variables or values.

The literal values can be a list of letters or numbers, or an unspecified sequence of characters put between quotation marks.

The variables are always prefixed by the sign ` $ ', and are made up of letters, numbers, or the sign ` _ ' (underlined). For example $toto, $var, $i is valid variables.

The variables are not typified, and can contain as well character strings as whole numerical values. The interpreter does automatically the choice which is appropriate according to the values. For example 3 + $toto will correspond to an addition if $toto contains a numerical value, and with a concatenation of the chain "3" and contents of the variable $toto if the latter contains a chain.

List manipulation allow the system to create dynamically function to evaluate, a List is define like this [elt1 elt2 … eltn]. The element of the list can be string or other lists.

The elements of a list are separated by a space char 4 types of element can be manipulated:

- **Word:** a word is a sequence of ascii char with no space. Like [MyWord1 MyWord2 ..]
- **String:** a string is an element which start with ' and finish with the same ', inside this sequence with use this escape sequence \' and \\.

- **List:** List can contain sub list, a sub list begin with [and finish with], of course inside this list all the type can be used.
- **Variable:** A variable start with $ with only ASCII char the end of this variable will be the next space. The function Evaluate will replace in list the list evaluated all variable by their value, but the other functions (GET,PUT,UPDATE,…) will return the string representing the variable

A special function [Evaluate $List] will allow the command node to evaluate the list given in parameter.

1.7 The tests

A certain number of tests are available, and are described in the table below.

Operator	Description	Example

==	Test the equality between two operands.	`$toto == "truc"`
		`$tutu == 3`

!=	Test the inequality between two operands.	`$toto != "machine"`
		`$tutu != 2443`

<	Test the relation of inferiority between two operands.	`$toto < 3`
		`3 < $tutu`

<=	Test the relation "inferior or equal" between two operands.	`$toto <= 3`
		`3 <= $tutu`

<	Test the relation of superiority between two operands.	`$toto > 3`
		`3 > $tutu`

>=	Test the relation "superior or equal" between two operands.	`$toto >= 3`
		`3 >= $tutu`

||	OR: the relation is true so at least one of the two operands is true.	`($toto >= 3) || (3 >= $tutu)`
&&	AND: the relation is true if the two operands are true.	`($toto >= 3) && (3 >= $tutu)`

The test result is always 0 (False) or 1 (True).

ASSIGNMENT:

Assign a value to a variable name. The assignment is done with the sign = in the following way: $VARIABLE = EXPRESSION with VARIABLE the name of the variable, and EXPRESSION an unspecified expression, a variable, character strings, or a whole numerical value.

1.8 Operations

They make it possible to calculate values according to the operator and operands. The operations available are as follows:

Operator	Description	Example
+	Addition, if the two operands are numerical values, and concatenation of chains if one of the two operands is character strings.	`3 + 5` `"toto"+"titi"` `3+"titi"`
-	Subtraction, applies only to numerical values.	`2133 – 5`
*****	Multiplication, applies only to numerical values.	`532 * 873`
/	Division, applies only to numerical values.	`100 / 9`
%	The modulo, applies only to numerical values, and makes it possible to obtain the remainder of division.	`100 % 9`

1.9 Functions

A function is an embedded process that returns a value. A function is always presented between hooks in the following way: [name ARG...]

This is some basic functions commonly used:

- **[sub GRAPH]:** The function sub makes it possible to call a particular graph. The function returns like value the text generated by the graph. This text is not inserted in the result; it is just affected with the value of the function.

Example	Explanation
`$nom = [sub subGraph]`	The graph "subGraph" is called, and the variable $nom takes as value the text generated by this graph.

- **[sub GRAPH param1 param2 … paramN]:** This function will extend the function **[sub GRAPH]** the paramX will define the value of $paramX variable inside the GRAPH, the integer after param is defined by its position inside the List, 1 is the first param after GRAPH name. If the X value is out of range the value set will be null. This variable are ***local*** to the graph it means that $paramX is only set by the value given to the function even if a variable $paramX is set in the calling graph or upper.

Example	Explanation
`$return = [sub GraphName 1 'a string' $vName]`	The graph "GraphName" is called, and the variable $return takes as value the text generated by this graph. With this variable defined like this: • $param1==1 > Numerical • $param2=='a string' > String • $param3==$vName > Same type as $vName

- **[subprefix GRAPH PREFIX]:** The function subprefix allows, just like the function sub to call a particular graph, but in this case, all the variables name affected by this graph are modified, and a prefix is added. The first argument of the function is the name of the graph, and the second argument is the préfix added to all the names of variables used in the left part of an assignment (in $val = $v1 + $v2), $val is known as the left part, and $v1 + $v2 is known as the right part).

Exemple	Explanation
`[subprefix nomcomplet heros]`	The graph "**nomcomplet**" is called, and the word "**heros**" is used like prefixes of all the variables calculated in the graph. If for example the graph calculates two variables, $nom = "Valjean", and $prenom = "Jean", then with leaving the graph, the variables $nom and $prenom will not have been affected. On the other hand, the two variables herosnom and herosprenom will have taken respectively the values "Valjean" and "Jean".
`$val = [subprefix nomcomplet heros]`	As for the preceding example, the variables calculated in the graph "**nomcomplet**" take the prefix "heros", but in more all the text generated normally by the graph is affected with the variable $val.

- **[question 'the question' '*' or lists answers possible or '#()']** : The function question makes it possible to raise a question with the user at the time of the execution of the generator, the answer is turned over in the graph as a value which one can assign to a variable.

  Question allow also to upload file from user Hard Disk, in this the second argument has to be #() or #(mask), if mask is provided like *.jpg or *.* the windows will allow to select only the file witch fit with the mask.

Example	Explanation
`$Ste = [question 'What is the name of your company' '*']`	At the time of the execution of the graph the question is put and the engine is stopped until the answer is recorded by the user. The variable $Ste then takes the returned value.
`$Color = [question 'Choose a color' 'Red,Blue,Green']`	In this example the mechanism is the same than the previous example, but in this case the user has a choice between the tree value Red,Blue,Green.
`$FileName = [question 'Select a file' '#(*.*)']`	This example show how to get a file name from the question to use it next in the graph.

- **[convert 'type' 'VALUE']** : This function convert the VALUE into the type defined by the type parameter.

Type	Description	Example
HexColor	HexColor makes it possible to transform a color represented in Hexadecimal form into integer value.	#000000 -> 0
IntColor	IntColor do the opposit function than HecColor	0 ->#000000
Base64	In this case **VALUE** represent a file, the system read it and convert it into a Base64 format.	[convert 'Base64' './logo.jpg']

- **[writeTo 'FileName']** : The function makes it possible to write the result of the current state of the graph into a file whose name is specified by FileName, it can be a chain or a variable. The closing of the file is carried out at the time of the opening of another file or at the end of the execution of the graph.

- **[writeVarTo 'FileName' 'VALUE']** : This function write the VALUE parameter into a file specified by FileName.

- **[guid]:** This function return a unique identifier compatible with the internal VDOM unique Identifier.

1.10 Advanced variable declaration

It is possible to put a variable like name of variable, so as to cause a double dereferenciation. For example ${$toto}, if 'xxx' is the value of $toto, it will be equivalent to $xxx. It is possible to put complete expressions between the {}. For example: ${$toto + "-" + ${titi+1}} is a valid name of variable.

2. VDOM XML File.

2.1 Description of the different XML sections.

```
 <?xml version="1.0" encoding="ISO-8859-1" ?>

<Application>
               <Information>
                              <Active>1</Active>
                              <Description>-</Description>
                              <ID>ab579906-5eed-4626-bc53-4cc27786b37c</ID>
                              <Index>f4fa85a0-b0fb-4425-b135-d2709d887bca</Index>
              </Information>
              <Structure>
                              <Object ID="f37a20e4-873c-48b0-94c6-744929afac7c" Top="0" Left="0" ResourceID="">
                                             <Level Index="0" />
                              </Object>
                              <Object ID="f4fa85a0-b0fb-4425-b135-d2709d887bca" Top="0" Left="0" ResourceID="">
                                             <Level Index="0">
                                                            <Object ID="a8ac0f13-4f0d-46ce-9f3e-aaf49615b858" />
```

```
            <Object ID="7ed82bdf-aef7-4176-a571-0f5fa6799ae4" />
            <Object ID="69e12547-a4f5-4032-a09e-4c5fc40757ca" />
            <Object ID="c3be8d80-c021-40d2-9d5f-73aee411eaa2" />
            <Object ID="6bedf832-2aaf-40bb-9063-237f0bd7ea41" />
            <Object ID="3f5d9839-d3ec-48fa-b3d2-fd5008b04bd9" />
        </Level>
    </Object>
    …..
</Structure>
<Objects>
    <Object ID="a8ac0f13-4f0d-46ce-9f3e-aaf49615b858" Type="2330fe83-8cd6-4ed5-907d-
11874e7ebcf4" Name="object_a8ac0f13_4f0d_46ce_9f3e_aaf49615b858">
        <Attributes>
            <Attribute Name="title">Contacts</Attribute>
            <Attribute Name="hierarchy">1</Attribute>
            <Attribute Name="visible">1</Attribute>
        </Attributes>
        <E2VDOM>
            <EVENT objSrcID='' name='' XPosition='10' YPosition='20' >
                <ACTION objTgtID='' methodName='name' XPosition='40'
                YPosition='40'>
                    <PARAMETER scriptName='MyParameter'></ PARAMETER>
                    ….
                </ACTION>
            </EVENT>
        </E2VDOM>
        <Objects>
            <Object Type="0d36c35d-9508-440f-bfec-668f3db8cfeb" ID="c0ad24a3-27b9-
            4687-9f81-ef50c9b1a920"
            Name="object_c0ad24a3_27b9_4687_9f81_ef50c9b1a920">
                <Attributes>
                    <Attribute Name="visible">1</Attribute>
                    <Attribute Name="height">41</Attribute>
                    <Attribute Name="value">#Res(3c58c602-c358-4af0-a2ab-
                    a6be783a65a1)</Attribute>
                    <Attribute Name="width">840</Attribute>
                    <Attribute Name="top">727</Attribute>
                    <Attribute Name="zindex">7</Attribute>
                    <Attribute Name="hierarchy">2</Attribute>
                    <Attribute Name="left">0</Attribute>
                    <Attribute Name="visible">1</Attribute>
                </Attributes>
                <Objects />
        </Object>
        ……
</Objects>
<Resources>
    <Resource Type="gif" ID="3c58c602-c358-4af0-a2ab-a6be783a65a1" Name="3c58c602-c358-4af0-
    a2ab-a6be783a65a1">
    <![CDATA[
        R0lGODlhSAMpAPcAAOPk5urs7uzq7+fo6u/w8tra2+zs7vb2+Ojm6u7w8tvc3fj5+uTm6N
        9Obk6PX29tze4PP09eTi5vf4+eDi5O3r8Pn6+/v8/ODf4vDt8uTl6O/s8vHy8+3u7u3s8OLh
        5Pb3+ejp7OTm5/X29+Lg4/Lu9Ofm6t7f4tzd4O7w8e7s7+bk5+/u8vb19 …..
    ]]>
    </Resource>
    <Resource Type="jpg" ID="2e0a791d-65b4-4616-9002-3e61c6d37dcc" Name="2e0a791d-65b4-
    4616-9002-3e61c6d37dcc">
    <![CDATA[
        kZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAPAAA/+4ADkFkb2JlAGTAAAAAA
        EBAUEBgUFBgkGBQYJCwgGBggLDAoKCwoKDBAMDAwMDAwQDA4PEA8ODBMTF
        GxscHx8fHx8fHx8fHwEHBwcNDA0YEBAYGhURFRofHx8fHx8fHx8fHx8fHx8…..
    ]]>
    </Resource>
    …..
```

```
        </Resources>

</Application>
```

- **<Application>:** Main tag starting the application section.
    - **<Information>:** General information about the application.
        - **<Active> :** Is this application will be considered as an active one or not.
        - **<Description> :** Test description of the application.
        - **<ID>:** Unique identifier of this application.
        - **<Index>:** Witch Top Level container is the default one.

    - **<Structure>:** Section describing the link between the top level containers. This section stores the architecture of the application. The structure is a collection of top level container connected with arrows that define a source object and a target one.
        - **<Object ID='' Top='x' Left='y' ResouceID='Guid'> :** This is the source object of this structure, ID is the reference of the object, Top|Left give the position in the IDE and ResourceID represent the picture associate the this top level container in the IDE.
            - **<Level Index='n'> :** The connection between the objects can be done thru multiple group of links (7), index represent the group of this links.
                - **<Object ID='Guid' />:** Target object, ID is its reference.

    - **<Objects>:** This section describe the object structure of the VDOM Application, this structure is a hierarchical composition of objects, on the first level we have the Top Level object that the main container and can represent the page. In fact it's the only object that you can reach thru url. Inside this upper level container you can have an infinite depth of object construction.
        - **<Object ID='Guid' Type='Guid' Name='???'> :** First level object, it means this one is a Top Level container, not all object can be Top Level container, you have for example HTML,Flash, PDF, etc … This is define in the object TYPE.

            - **<Attributes> :** Attributes section of this object, each object has a collection of attributes, this section define the value of each one.
                - **<Attribute Name='????' >:** An attribute, the name has to correspond to the name defined in the Object Type.
            - **<E2VDOM> :** Each container can implement an event driven behaviour, this section define the event processed by the engine and the action associated but only for the client -> client type, the client -> server -> client event are managed inside the script.
                - **<Event objSrcID='Guid' name='???' XPosition='xx' YPosition='yy'>:** Event catch by the engine, objSrcID is the reference of the object that can raise this event, the object can't be inside an other container. XPosition & YPosition define the position in the IDE.
                    - **<Action objTgtID='Guid' methodName='name' XPosition='xx' YPosition='yy'>:** One or more action associated to this event.
                        - **<Parameter scriptName='MyParameter'>:** If this action allow parameters this tag define the value.
            - **<Objects> :** Second level of objects can be terminal object or container.
                - **<Object ID='Guid' Type='Guid' Name='???'>:** Define the characteristics of the object.
                    - **<Attributes>:** Attribute section of this object.
                        - **<Attribute Name='???'>**Value of each attribute.
                - **<Objects />:** If there is a new level of object this is the start of this new section, if there is not this tag can be used but it's optional.
    - **<Resources>:** The VDOM XML application file save also the binary resources, to fit with a text format each resources are converted in Base64 format. This tag define the start section of resources.
        - **<Resources Type='???' ID='Guid' Name='????'><![CDATA[Base64 data]]>:** One resource, type define the original file type like jpg,gif,etc … ID is the internal GUID for reference and name is a friendly one which is used in the resources browser.

2.2 General graph & Matrix importation of the XML structure

MAIN GENERAL GRAPH

The importation from XML will generate several Graphs with a logical Structure, the first high level structure is the general application Graph, the name of this main Graph will be VDOMApplication and it is shown on Figure 4.

It represents the mains XML node of the VDOM XML Application file, the application start with <Application> and then we have the node <Information>, no option is given for the <Active> node that it switch to 1.

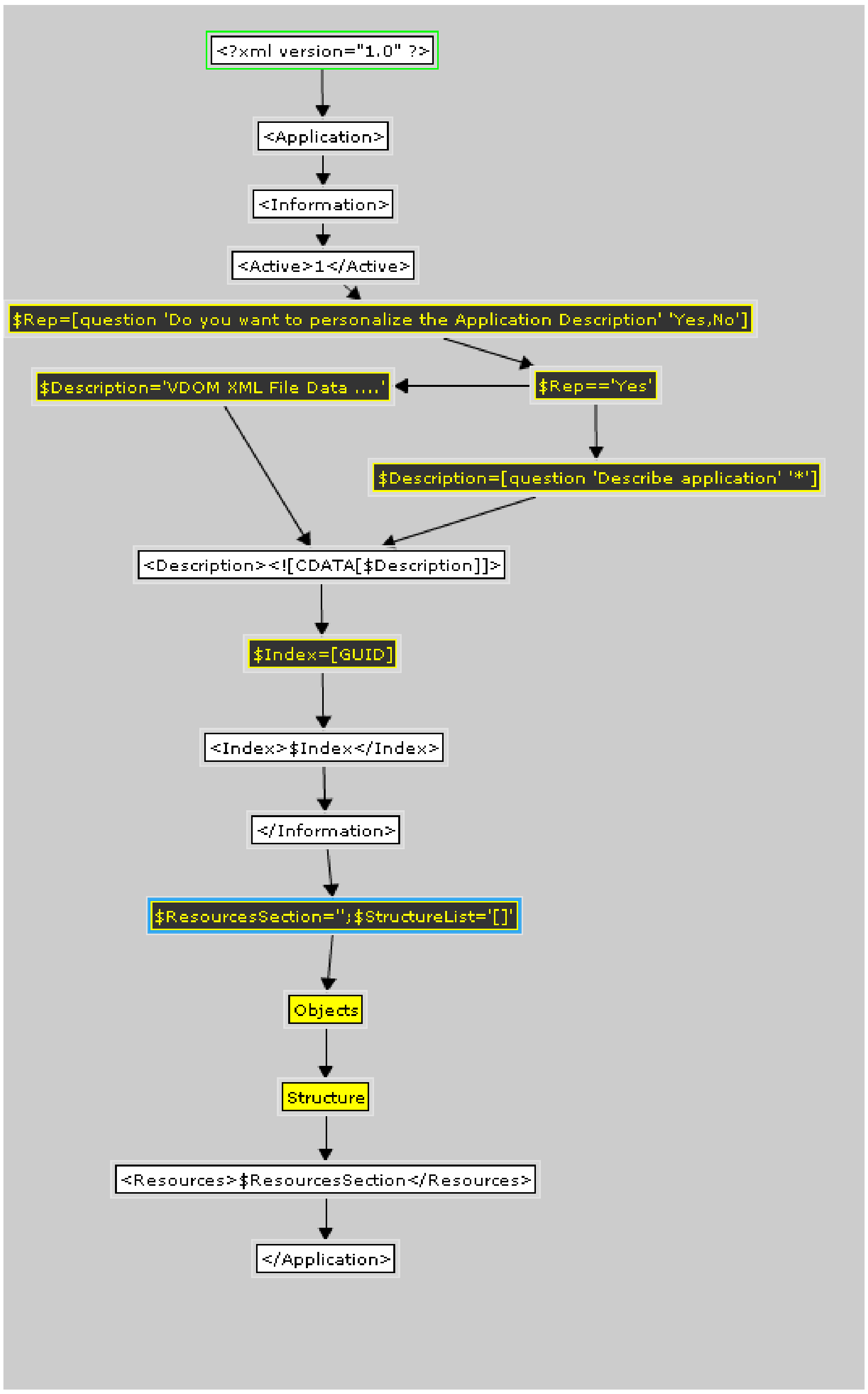

<?xml version="1.0" ?>
<Application>
<Information>
<Active>1</Active>
$Rep=[question 'Do you want to personalize the Application Description' 'Yes,No']
$Description='VDOM XML File Data ....'
$Rep=='Yes'
$Description=[question 'Describe application' '*']
<Description><![CDATA[$Description]]>
$Index=[GUID]
<Index>$Index</Index>
</Information>
$ResourcesSection='';$StructureList='[]'
Objects
Structure
<Resources>$ResourcesSection</Resources>
</Application>

Figure 4: General Application Graph

The first option that can be defined by the user is the Description of this application; we give the opportunity to the user to input his own description.

In the XML file the node index define the first Top level container that will be shown if the URL doesn't define it. This GUID is unknown at this stage because the object section comes after, so we generate a GUID to be used after as one of a Top level container to set it as the HOME PAGE.

Next we initialize two variables that will be used to store the structure and Resources data. Those information are generated during the process of object section generation, to be flexible we are not going to write (Structure/Resources) as static graph but as a structured data stored in a variable to be able to modify it easily after the importation.

Resources Data

The resource section will be stored in $ResourcesSection variable, the importation will use one sub Graph to set the value.

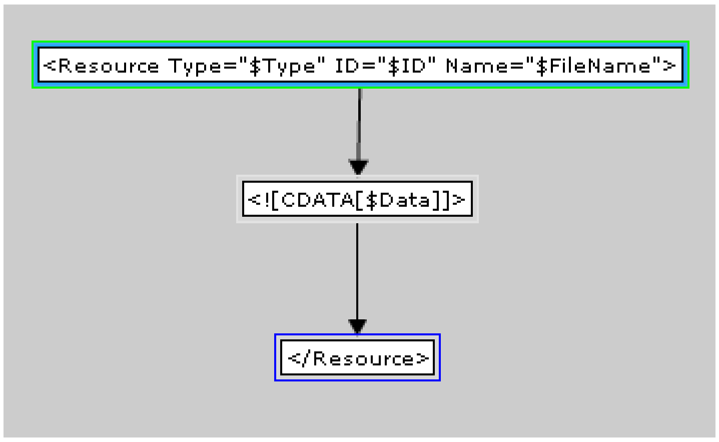


Figure 5: SetResource Graph

The Figure 5 show the simple graph to generate the Resource node of the XML file, we can see several variables $Type, $ID, $FileName and $Data. The calling graph has to set them before the call like on Figure 3. All the data come from the VDOM Application XML File.

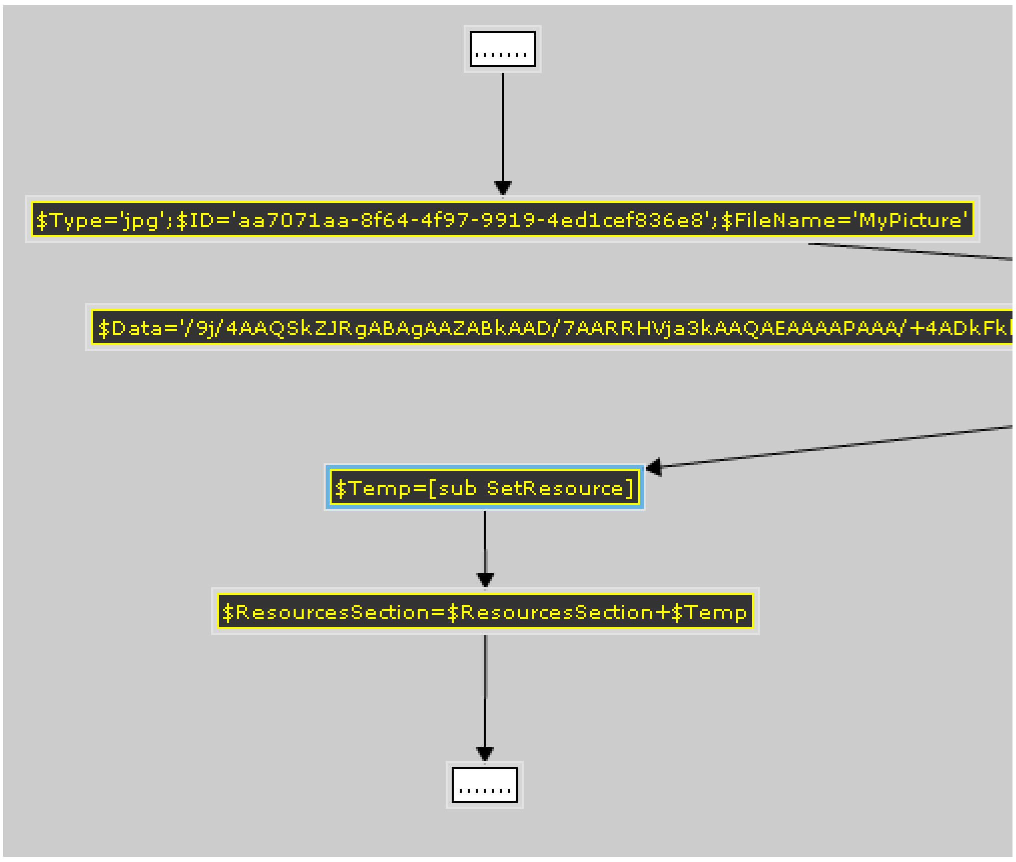

....
$Type='jpg';$ID='aa7071aa-8f64-4f97-9919-4ed1cef836e8';$FileName='MyPicture'
$Data='/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAPAAA/+4ADkFk
$Temp=[sub SetResource]
$ResourcesSection=$ResourcesSection+$Temp
....

Figure 6: General Call to generate Resources XML Node

An alternative graphs use the enhanced **[Sub ...]** function. The following example (Figure 7) show the same graph as figure 6 using $paramX variable.

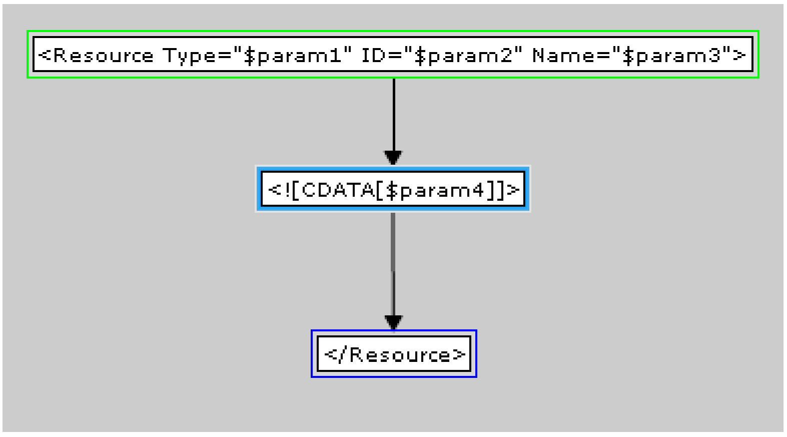


Figure 7: SetResource Graph using param

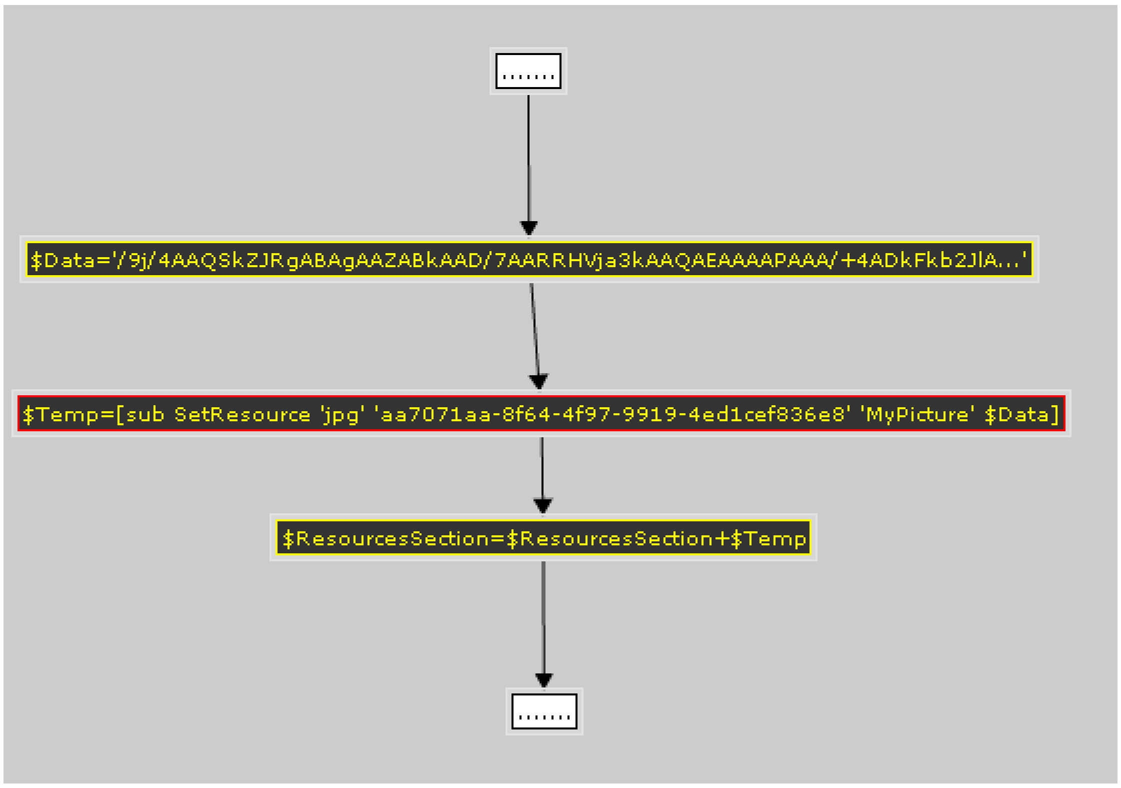

....
$Data='/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja3kAAQAEAAAAPAAA/+4ADkFkb2JlA...'
$Temp=[sub SetResource 'jpg' 'aa7071aa-8f64-4f97-9919-4ed1cef836e8' 'MyPicture' $Data]
$ResourcesSection=$ResourcesSection+$Temp
....

Figure 8: Graph Call using enhanced sub function

Structure Section

The structure section define how the Top Level Containers are connected each other, this is a XML example of this section.

```
<Structure>
        <Object ID="760b051c-e0f3-430c-9d5a-a07831275d6a" Top="0" Left="0" ResourceID="">
                <Level Index="0">
                         <Object ID="90fa5627-ee08-4d4e-aa41-75682f2dc142" />
                </Level>
        </Object>
        <Object ID="3739e938-0559-4358-bb5d-e72d0c4f6e7a" Top="0" Left="0" ResourceID="">
                <Level Index="0">
                        <Object ID="1e8019d0-3213-4f07-9007-fadf382ceb85" />
                        <Object ID="ec9494fe-e5cc-4a9e-9012-8b5a04f58762" />
                        <Object ID="cf4ec7aa-edf6-4561-9aeb-ac4fc6e3258e" />
                        <Object ID="5324bda5-f053-46d4-8141-e123eb8b4cee" />
                        <Object ID="760b051c-e0f3-430c-9d5a-a07831275d6a" />
                        <Object ID="60dd6a22-b502-4121-aef8-b15116d5327e" />
                </Level>
        </Object>
        <Object ID="78bd0bcc-70a8-4593-a559-dcedd2635e03" Top="0" Left="0" ResourceID="">
                <Level Index="0">
                        <Object ID="78bd0bcc-70a8-4593-a559-dcedd2635e03" />
                </Level>
        </Object>
        ....

</Structure>
```

The graph conversion of this structure will be split in two parts, a graph witch manipulate de data structure during the object section generation and a graph engine that will convert the data structure into XML structure.

List Data Structure

The PowerPack Generator is able to manipulate List data structure, so it will store all the structure into Lists and sub Lists.

- Main List -

```
[TopLevelList_1 TopLevelList_2 …. TopLevelList_n]
```

TopLevelList_x(***List***): Sub list of each Top level container.

- TopLevelList_x -

```
[ObjectSrcList LevelLinkList]
```

ObjectSrcList(***List***): Sub list representing information about the top level container source of the link.

LevelLinkList(***List*****):** Sub list representing the level number and all the targeted Top Level Container

- ObjectSrcList -

```
[IdObject Top Left ResID]
```

IdObject(***Word***) : Top Level ID Source object

Top(***Word***) : Top position of this object

Left(***Word***) : Left position of this object

ResID(***Word***) : Resource ID of the picture shown on tree extended mode.

- LevelLinkList -

```
        [Level_0_ObjTgtList Level_1_ObjTgtList …
Level_n_ObjTgtList ]
```

Level_x_ObjTgtList(***List***) : The sub list with level value and list of targeted objects.

- Level_x_ObjTgtList -

```
[Level ObjTgtList]
```

Level(***Word***): Integer representing the level link.

ObjTgtList(***List***): Sub list of targeted objects.

- ObjTgtList –

  `[IdObjTgt1 IdObjTgt2 ... IdObjTgtn]`

  IdObjTgtx(***Word***): GUID representing the targeted object.

GRAPH ENGINE TO ADD/UPDATE/REMOVE DATA IN GRAPH LIST STRUCTURE

The graph will be called by this sub function:
[Sub **AddStructure** ObjSrc ObjTgt Level [StructureList]]

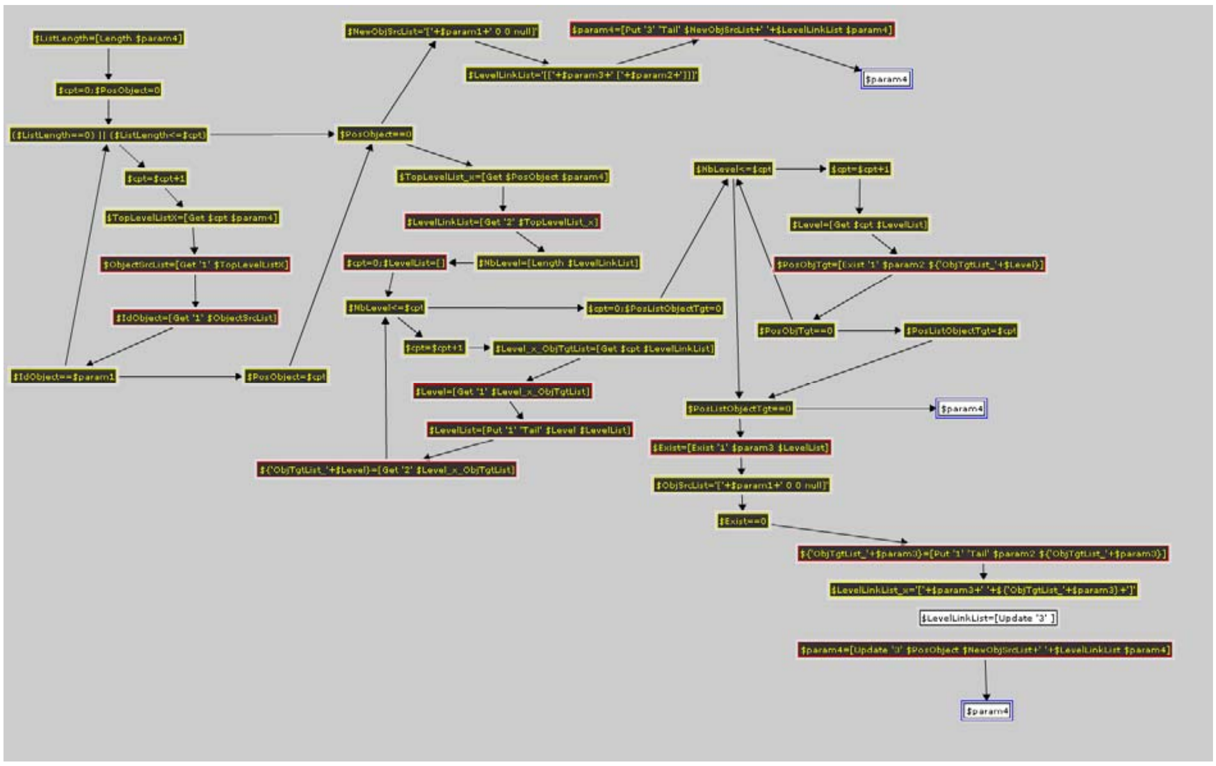

$ListLength=[Length $param4]
$cpt=0;$PosObject=0
($ListLength==0) || ($ListLength<=$cpt)
$cpt=$cpt+1
$TopLevelListX=[Get $cpt $param4]
$ObjectSrcList=[Get '1' $TopLevelListX]
$IdObject=[Get '1' $ObjectSrcList]
$IdObject==$param1
$PosObject=$cpt
$PosObject==0
$NewObjSrcList='['+$param1+' 0 0 null]'
$LevelLinkList='[['+$param3+' ['+$param2+']]]'
$param4=[Put '3' 'Tail' $NewObjSrcList+' '+$LevelLinkList $param4]
$param4
$TopLevelList_x=[Get $PosObject $param4]
$LevelLinkList=[Get '2' $TopLevelList_x]
$NbLevel=[Length $LevelLinkList]
$cpt=0;$LevelList=[]
$NbLevel<=$cpt
$cpt=$cpt+1
$Level_x_ObjTgtList=[Get $cpt $LevelLinkList]
$Level=[Get '1' $Level_x_ObjTgtList]
$LevelList=[Put '1' 'Tail' $Level $LevelList]
${'ObjTgtList_'+$Level}=[Get '2' $Level_x_ObjTgtList]
$cpt=0;$PosListObjectTgt=0
$NbLevel<=$cpt
$cpt=$cpt+1
$Level=[Get $cpt $LevelList]
$PosObjTgt=[Exist '1' $param2 ${'ObjTgtList_'+$Level}]
$PosObjTgt==0
$PosListObjectTgt=$cpt
$PosListObjectTgt==0
$param4
$Exist=[Exist '1' $param3 $LevelList]
$ObjSrcList='['+$param1+' 0 0 null]'
$Exist==0
${'ObjTgtList_'+$param3}=[Put '1' 'Tail' $param2 ${'ObjTgtList_'+$param3}]
$LevelLinkList_x='['+$param3+' '+${'ObjTgtList_'+$param3}+']'
$LevelLinkList=[Update '3']
$param4=[Update '3' $PosObject $NewObjSrcList+' '+$LevelLinkList $param4]
$param4

Figure 9: Graph Engine to add a New Link to the Structure

Others sub graph useful in the case of importation:

[Sub **UpdateStructure** ObjSrc ObjTgt Level [StructureList]]: This subGraph updates the level link between the two objects.

[Sub **DeleteStructure** ObjSrc ObjTgt Level [StructureList]]: This SubGraph deletes a link between two objects.

Power Pack Advanced functions

In a normal Web application most of time there is a lot of pictures and it could be sometimes needed to be able to manipulate this pictures this is the goal of the graphic functions. The power pack needed also to have an internal data structure with enough function to manipulate it, this is the goal of list functions.

1. Advanced graphic functions

The function structure is the same than the other functions like this: [NOM ARG ...]. Until now the variable could be two types Number or String, even if we still work with only this two type to manipulate pictures, it's better to introduce something like a pointer to an internal object that represent the picture.

The input format of the picture can be JPG,BMP,PNG,GIF the output format will be PNG.

SUB-LIST USED AS PARAMETERS FOR FUNCTIONS :

- **[Pen PenColor PenStyle PenWidth] :** This function define the pen used for functions witch use it.
  PenColor : Numeric value from 0 to 16777215 for a depth of 24 bits
  PenStyle: Define the style of the pen regarding the following table.

0 – Solid	Solid Line	————————
1 – Dash	A line made up of a series of dashes.	—— —— ——
2 – Dot	A line made up of dots	- - - - - - - - -
3 – DashDot	A line made up of alterning dashes and dots.	- - — - —
4 – DashDoDot	A line made up of a serious of dash-dot-dot combinations.	- - - — - - —
5 – Clear	No line is drawn (used to omit the line around shapes that draw an outline using the current pen).	
6 – InsideFrame	A solid line, but one that may use a dithered color if Width is greater than 1	

PenWidth : Define the size in pixel of the pen.

- **[Brush BrushStyle BrushColor] :** This function define a brush style to fill a form

BrushStyle :

0 – Solid	■	
1 – Clear	□	
2 – BDiagonal	▧	
3 – FDiagonal	▨	
4 – Cross	▦	
5 – DiagCross	▩	
6 – Horizontal	▤	
7 – Vertical	▥	

BrushColor : Numeric value from 0 to 16777215 for a depth of 24 bits

- **[GradiantOneWay BeginColor EndColor Direction] :** Generate a one Way gradient from the color BeginColor to the EndColor with a direction define in degree 0 mean Horizontal gradient.
- **[GradiantTwoWay BeginColor EndColor Direction] :** Generate a two way gradient from the color BeginColor to the EndColor with a direction define in degree 0 mean Horizontal gradient.
- **[Texture $ObjectPicture] :** Define a texture to fill the form.
- **[Filter] :**
  BevelFilter, BlurFilter, ColorMatrixFilter, ConvolutionFilter, DisplacementMapFilter, DropShadowFilter, GlowFilter, GradientBevelFilter, GradientGlowFilter

- **[Font 'Name' Size Color Bold Italic Underlined] :** Define a font used by any function using text.

MAIN GRAPHIC FUNCTIONS :

- **[LoadPicture 'Path'] :** This function allow to load into memory a picture from file it return a pointer into the variable witch it used for other manipulations.

Example	Explanation
`$ObjPicture = [LoadPicture './Mupicture.Jpg']`	Load a picture from file.

- **[CreatePicture Width Height BackgroudColor]** : This function create a new empty picture with the color set by BackgroudColor (numeric value), it return the pointer to the Object created.

Example	Explanation
`$ObjPicture = [CreatePicture 100 100 0]`	Create a picture of 100x100 pixels with a Black background

- **[GetWidth $ObjPicture] :** Return the width of the picture stored in $ObjectPicture.

Example	Explanation
`$PWidth = [GetWidth $ObjPicture]`	As explained get Width !!!

- **[GetHeight $ObjPicture] :** Return the hieght of the picture stored in $ObjectPicture.

Example	Explanation
`$PWidth = [GetWidth $ObjPicture]`	As explained get height !!!

- **[AddPicture $ObjPicture1 $ObjectPicture2 X Y Transparency TransparencyColor] :**

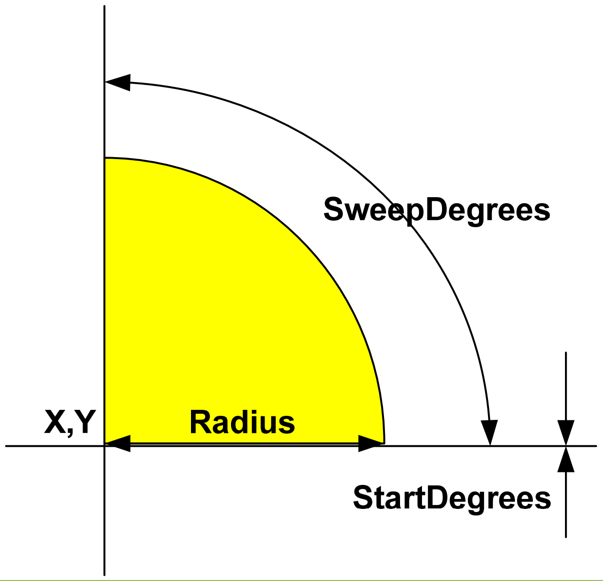
  Add image $ObjectPicture2 to $ObjectPicture1 located to X Y.

  Transparency : Percentage of transparency of the Picture2 on the Picture1

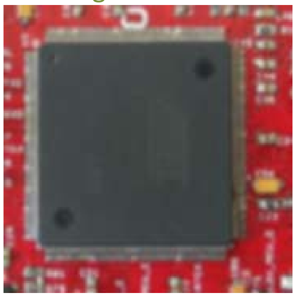
  TransparencyColor : Define the transparent color of the Picture2

Example	Explanation
`$ObjPct = [AddPicture $ObjPicture 10 10 70 2546]`	This function return a pointer to a picture object , if the variable is the same the function will modify the current object picture. Transparency is 70 %, the range is from 0 to 100, 0 means no transparency and 100 means full transparency. Color is the int value of the equivalent Hex.

- **[DrawAngleArc $ObjPicture X Y Radius StartDegrees SweepDegrees Pen Fill] :** The DrawAngleArc function draws 2 lines segment and an arc. The line segment is drawn from the X,Y position to the beginning of the arc. The arc is drawn along the perimeter of a circle with the given radius and center (X,Y). The length of the arc is defined by the given start and sweep angles. Pen is defined by the Pen function and Fill by any function that witch define a content.

Example	Explanation
`$ObjPct = [DrawAngleArc $ObjPicture 100 100 50 0 90 [Pen 0 0 1] [Brush 0 16776960]]`	This Draw Angle Arc located on X,Y 

- **[Blur $ObjPicture Times] :** Apply a blur filter to the picture, times parameter define how many times this effect will be apply to the picture.

Example	Explanation
`$ObjPct = [Blur $ObjPicture 5]`	Original Picture  Picture after blur filter

- **[BrightenImage $ObjPicture Degree] :** Transform the current picture by increasing or decreasing the luminosity level, positive number increase, negative one decrease.

Example	Explanation
`$ObjPct = [BrightenImage $ObjPicture 100]`	Original Picture Picture after Bright filter

- **[Contrast $ObjPicture Degree] :**
- **[CreateBlackWhite $ObjPicture] :**
- **[CreateGrayScale $ObjPicture] :**
- **[CreateNegative $ObjPicture] :**
- **[CropImage $ObjPicture StartX StartY Width Height] :**
- **[DarkenImage $ObjPicture Degree] :**
- **[Emboss $ObjPicture] :**
- **[Merge $ObjPicture1 $ObjPicture2 Percent] :**
- **[DrawEllipse $ObjPicture1 X1 Y1 X2 Y2 Pen Fill Transparency] :**
- **[DrawRect $ObjPicture X1 Y1 X2 Y2 Pen Fill Transparency] :**
- **[FlipImage $ObjPicture Direction] :**
- **[GetImage FileSize ‘PathFile’] :**
- **[GetPixel $ObjPicture X Y] :**
- **[DrawLine $ObjPicture X1 Y1 X2 Y2 Pen Transparency] :**
- **[DrawBezier $ObjPicture [[X1 Y1] [X2 Y2] …. [Xn Yn]] PenStyle Pen Transparency] :**
- **[FillBezier $ObjPicture [[X1 Y1] [X2 Y2] …. [Xn Yn]] PenStyle Pen Fill Transparency] :**
- **[DrawPolygon $ObjPicture [[X1 Y1] [X2 Y2] …. [Xn Yn]] Pen Fill Transparency] :**
- **[FillPolygon $ObjPicture [[X1 Y1] [X2 Y2] …. [Xn Yn]] Pen Fil Transparency] :**
- **[DrawPolygon $ObjPicture[[X1 Y1] [X2 Y2] …. [Xn Yn]] Pen Fill Transparency] :**
- **[Resize $ObjPicture Width Height] :**
- **[ResizeResampling $ObjPicture Width Height] :**
- **[Rotate $ObjPicture Degree BackGroundColor] :**
- **[DrawRoundRect $ObjPicture X1 Y1 X2 Y2 PenStyle PenColor FillColor Transparency] :**
- **[Saturation $ObjPicture Degree] :**

- **[SetPixel $ObjPicture X Y Color] :**
- **[Saturation $ObjPicture Value] :**
- **[WriteText $ObjPicture X Y Width Height Font FontColor FontSize Transparency TextAlign Bold Italic] :**

2. Advanced List manipulation functions

Before defining this function we have to define the Position parameter in a list.

Position: Can be an integer and the value define the position is like this [Pos:1 Pos:2 … Pos:n] or Head/Tail, Head position is like Integer value 1, Tail has different value depending the function, for GET/UPDATE it's the element with position n (the last one), for PUT function it's n+1, it means we add a new element at the end of the list.

- **[Get Position [List]] :** Return the elt of the list at the position given by Position parameter.

Example	Explanation
`$elt = [Get '2' [elt1 elt2 elt3]]`	This example return the value elt2
`$elt = [Get 'Head' [elt1 elt2 elt3]]`	This example return the value elt1
`$elt = [Get 'Tail' [elt1 elt2 elt3]]`	This example return the value elt3

- **[Put Type Position Value [List]] :** Put the value at the position indicated by Position parameter and move all the other element to the left, if tail is used it add the element at the end of the list. All the parameters are String except the list. The Type parameter define in which type this value will be added:
  - **1 -** Word
  - **2 -** String
  - **3 -** List
  - **4 -** Variable

Example	Explanation
`$elt = [Put '1' '2' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 eltN elt2 elt3]
`$elt = [Put '1' 'Head' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [eltN elt1 elt2 elt3]
`$elt = [Put '1' 'Tail' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 elt2 elt3 eltN]
`$elt = [Put '2' '2' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 'eltN' elt2 elt3]
`$elt = [Put '3' '2' 'eltN eltM' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 [eltN eltM] elt2 elt3]
`$elt = [Put '4' '2' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 $eltN elt2 elt3]

- **[Update Type Position Value [List]] :** The update function replace an element of the list by another one, all the parameters of Get function are the same here, type value, position is an integer or Head/Tail, value is a string.

Example	Explanation
`$elt = [Update '1' '2' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 eltN elt3]
`$elt = [Update '1' 'Head' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [eltN elt2 elt3]
`$elt = [Update '1' 'Tail' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 elt2 eltN]
`$elt = [Update '2' '2' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 'eltN' elt3]
`$elt = [Update '3' '2' 'eltN eltM' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 [eltN eltM] elt3]
`$elt = [Update '4' '2' 'eltN' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 $eltN elt3]

- **[Delete Position [List]]:** The function delete remove an element at the given position. Position support integer and Head/Tail value.

Example	Explanation
`$elt = [Delete '2' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 elt3]
`$elt = [Delete 'Head' [elt1 elt2 elt3]]`	The result in $elt variable is [elt2 elt3]
`$elt = [Delete 'Tail' [elt1 elt2 elt3]]`	The result in $elt variable is [elt1 elt2]

- **[Length [List]]:** This function return the number of element inside the given List.

Example	Explanation
`$elt = [Length [elt1 elt2 elt3]]`	The result in $elt variable is 3
`$elt = [Length ['elt1' $elt2 [elt3 elt4] elt5]`	The result in $elt variable is 4

- **[GetType Position [List]]:** This function returns the integer value representing the element type inside the given List at Position.

  Type value returned by this function.
  - **1 -** Word
  - **2 -** String
  - **3 -** List
  - **4 -** Variable

Example	Explanation
`$elt = [GetType 'Head' ['elt1' $elt2 [elt3 elt4] elt5]]`	The result in $elt variable is 2
`$elt = [GetType '2' ['elt1' $elt2 [elt3 elt4] elt5]]`	The result in $elt variable is 4
`$elt = [GetType '3' ['elt1' $elt2 [elt3 elt4] elt5]]`	The result in $elt variable is 3
`$elt = [GetType 'Tail' ['elt1' $elt2 [elt3 elt4] elt5]]`	The result in $elt variable is 1

- **[Mid Start Length 'String to cut']:** Return a part of the string beginning at the start char (0 is the first value) with a length of length char.

- **[Split 'delimiter' 'String to split']:** Return a list of elt like [elt1 elt2 ... eltn] using the delimiter as character for spliting.
- **[Exist Type Value [List]]:** Return 0 if this value doesn't exist in the current list or the position of the elt matching in the list.

Power Pack Debug Mode

The implementation of Complex Graph in the Power Pack Builder could represent a difficult task, just running the graph to see the result is not enough to understand where your mistakes are. Moreover, some graphs can run as infinite loop and completely freeze the software. To avoid such trouble and allow the PowerPack developer to be efficient we need an option of graph debugging.

The Debug option will take place in Run menu like it shown below:

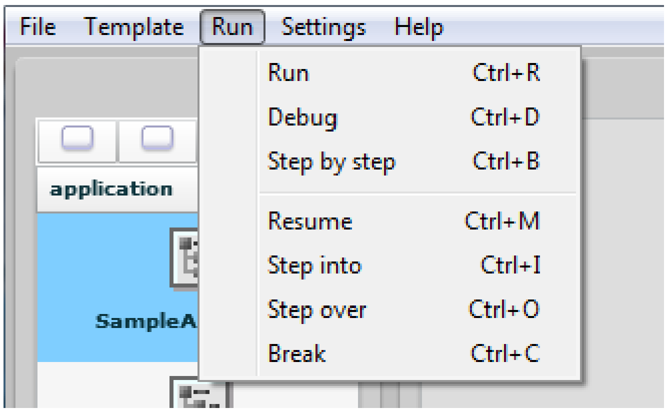
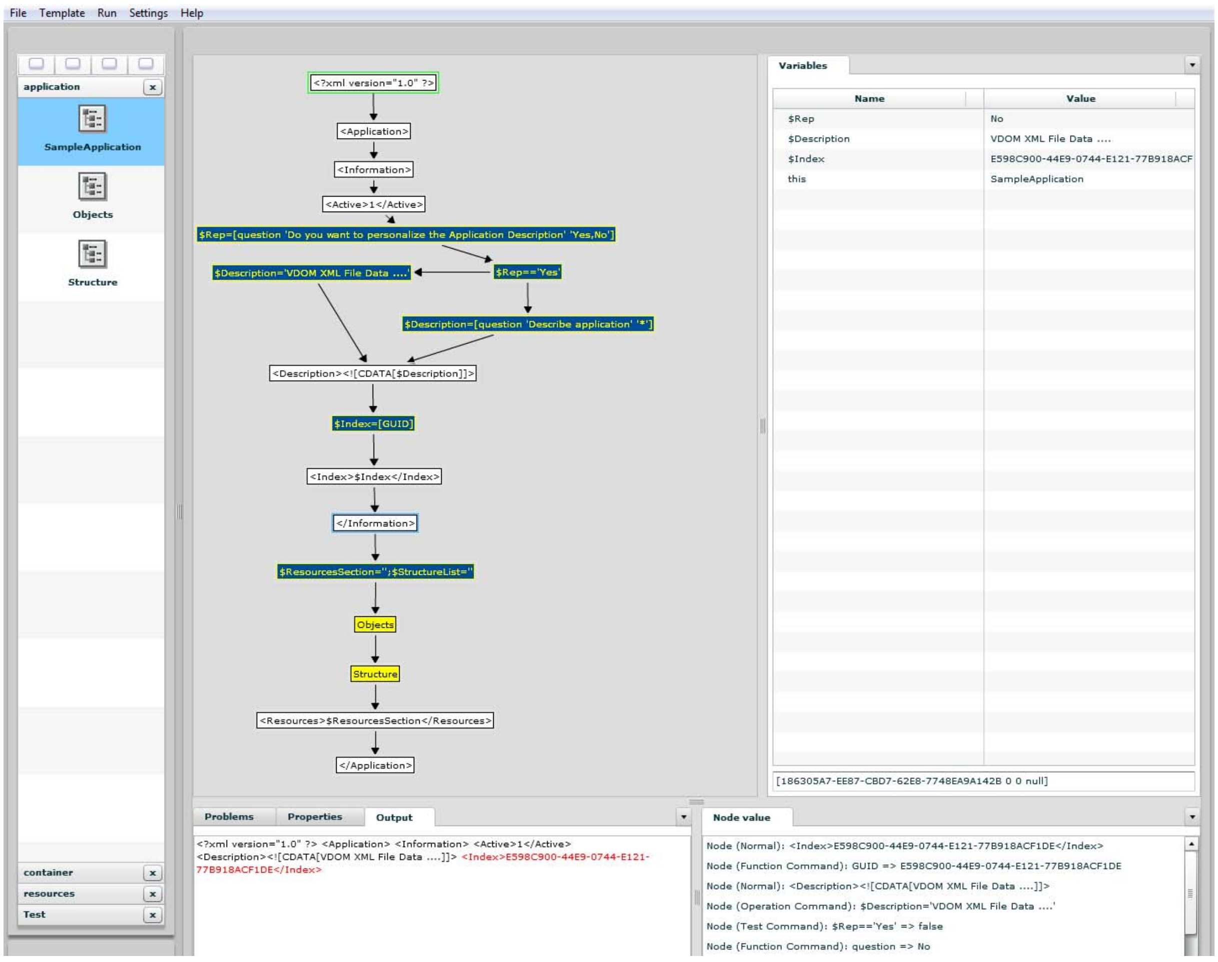


Figure 10: Generator debug option.

When the (Debug) or (Step by Step) option is launched instead of Run, the PowerPack Builder will execute the graph in debug mode:


File Template Run Settings Help
application
SampleApplication
Objects
Structure
container
resources
Test
<?xml version="1.0" ?>
<Application>
<Information>
<Active>1</Active>
$Rep=[question 'Do you want to personalize the Application Description' 'Yes,No']
$Description='VDOM XML File Data ....'
$Rep=='Yes'
$Description=[question 'Describe application' '*']
<Description><![CDATA[$Description]]>
$Index=[GUID]
<Index>$Index</Index>
</Information>
$ResourcesSection='';$StructureList=''
Objects
Structure
<Resources>$ResourcesSection</Resources>
</Application>
Problems
Properties
Output
Variables
Node value

Name	Value
$Rep	No
$Description	VDOM XML File Data ....
$Index	E598C900-44E9-0744-E121-77B918ACF
this	SampleApplication

```
[186305A7-EE87-CBD7-62E8-7748EA9A142B 0 0 null]
```

```
<?xml version="1.0" ?> <Application> <Information> <Active>1</Active>
<Description><![CDATA[VDOM XML File Data ....]]> <Index>E598C900-44E9-0744-E121-
77B918ACF1DE</Index>
```

```
Node (Normal): <Index>E598C900-44E9-0744-E121-77B918ACF1DE</Index>
Node (Function Command): GUID => E598C900-44E9-0744-E121-77B918ACF1DE
Node (Normal): <Description><![CDATA[VDOM XML File Data ....]]>
Node (Operation Command): $Description='VDOM XML File Data ....'
Node (Test Command): $Rep=='Yes' => false
Node (Function Command): question => No
```

Figure 11: PowerPack builder debug mode window.

In debug mode there is 3 windows used, like it's shown on figure 11, one **Output** witch show the current graph output, the last text generated is colored in **Red.** A **Node Value** witch show an historic track of each node cross with this information:

Node(Type):Output value of the node
Type can be:

- Text -> for simple text node, the out is the text.
- subGraph :
  - If not enter in sub graph the return is output generated by this graph
  - If enter in the sub Graph the return is Jump subGraph(SubGraphName)
- Command:
  - If it's affectation the output value will be $variableName='Value' or Value depend if it's numerical or string
  - If it's a test True or False
  - If it's a function the value returned by the function.

The last window show all the variables already evaluated during the process.

During the debugging process, the Power Pack Graph Design show the progression, a blue rectangle is lighting on the node which going to be evaluated and a highlight arrow is shown to indicate which one was selected to reach this node.

The Key commands are this:

Crt+R:	To run normally the Graph without Debug option.
Crt+D :	To enter in debug mode and stop to the first Break point.
Crt+B:	To enter in Step by Step mode.
Crt+M:	To resume to normal process.
Crt+I:	To go to the next step and enter in sub graph in the case of encounter one.
Crt+O:	To go to the next step but not enter in sub graph
Crt+C:	To break the normal process of graph.