Tutorial on text transformation with pure::variants

Table of Contents

1. Overview

This tutorial shows how to perform text transformations using the source elements *ps:fragment*, *ps:condtext*, and *ps:condxml*.

The usage of these source elements will be demonstrated on a **pure::variants** project used to configure a simple C++ application calculating the factorial of a given number. The application prints intermediate calculation results in different formats depending how it is configured. When the application is started the version and build date are shown. Additionally the example project shall include documentation and application build files.

The following steps have to be performed for this task.

1. A new pure::variants project has to be created.
2. The application files have to be written and included in the project.
3. The configurable features of the application have to be mapped to a feature model.
4. The components of the application have to be mapped to a family model.
5. Finally a transformation has to be configured to transform the variation points of the application.

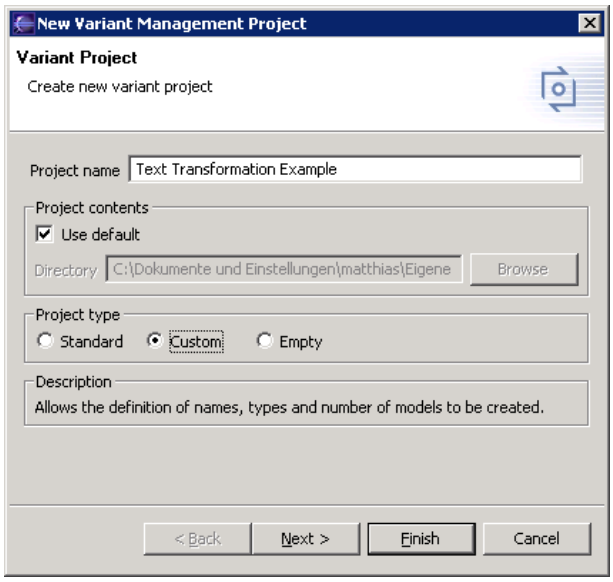
2. About this tutorial

The reader of this tutorial is expected to have basic knowledge of **pure::variants** and how the **pure::variants Standard Transformation** works. Please consult the pure::variants introductory material before reading this tutorial. This tutorial is available in online help or in printable PDF format here.

3. Setting up the pure::variants project

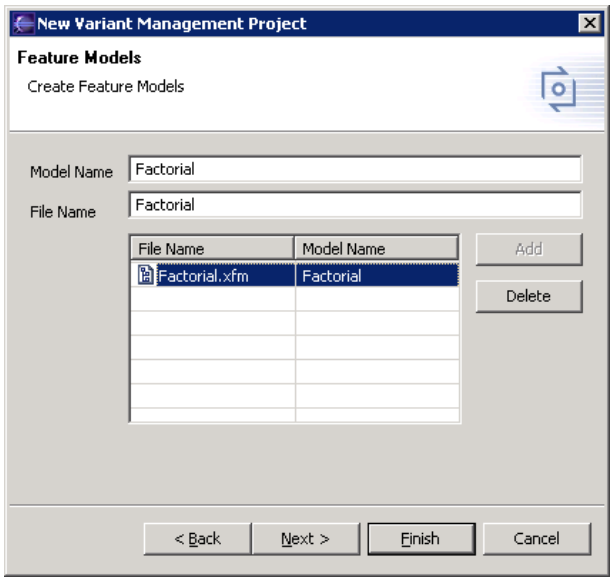
The first step is to create the **pure::variants** project. Switch to the *Variant Management* perspective and choose *New -> Variant Project* from the context menu of the *Variant Projects* view. Enter "Text Transformation Example" as project name, choose *Custom* project type, and press *Next* two times.

Figure 1. The new project wizard



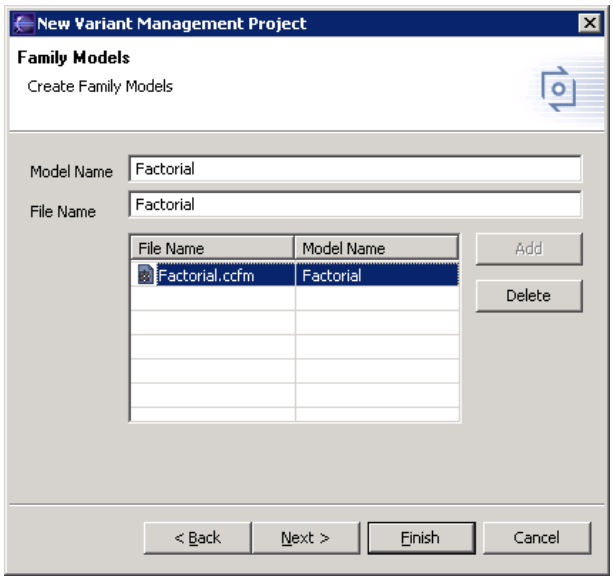
On the *Feature Models* page enter "Factorial" as model name and click *Add*. This adds a new feature model named **Factorial** to the project.

Figure 2. Adding a feature model



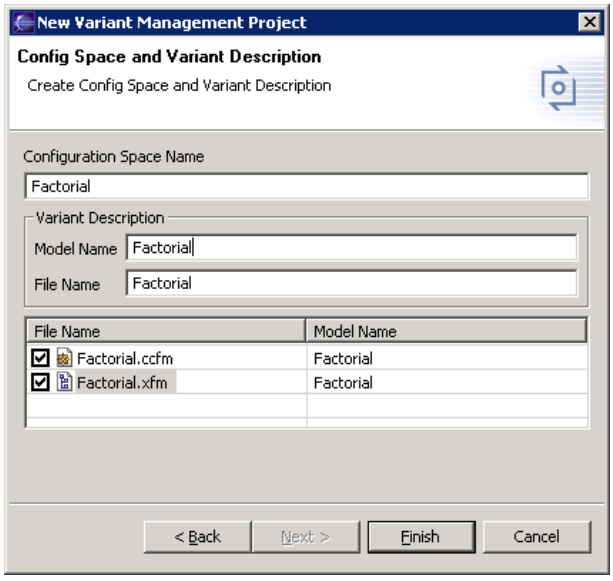
Click *Next* and also enter "Factorial" as the family model name and click *Add*. This adds a new family model named **Factorial** to the project.

Figure 3. Adding a family model



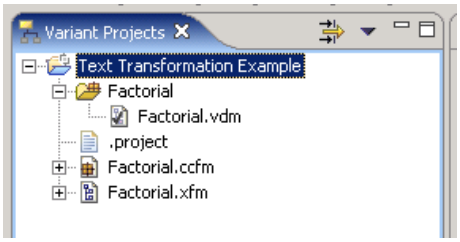
Click *Next* again. On the *Config Space* page enter "Factorial" for both the configuration space name and the variant description model name. Ensure that `Factorial.xfm` and `Factorial.ccfm` are selected. This adds a configuration space and a variant description model to the project.

Figure 4. Adding the configuration space and variant description model



After clicking *Finish*, the basic project structure including the models and the configuration space are created.

Figure 5. The created project structure



The next step is to write the application files. These should be located in their own folder inside the project containing source, build, and documentation files. This folder is created by selecting the project and clicking the right mouse button. From the context menu select the *New -> Folder* menu item. Enter "Source" as folder name and click *Finish*.

3.1. Source Files

The source files of the application are the main application file `fact.cc` and the header file `Factorial.h`. Create a file `fact.cc` in the **Source** folder with the following content:

```
#include "Factorial.h"
#include <iostream>
#include <stdlib.h>

int main(int argc, char** argv) {
  info();
  if (argc > 1) {
    int x = atoi(argv[1]);
    std::cout << x << "! = " << Factorial(x) << std::endl;
  }
  return 0;
}
```

The **main** function first prints a short application info by calling **info()**, and then calculates and prints the factorial of a given number. The function **info** will be generated during the transformation using the *ps:fragment* source element. It will contain code for printing the name, version, and build time of the application.

In the **main** function, the class **Factorial** is called to calculate the factorial of the given number. This class is defined in the file `Factorial.h`. Create a new file with the following content and name it "Factorial.h":

```
// PV:IFCOND(pv:hasFeature('IntermediateResults'))
#include <iostream>
// PV:ENDCOND

class Factorial {
  int result;
public:
  Factorial(int x) {
    result = factorialOf(x);
  }
  operator int() {
    return result;
  }
private:
  int factorialOf(int x) {
    if (x <= 1) {
      result = 1;
    } else {
      result = x * factorialOf(x-1);
    }
    // PV:IFCOND(pv:hasFeature('IntermediateResults'))
    //   PV:IFCOND(pv:getAttributeValue('IntermediateResults','ps:feature',
'Format')='number')
    std::cout << result << std::endl;
    //   PV:ELSEIFCOND(pv:getAttributeValue('IntermediateResults','ps:feature',
```

```
'Format')='sentence')
    std::cout << "Factorial of " << x << " is " << result << std::endl;
    //   PV:ELSECOND
    std::cout << x << "! = " << result << std::endl;
    //   PV:ENDCOND
    // PV:ENDCOND
    return result;
  }
};
```

This class implements the factorial calculation algorithm. Code for printing intermediate calculation results is activated or disabled depending on the selection of a feature **IntermediateResults**. This is realized by enclosing the corresponding code in special conditional statements supported by the *ps:condtext* source element. The conditions are ordinary **XPath** expressions and thus can contain comparisons and calculations. The intermediate results are printed in different formats (as sentence, as formula, or as single number) depending on the value of the **Format** attribute of feature **IntermediateResults**.

3.2. Documentation Files

The documentation of the application will be a single **XHTML** file. Create a file named "Docu.xhtml" in the **Source** folder with the following content:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<title>Factorial Calculation Program</title>
</head>
<body>

<h1>Factorial Calculation Program</h1>

<p></p>

<h2>Usage</h2>

<p>The only argument of the program is a number for which the factorial is
calculated.</p>

<h2>Result</h2>

<p>The result of invoking the program is a formula like:</p>
<p>3! = 6</p>

<h2 condition="pv:hasFeature('IntermediateResults')">
Intermediate Results
</h2>

<p condition="pv:hasFeature('IntermediateResults')">
  <p condition="pv:getAttributeValue('IntermediateResults','ps:feature',
'Format')='number'">
  Intermediate results are shown as simple numbers.
  </p>
  <p condition="pv:getAttributeValue('IntermediateResults','ps:feature',
'Format')='formula'">
  Intermediate results are shown as formulas, e.g. 3! = 6.
  </p>
  <p condition="pv:getAttributeValue('IntermediateResults','ps:feature',
'Format')='sentence'">
  Intermediate results are shown as sentences, e.g. Factorial of 3 is 6.
  </p>
</p>

</body>
</html>
```

Just as for file `Factorial.h` the content of this file varies depending on the selection of feature **IntermediateResults** and the value of its **Format** attribute. This is achieved by placing conditions as special attributes on tags in the **XHTML** document. These attributes are evaluated by the *ps:condxml* source element during the transformation to decide which tags become part of the transformed document.

3.3. Build Files

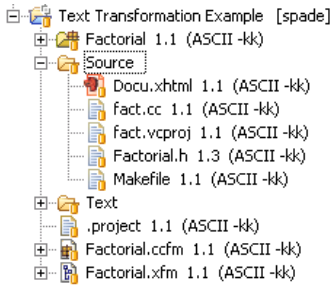
The application will have two build files. The first is a Visual Studio project file to build the application on Windows. Start Visual Studio and create an **empty** *Visual C++ Win32 Console Project* with name "fact" in the **Source** folder of the project. Add the source files `fact.cc` and `Factorial.h` to the Visual Studio project.

The second build file is a GNU **make** file to build the application on UNIX-like platforms. Create a new file in the **Source** folder with name "Makefile" and the following content:

```
fact: fact.cc Factorial.h
	$(CXX) -o fact fact.cc -I.
```

The final project structure should look as follows.

Figure 6. The final project structure



4. Setting up the feature model

The next step is to create the feature model for the application including its configurable features.

Open the feature model `Factorial.xfm`. Right click on the root feature and select *New -> Generic Feature* from the context menu. The new feature wizard will be opened. Enter "Version" in the *Unique Name* field and "Program Version" in the *Visible Name* field. Add a *non-fixed* attribute "Version" with no value. This attribute will be used to configure the version number of the application. After clicking the *OK* button the new feature is created.

Create an optional feature below the root feature with the unique name "IntermediateResults" and the visible name "Show Intermediate Results". Create three alternative features below the feature **Show Intermediate Results**. The first feature has the visible name "Show intermediate results as simple numbers" and the unique name "Number". The second feature has the visible name "Show a formula for each intermediate result" and the unique name "Formula". The third feature has the visible name "Show a sentence for each intermediate result" and the unique name "Sentence".

Add an attribute with name "Format" and three values, "number", "formula", and "sentence", to the feature **Show Intermediate Results**. Add the following restriction to value **number** by selecting *New-> Restriction* from the context menu of the value.

```
hasFeature('Number')
```

This means that the value of attribute **Format** shall be "number" if the feature **Number** is selected. For the value **formula** add the following restriction.

```
hasFeature('Formula')
```

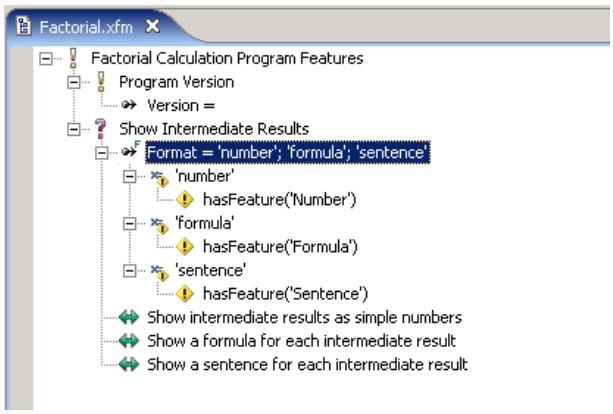
Finally add the following restriction to the value **sentence**.

```
hasFeature('Sentence')
```

These features and the attribute **Format** will be used to configure whether the application shall print intermediate results, and in which format.

After that the feature model should look as follows.

Figure 7. The Factorial feature model



5. Setting up the family model

After mapping the features of the application it is now time to map its components to a family model. The family model doesn't just define which files belong to the application by choosing corresponding family model elements for the files it can also be used to define transformations of the application files This is where the text transformation elements *ps:condtext*, *ps:condxml*, and *ps:fragment* come into play.

The application components can be divided into three groups: **Build Files**, **Source Files**, and **Documentation Files corresponding to the different file types in the Source directory.**

5.1. Build Files

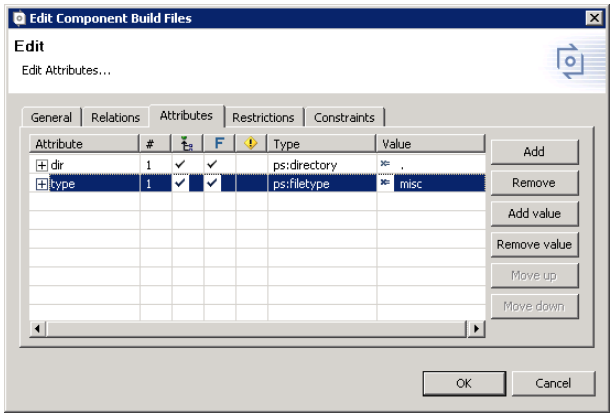
We'll deal with the build files first. These will not be transformed during the transformation/configuration of an application. Instead, they will simply be copied to the directory containing the final configured application files, i.e. the transformation output directory. So *ps:file* source elements should be used to represent the build files.

Open the family model `Factorial.ccfm`. Right click on the root element of the model and choose *New -> Component* from the context menu. Enter "Build Files" as the visible name and "Build" as the unique name, and click *Finish*. This will create a **Build Files** component that we will use to group all build files.

Since we'll use the **dir** and **type** attributes with the same value for several child elements, we'll create these attributes as *inheritable* attributes on the **Build Files** component. Right click on the **Build Files** component and select *New -> Attribute* in the context menu. Name the attribute "dir". Select *ps:directory* as attribute type and set the value to ".". Now select the *inheritable* option for the new attribute (click in the third column in the row of the attribute). This means that all children will inherit this value and need not define it. Add an-

other attribute with name "type" and type *ps:filetype*. Set the value of the attribute to "misc" and also make this attribute inheritable.

Figure 8. The inheritable attributes for the child elements

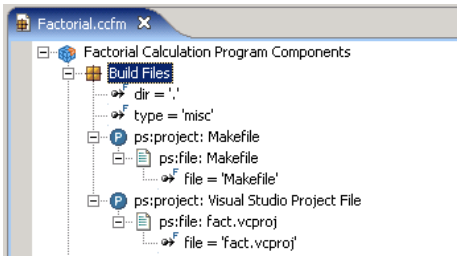


There are two build files, `fact.vcproj` and `Makefile`. Right click on the component **Build Files** and select *New -> Project* in the context menu. Enter "Makefile" as the unique and visible names and click *Finish*. Right click on the element **Makefile** and select *New -> File* in the context menu. The *New File* wizard is opened. Enter "Makefile" as the value for the **file** attribute. Select the *inherit* check box of the attributes **dir** and **type** (if available). The new element representing the **make** file will be created after clicking *Finish*.

The model elements for the second build file, the Visual Studio project file, are created in the same way as for the first build file. Create a new *Project* element with the visible name "Visual Studio Project File" and the unique name "VSProject". Create a new *File* element below the project file element with "fact.vcproj" as the value for the **file** attribute and inherit the **dir** and **type** attributes.

The family model structure should now look as follows.

Figure 9. The family model with the Build Files component



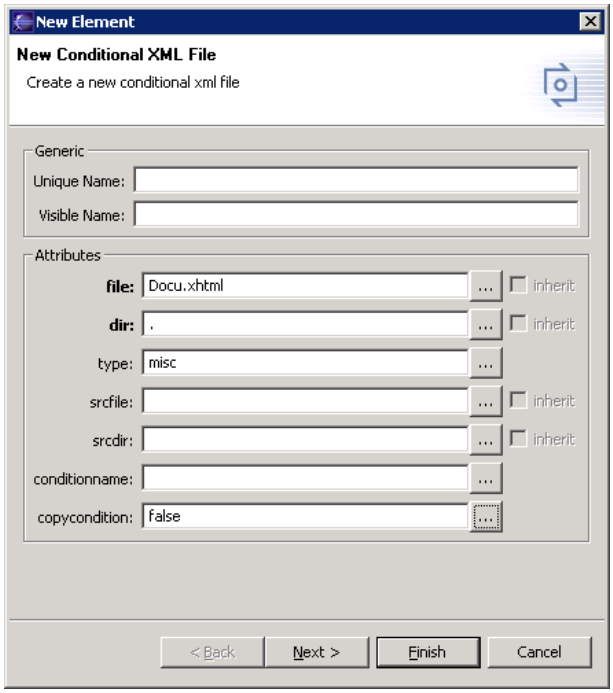
5.2. Documentation Files

The next step is to map the documentation file `Docu.xhtml` to the family model. In constrast to the build files this file will be transformed before it is placed in the transformation output directory. The result of the transformation will be that the transformed documentation file will only contain text corresponding to the selected features. To support this, the XHTML document has special attributes with conditions on its tags, as shown above. These conditions will be evaluated by the *ps:condxml* source element. If a condition evaluates to false, then the corresponding tag is removed from the document.

Start by adding a new *Component* below the root element of the family model with "Documentation Files" as the visible name and "Documentation" as the unique name. This creates a **Documentation Files** component that will be used to group all documentation files. Create a new *Generic Element* with the visible name "Docu.xhtml" and type "xhtml" below the component **Documentation Files**.

Below the element **Documentation Files** add a new *Conditional XML* element. In the wizard that is opened enter "Docu.xhtml" as the value for attribute **file**, "." as the value for attribute **dir**, "misc" as the value for attribute **type**, and "false" as the value for attribute **copycondition**. Setting the value of attribute **copycondition** to "false" means that the special attributes on the tags of the XHTML document containing the conditions will be removed after a condition is evaluated. This is important here because these attributes are not valid XHTML.

Figure 10. The New Conditional XML File Wizard



For more information about the *ps:condxml* source element please read section "Standard transformation for source elements" of the pure::variants User's Guide.

5.3. Source Files

Finally the source files, `fact.cc` and `Factorial.h`, have to be mapped to the family model. We need to transform both files.

`fact.cc` contains a call to the function **info**. This function has to be generated during the transformation to print the name, the configurable version, and the build time of the application. The function **info** will be generated using the source element *ps:fragment*.

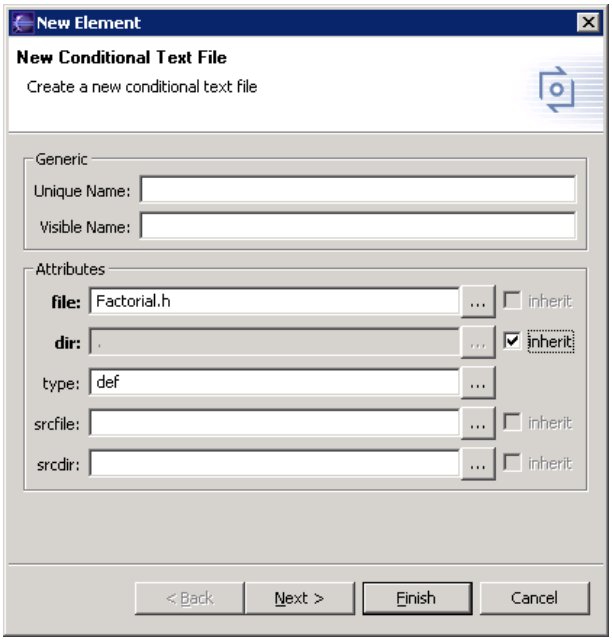
`Factorial.h` contains special statements with conditions as guards for the code to print the intermediate results. These statements will be evaluated using the source element *ps:condtext*.

Create a new *Component* with the visible name "Source Files" and the unique name "Sources". Add a new inheritable attribute "dir" with type *ps:directory* and value "." to the component.

Now the header file of class **Factorial** has to be added. Create a new *Class* element below the component **Source Files**. In the opened *New Class* wizard enter "Factorial" as the unique and visible name. Enable the *inherit* button for the attribute **dir** and click *Finish*.

Create a new *Conditional Text* source element below the **Factorial** element. In the opened *New Conditional Text* wizard set the value of the attribute **file** to "Factorial.h", the value of the attribute **type** to "def", and enable the *inherit* button for the attribute **dir**.

Figure 11. The New Conditional Text File Wizard



For more information about the *ps:condtext* source element read the section "Standard transformation for source elements" of the pure::variants User's Guide.

For the second source file, `fact.cc`, create a new *Generic Element* below component **Source Files** with type "main" and visible name "fact.cc". Add the new *inheritable* attributes **type** with value "impl" and **file** with value "fact.cc" to the element **main**. These attributes are required for the *ps:fragment* elements that we will create next.

Create a new *Fragment* element below the **main** element. In the opened wizard enter "Insert info() function" as the visible name and select the *inherit* boxes for the attributes **file**, **dir**, and **type**. Enter the following text into the *content* input field and click *Finish*.

```
#include <iostream>

void info() {
  std::cout << "Factorial Calculation,
```

This is the first fragment of the function **info**. It will be written written to the file `fact.cc` in the transformation output directory. The next fragment appends the version number and the current time to the file `fact.cc`.

Create a second *Fragment* element below the **main** element and name it "Add version and

time to info". Select the *inherit* boxes for the attributes **file**, **dir**, and **type**. Enter the following text into the *content* input field and click *Finish*.

```
getAttribute('Version','Version',Version) and
get_time(CurrentTime) and
convert_time(CurrentTime,Time) and
sformat(Value,'Version ~w, Build ~w',[Version,Time])
```

The value of the **content** attribute is not simple text but a calculation. Open the *Properties* dialog of the fragment element by double-clicking on it. Switch to the *Attributes* page and change the value of attribute **content** into a calculation by clicking into the *Value* field of the attribute, then on the button "..." and on *Calculation* in the opened dialog.

The calculation starts by getting the value of attribute **Version** of element **Version** and saves this value into the variable **Version**. Then it gets the current time and saves it in the variable **CurrentTime**. The variable **Time** then is filled with a human readable time string corresponding to the current time. Finally the result of the calculation, saved in variable **Value**, is a string where the version and the current time are inserted. This string will be appended to the file `fact.cc` during the transformation.

The next fragment contains the rest of function **info**. Create a third fragment below element **main** with the visible name "Finish info() function" and inherited attributes **file**, **dir**, and **type**. Enter the following text as value of attribute **content**.

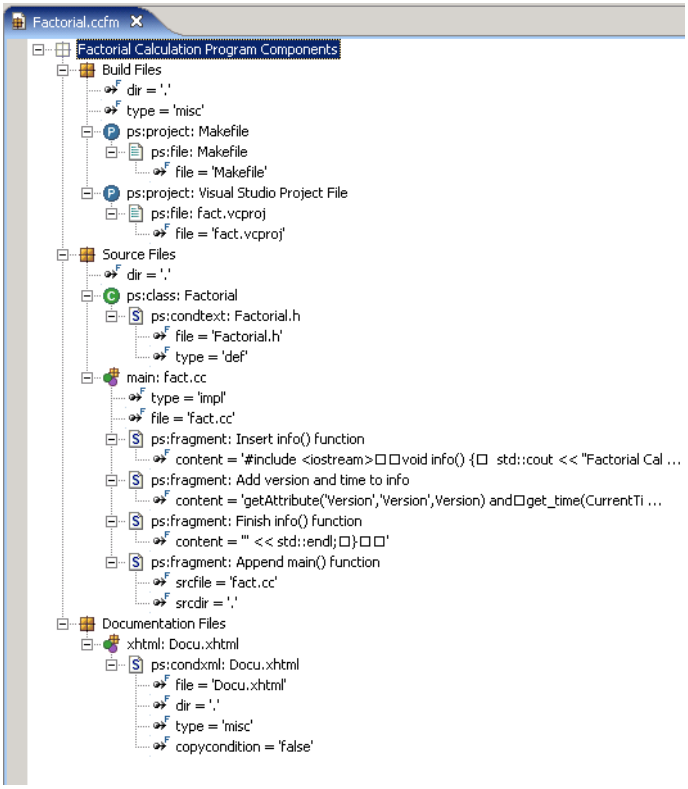
```
" << std::endl;
}
```

Now the function **info** is complete. The last fragment appends the original content of file `fact.cc`, i.e. the **main** function, to the file `fact.cc` in the transformation output directory.

Create a new Fragment element below element main with the visible name "Append main() function" and inherited **file**, **dir**, and **type** attributes. In contrast to the other fragments the fragment text does not come from the attribute **content** but from a file. Click on the button *Take fragment from file* in the *New Fragment* wizard. Enter "fact.cc" as value of attribute **srcfile** and "." as value of attribute **srcdir** and press *Finish*.

The final family model structure should look as follows.

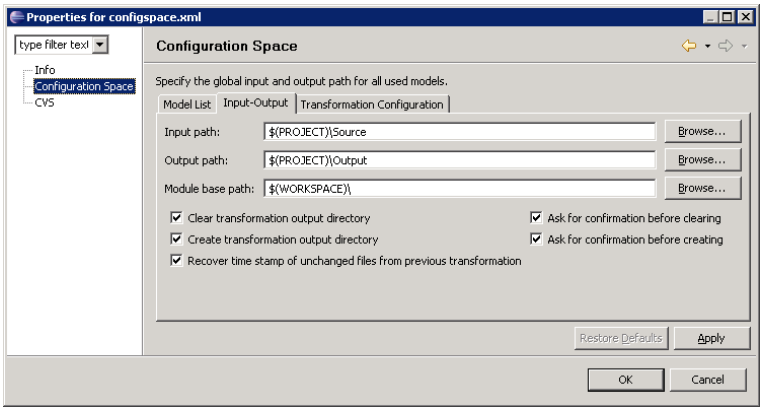
Figure 12. The finished family model Factorial.ccfm



6. Setting up the transformation

Some configuration options have to be changed to support the transformation. Switch to the *Variant Projects* view and select the **Text Transformation Example** configuration space. Open the *Properties* dialog from the context menu of the configuration space and switch to the *Configuration Space* page. On the *Input-Output* tab of the dialog enter "$(PROJECT)\Source" as the input directory and "$(PROJECT)\Output" as the output directory for the transformation. Select the "Clear transformation output directory" and "Create transformation output directory" boxes.

Figure 13. The input and output paths configuration



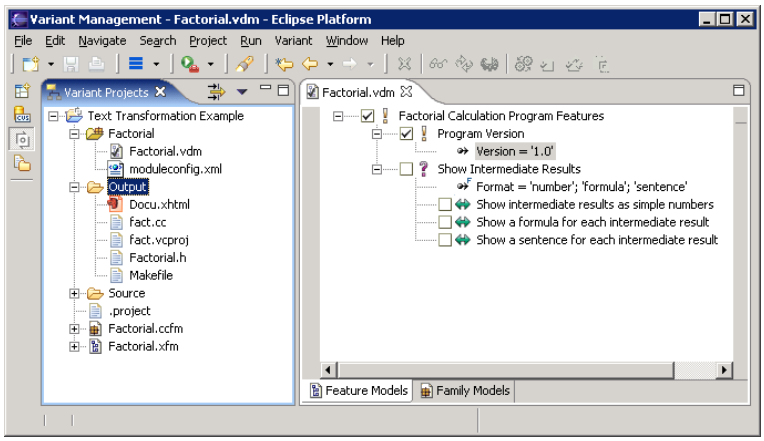
Switch to the *Transformation Configuration* tab and click *Add*. In the opened dialog select the **standard transformation** module, enter "Create action list" as the name, and click *Finish*. This will add a new *pure::variants Standard Transformation* module to the transformation module configuration. This module creates a list of transformation actions according to the elements of the family model executed by the next module to add. Click on *Add* again and select the **actionlist** module, enter "Execute action list" as name, and press *Finish*. This is all there is to setting up the transformation.

7. Generating a variant

Now the project and application files are prepared to start a first transformation. Open the variant model by double clicking on the file `Text Transformation Example.vdm` in the configuration space folder. Enter a value for the attribute **Version** of feature **Program Version**, e.g. "1.0". This configures the version of the application to be "1.0".

To start the transformation, click the *Transform Model* button in the tool bar. After the transformation is finished refresh the project in the *Variant Projects* view by selecting the project and pressing **F5**. The new directory **Output** appears in the project containing the transformed files of the application.

Figure 14. Contents of the Output directory



The file `fact.cc` in the **Output** directory should now contain the following code (except for a different time string).

```
#include <iostream>

void info() {
  std::cout << "Factorial Calculation, Version 1.0, Build Tue May 22 10:00:58 2006" <<
std::endl;
}

#include "Factorial.h"
#include <iostream>
#include <stdlib.h>

int main(int argc, char** argv) {
  info();
  if (argc > 1) {
    int x = atoi(argv[1]);
    std::cout << x << "! = " << Factorial(x) << std::endl;
  }
  return 0;
}
```

Code for the class **Factorial** and the documentation file have changed correspondingly after the transformation.

```
//

class Factorial {
  int result;
public:
  Factorial(int x) {
    result = factorialOf(x);
  }
  operator int() {
    return result;
  }
private:
  int factorialOf(int x) {
    if (x <= 1) {
      result = 1;
    } else {
      result = x * factorialOf(x-1);
    }
    //
    return result;
  }
};
```

Now select the features **Show Intermediate Results** and **Show a sentence for each intermediate result** in the variant model editor. Press the *Transform Model* button and refresh the project as described above. Now the code for the class **Factorial** looks like this.

```
//
#include <iostream>
//

class Factorial {
  int result;
public:
  Factorial(int x) {
    result = factorialOf(x);
  }
  operator int() {
    return result;
  }
private:
  int factorialOf(int x) {
    if (x <= 1) {
      result = 1;
    } else {
      result = x * factorialOf(x-1);
    }
    //
    //
    std::cout << "Factorial of " << x << " is " << result << std::endl;
    //
    //
    return result;
  }
};
```