

< pure::variants テキスト変換について >

1. 概要

このチュートリアルでは、`pure::variants` でバリエーションごとのテキストやドキュメントを生成させる方法を紹介합니다。英語、日本語、XML などあらゆるフォーマットをサポートし、仕様書や、make ファイル、ソースコードをバリエーションに定義されたフィーチャに従って生成させる仕組みについて。これにはソースエレメント (`ps:fragment`, `ps:condtext`, `ps:condxml`) を使用します。これらはファミリーモデルのバリエーションとなる部品に設定されて、バリエーションを定義したモデルをトランスフォーメーション (変換) する時に、選択されたフィーチャに対応したテキストやドキュメントファイルなどを生成するために使用されます。

`ps:fragment` は、テキストの一部分や、個々のファイルなどを部品化するための仕組みです。

`ps:condtext` は、フィーチャの選択に応じて、テキスト形式の部品化された箇所の有効・無効の条件を指定する仕組みです。この処理では、`ps:condtext` がサポートする特別な条件命令 (マクロ) をテキストファイルにコメントとして挿入することで、テキストの一部分の出し入れをコントロールすることができます。仕様書や、ビルド時に読み込まれるコンフィグ用make、ソースコード上で特定処理の出し入れなど。

`ps:condxml` は、フィーチャの選択に応じて、XML形式の部品化された箇所の有効・無効の条件を指定する仕組みです。この動作は、`ps:condxml` がサポートする特別な属性をXMLファイル内のタグに設定することで実現されます。

以下 `pure::variants` のサンプルプロジェクト“与えられた数値の階乗を求める単純なC++アプリケーション”を用いて、これらソースエレメントの用法について説明します。このアプリケーションは、プログラムの構成内容によって、計算の中間結果を様々なフォーマットで出力します。アプリケーションが実行されると、バージョンとソースファイルの生成日時が表示されます。また、このサンプルプロジェクトには、ドキュメント生成やアプリケーションを構成するためのファイルも含まれます。

このチュートリアルでは、一からプロジェクトを作ることから紹介していますが、結果のプロジェクトは弊社サイトで公開しています。 <http://www.fuji-setsu.co.jp/products/purevariants/tutorials.html>

以下の手順で、このアプリケーションを作成します。

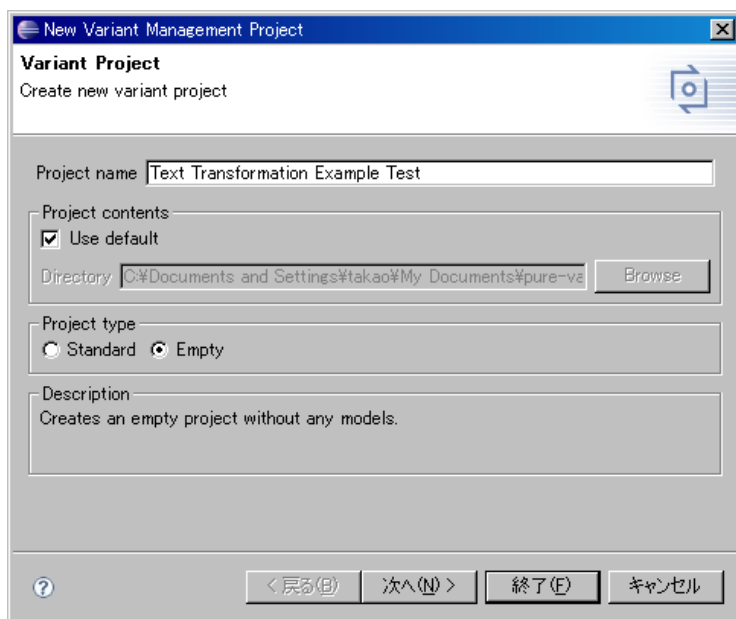
1. 新規に、`pure::variants` プロジェクトを作成する。
2. アプリケーションファイル (ソースやビルド用ファイル) を作成し、プロジェクトに含める。
3. 構成されるアプリケーションの機能を、フィーチャモデルにマップする。
4. アプリケーションの部品を、ファミリーモデルにマップする。
5. 最後に、アプリケーションのバリエーションポイントを変換するために、トランスフォーメーション (変換) の設定を行う。

2. このチュートリアルに関して

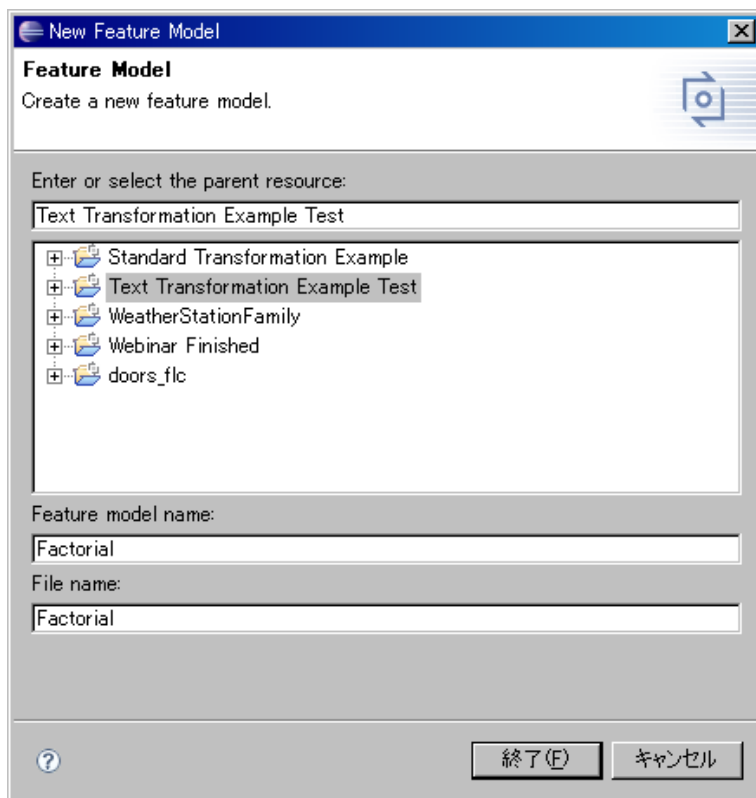
このチュートリアルを読む上で、pure::variantsの基礎知識と、pure::variantsのスタンダードトランスフォーメーションの動作原理を理解する必要があります。このチュートリアルを読む前に、pure::variants導入資料を参照してください。このチュートリアルは、弊社サイトから参照できるようになっています。 <http://www.fuji-setsu.co.jp/products/purevariants/index.html>

3. pure::variants プロジェクトの設定

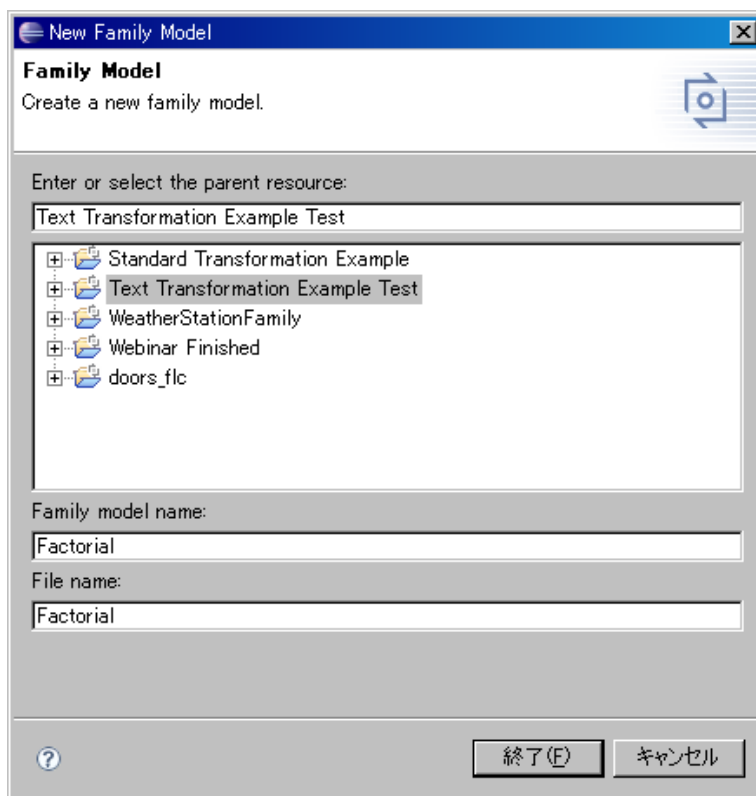
最初の手順は、pure::variants プロジェクトの作成です。Variant Management パースペクティブに切り替え（pure::variants を起動すれば、Variant Management パースペクティブになっています）Variant Project ビューのコンテキストメニューから、New -> Variant Project を選択します。Project name に “Text Transformation Example Test” を入力し、Project type で Empty を選択して Next ボタン、次に終了ボタンを押します。Variant Project ビューにプロジェクト “Text Transformation Example Test” が追加されます。



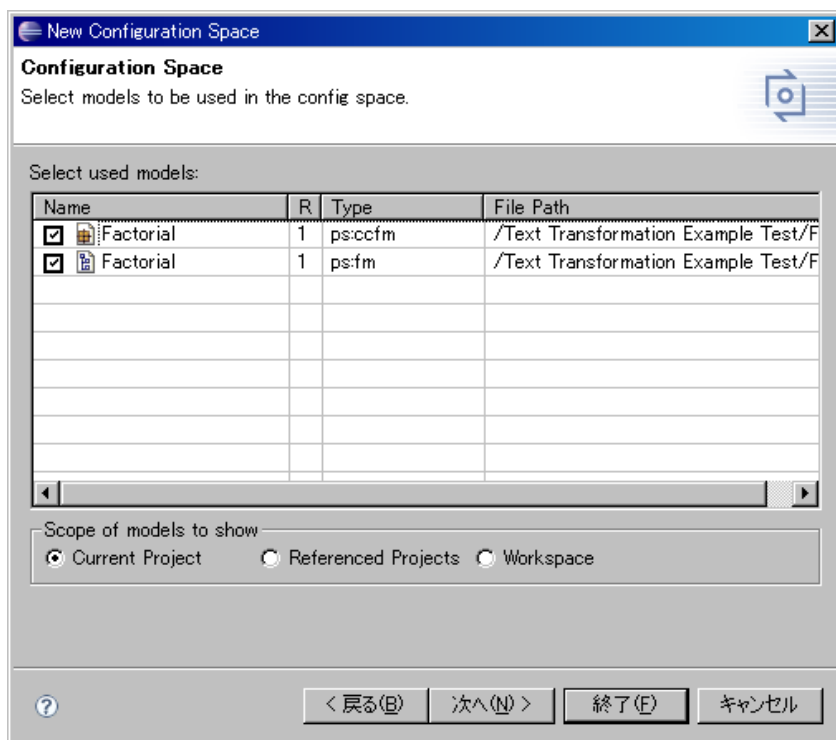
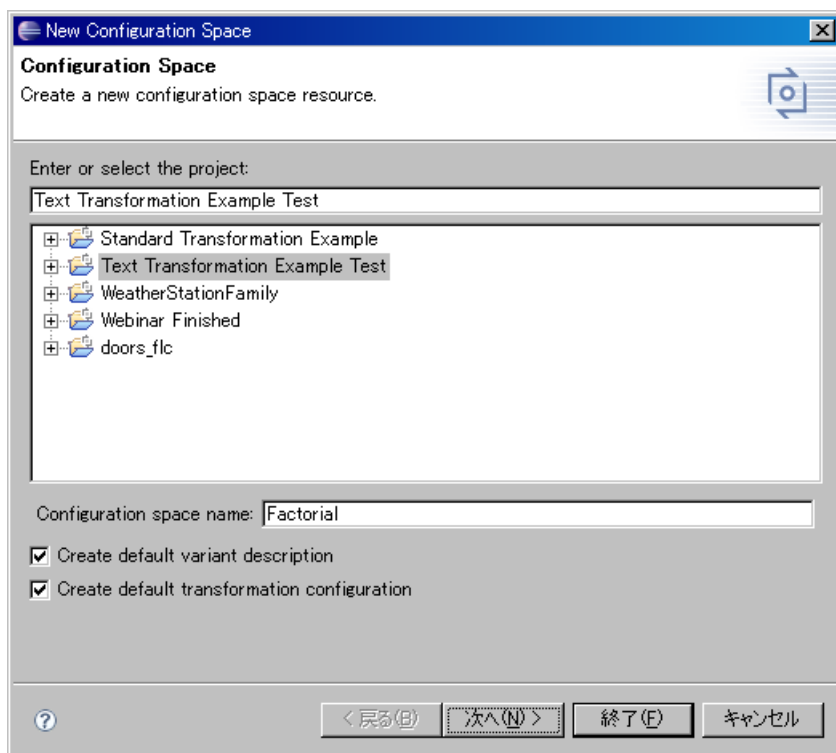
次に、Variant Project ビューの “Text Transformation Example Test” 上で右クリックし、New -> Feature Model を選択します。以下、Feature model name に “Factorial” と入力し、終了ボタンを押します。



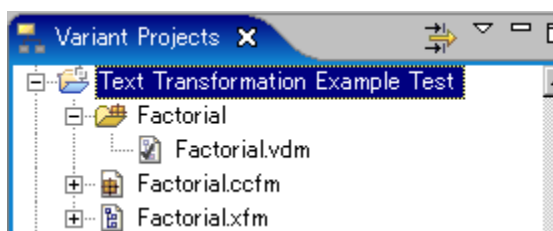
次に、Variant Project ビューの “Text Transformation Example Test” 上で右クリックし、New -> Family Model を選択します。以下、Family model name に “Factorial” と入力し、終了ボタンを押します。



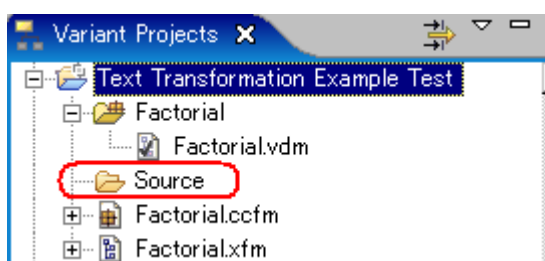
次に、Variant Project ビューの “Text Transformation Example Test” 上で右クリックし、New -> Configuration Space を選択します。以下、Configuration space name に “Factorial” と入力し、次へボタンを押し、終了ボタンを押します。



これで、フィーチャモデル、ファミリーモデル、コンフィグレーションスペースを含んだプロジェクト構造が作成されました。



次の手順では、アプリケーションファイル（ソースファイル、ビルドファイル、ドキュメントファイル）を作成します。これらは、プロジェクト内のフォルダに置きます。このフォルダは、プロジェクトを選択し、右クリックすることで作成できるようになっています。コンテキストメニューから New -> Folder を選択し、フォルダ名に “Source” と入力して終了ボタンを押します。以下のように “Source” フォルダが作成されます。



3.1. ソースファイル

この例では、ソースファイル “fact.cc”（メインのアプリケーションファイル）と “Factorial.h”（ヘッダファイル）を用います。Source フォルダの中に “fact.cc” を作成し、以下のコードを記述します。ファイルの作成方法は、Source フォルダを選択し、右クリックで New -> File を選択します。ファイル名に “fact.cc” と入力し、終了ボタンを押します。Variant Project ビューの “fact.cc” で右クリックし、Open with -> テキストエディター を選択して “fact.cc” を開き、コードを記述します。

```
[fact.cc]
#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;
}
```

```

#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;
}

```

main 関数は、最初に関数 info() を呼び出して、簡単なアプリケーション情報を出力し、その後、与えられた数値の階乗を計算して出力します。関数 info は、トランスフォーメーション実行時に、*ps:fragment* ソースエレメントを使って生成されます。この関数には、アプリケーションの名前、バージョン、ソースファイルの生成日時（トランスフォーメーション実行日時）を出力するためのコードが含まれます。

main 関数では、与えられた数値の階乗を計算するために、クラス “Factorial” が呼ばれます。このクラスは、“Factorial.h” 内で定義されます。“fact.cc” と同様に、以下の内容で、Source フォルダの中に “Factorial.h” ファイルを作成し保存しましょう。

[Factorial.h]

```

// PV:IFCOND(pv:hasFeature('IntermediateResults'))

#include <iostream>

// PV:ENDCOND

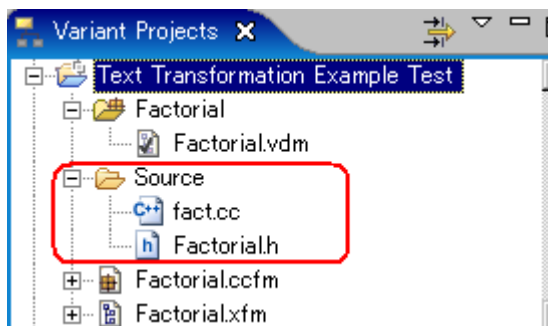
class Factorial {
    int result;
public:
    Factorial(int x) {
        result = factorialOf(x);
    }
    operator int() {
        return result;
    }
private:
    int factorialOf(int x) {
        if (x <= 1) {
            result = 1;
        } else {
            result = x * factorialOf(x-1);
  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

```



このクラスは、階乗計算のアルゴリズムを実装しています。計算の中間結果を出力する為のコードは、フィーチャ “IntermediateResults” の選択によって有効、無効が切り替えられます。

この動作は、*ps:condtext* ソースエレメントがサポートする特別な条件命令（マクロ）を、コードに挿入（コメントとして）することで実現されます。

“Factorial.h” ファイルに記述されている “PV:XXX” は、*pure::variants* 側で条件を見つけるための特別なマクロです。このヘッダファイルを Conditional Text (*ps:condtext*) エレメントとしてファミリーモデルに設定します。そうすることで、トランスフォーメーション実行時に、*ps:condtext* ソースエレメントによって、ヘッダファイル内のマクロが検知され、その条件に応じてファイルのどの部分が有効・無効であるかを判断し、ヘッダファイルを生成します(ファミリーモデルの設定に関しては、「5.3 ソースファイル」を参照)。“PV:XXX” に関して、以下に “Factorial.h” の一部を例として示します。

例：

```

// PV:IFCOND(pv:hasFeature('IntermediateResults')) // フィーチャ IntermediateResults が選択されている時、
#include <iostream> // この部分が結果ファイルに出力される
// PV:ENDCOND // マクロの終了
PV:IFCOND(条件)：与えられた条件が真なら、このマクロで囲まれた部分が結果ファイルに出力される

```

条件は、一般的な XPath (XML Path Language : XML 文書の特定の部分を指し示す構文を規定する)

で表現され、比較や計算を含めることもできるようになっています。中間結果は、フィーチャ “IntermediateResults” の属性 Format の値に応じて、様々な形式（文章、式、または数字）で出力されます。

3.2. ドキュメントファイル

この例では、ドキュメントとして、1つの XHTML (Extensible HTML : HTML を XML に適合するように定義し直したマークアップ言語)ファイルを使用します。ソースファイル同様、以下の内容で、Source フォルダの中に “Docu.xhtml” ファイルを作成し保存しましょう。

[Docu.xhtml]

```
<?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>
```

```
</p>
</body>
</html>
```

“ Factorial.h ”と同様に、このファイルの内容は、フィーチャ “ IntermediateResults ” の選択とその属性 Format の値に応じて変化します。これは、XHTML ドキュメント内のタグにある特別な属性を使い、条件を設定することで実現されます。これらの属性は、トランスフォーメーション中に *ps:condxml* ソースエレメントによって、どのタグの部分がドキュメントに出力されるか判断されます。

“ Docu.xhtml ” ファイルに記述されている “ condition ” は、*pure::variants* 側で条件を見つけるための特別な属性です。この XHTML ファイルを Conditional XML (*ps:condxml*) エレメントとしてファミリーモデルに設定します。そうすることで、トランスフォーメーション実行時に、*ps:condxml* ソースエレメントによって、XHTML ファイル内の属性 “ condition ” が検知され、その条件に応じてドキュメントファイルのどの部分が有効・無効であるかを判断し、ドキュメントファイルを生成します (ファミリーモデルの設定に関しては、「5.2 ドキュメントファイル」を参照)。condition に関して、以下に “ Docu.xhtml ” の一部を例として示します。

例：

```
<p condition="pv:hasFeature('IntermediateResults')"> // IntermediateResultsフィーチャが選択されている場合、
<p condition="pv:getAttributeValue('IntermediateResults','ps:feature','Format')='number'">
// IntermediateResultsフィーチャの属性Formatの値を取得し、
// その値が number の時、
Intermediate results are shown as simple numbers. // この部分がドキュメントとして抽出される
</p>
```

3.3. ビルドファイル

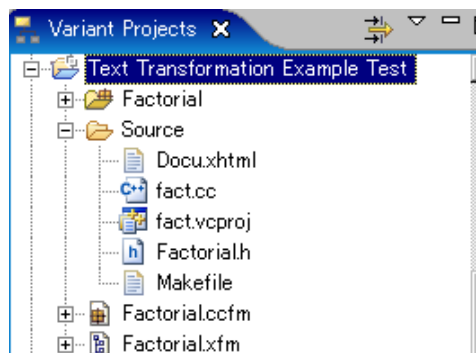
この例には、2 つのビルドファイルがあります。1 つ目は、Visual Studio のプロジェクトファイルで、Windows のアプリケーションとしてビルドします。Visual Studio を起動し、Source フォルダに “ fact ” という名前で空の Visual C++ Win32 Console Project を作成します。Visual Studio プロジェクトに、“ fact.cc ” と “ Factorial.h ” を追加します。(このチュートリアルでは、ビルドすることが目的ではないので、Visual Studio が無くても、以降の手順に問題はありませぬ。Visual Studio が無ければ、仮のプロジェクトファイルを作成しましょう。)

2 つ目のビルドファイルは、GNU の make ファイルで、UNIX 相当のプラットフォームで動作するアプリケーションを作成します。以下の内容で、プロジェクトの Source フォルダの中に、“ Makefile ” という名前のファイルを作成し保存しましょう。

[Makefile]

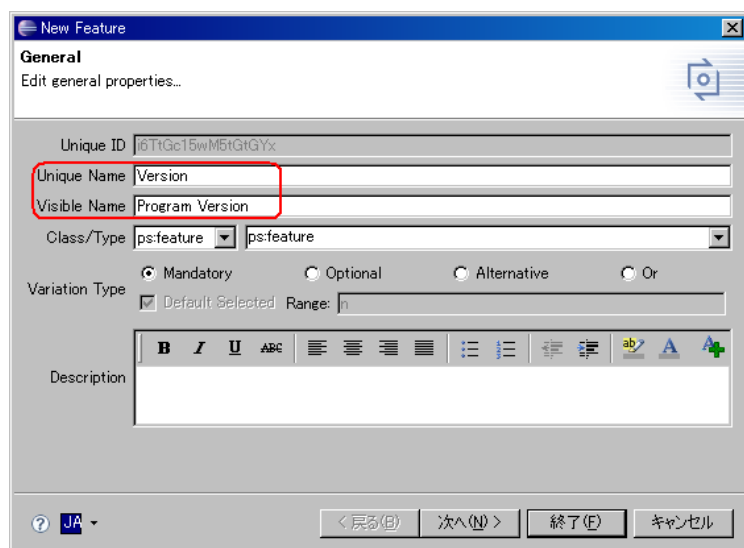
```
fact: fact.cc Factorial.h
$(CXX) -o fact fact.cc -I.
```

最終的に、pure::variants プロジェクトの構造は、下記のようになります。

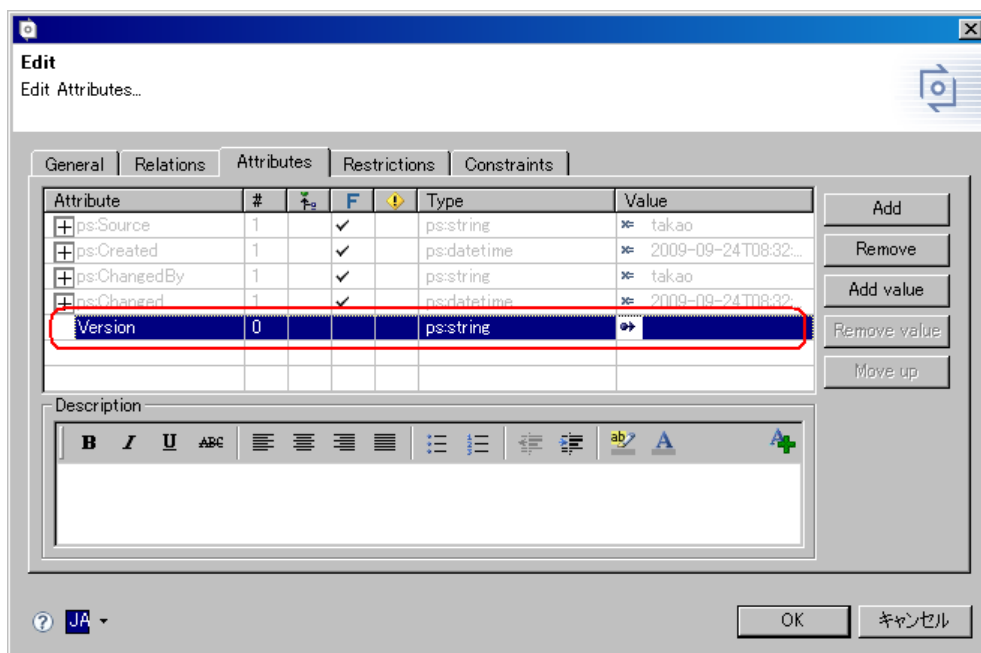


4. フィーチャモデルの設定

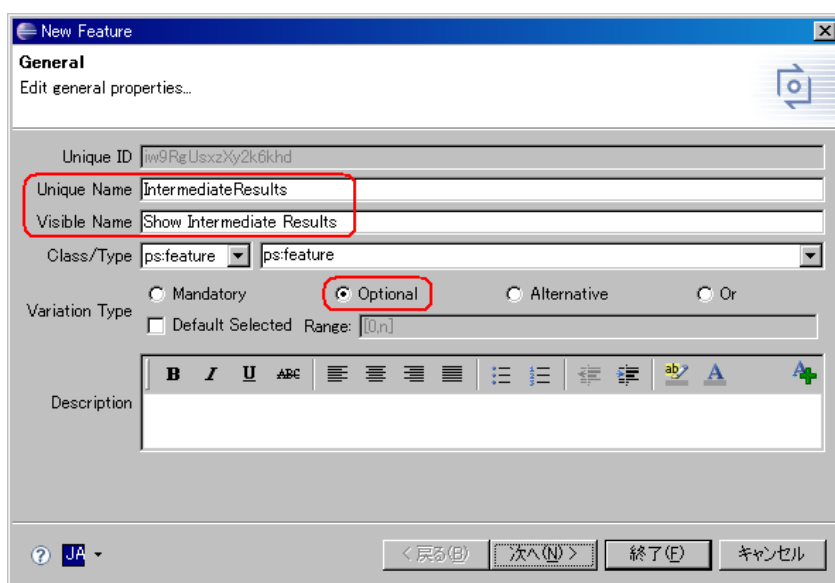
次の手順では、構成されるアプリケーションの機能を含む、フィーチャモデルを作成します。フィーチャモデル “Factorial.xfm” を開きます。ルートのフィーチャ (Factorial) で右クリックし、コンテキストメニューから New -> Feature を選択します。新規フィーチャウィザードが開くので、Unique Name フィールドに “Version”、Visible Name フィールドに “Program Version” を入力します。(Variation Type は、Mandatory を選択)



終了ボタンを押します。フィーチャ “Program Version” をダブルクリックし、ウィザードを開きます。Attributes タブを選択し、Add ボタンを押して以下のような値を持たない属性 “Version” を追加します (値は、トランスフォーメーション前に設定するので、4 列目 “F” のチェックを外しておきます)。この属性は、アプリケーションのバージョン番号を設定するために使用されます。OK ボタンをクリックします。



同様に、Unique Name が “ IntermediateResults ”、Visible Name が “ Show Intermediate Results ”、Variation Type が “ Optional ” であるフィーチャをルートフィーチャ配下に作成します。



終了ボタンを押します。

フィーチャ “ Show Intermediate Results ” の配下に、Variation Type が Alternative である 3 つのフィーチャを作成します。1 つ目は、Unique Name が “ Number ”、Visible Name が “ Show intermediate results as simple numbers ” であるフィーチャを作成します。

New Feature

General
Edit general properties...

Unique ID: fxz2VXRpffr0cd6mE

Unique Name: Number

Visible Name: Show intermediate results as simple numbers

Class/Type: ps:feature

Variation Type: Mandatory Optional Alternative Or

Description: [Empty text area]

<戻る(B) 次へ(N) > 終了(E) キャンセル

終了ボタンを押します。

2つ目は、Unique Name が “ Formula ”、Visible Name が “ Show a formula for each intermediate result ” であるフィーチャを作成します。

New Feature

General
Edit general properties...

Unique ID: lUfKyjwOjF3LCeHuS

Unique Name: Formula

Visible Name: Show a formula for each intermediate result

Class/Type: ps:feature

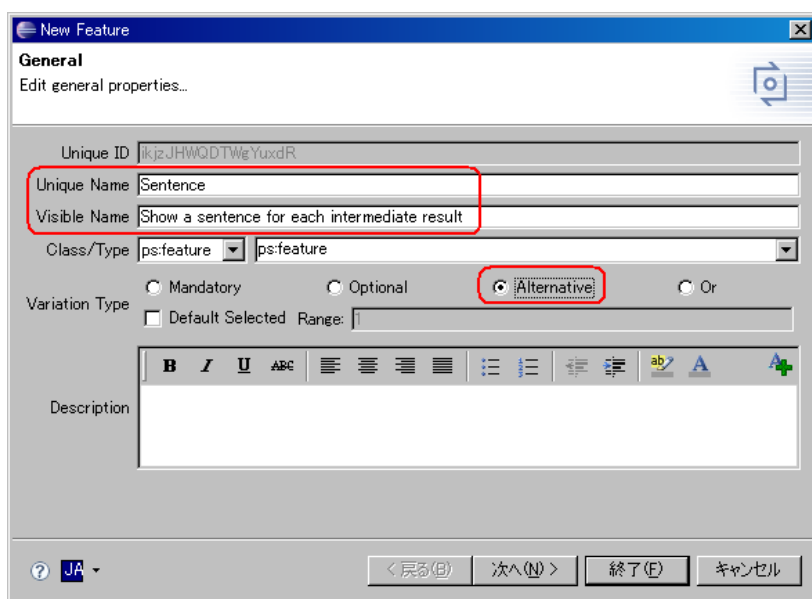
Variation Type: Mandatory Optional Alternative Or

Description: [Empty text area]

<戻る(B) 次へ(N) > 終了(E) キャンセル

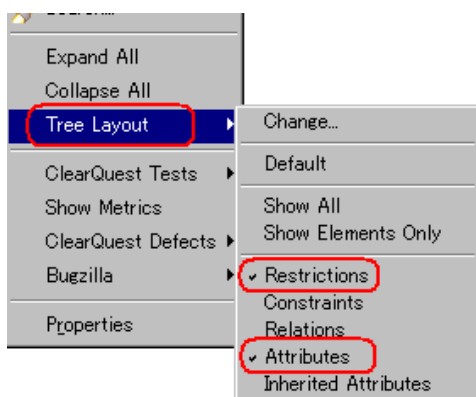
終了ボタンを押します。

3つ目は、Unique Name が “ Sentence ”、Visible Name が “ Show a sentence for each intermediate result ” であるフィーチャを作成します。

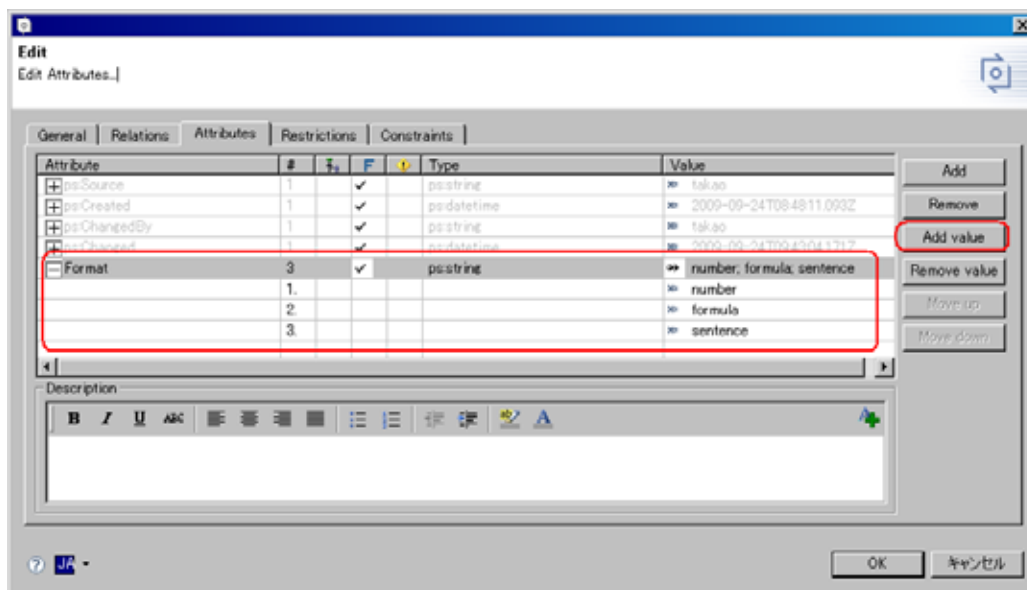


終了ボタンを押します。

フィーチャモデルの空白部分で右クリックし、コンテキストメニューから Tree Layout -> Restrictions と Attributes にチェックを入れ、フィーチャモデル上に Restriction と Attribute が表示されるように設定しましょう。

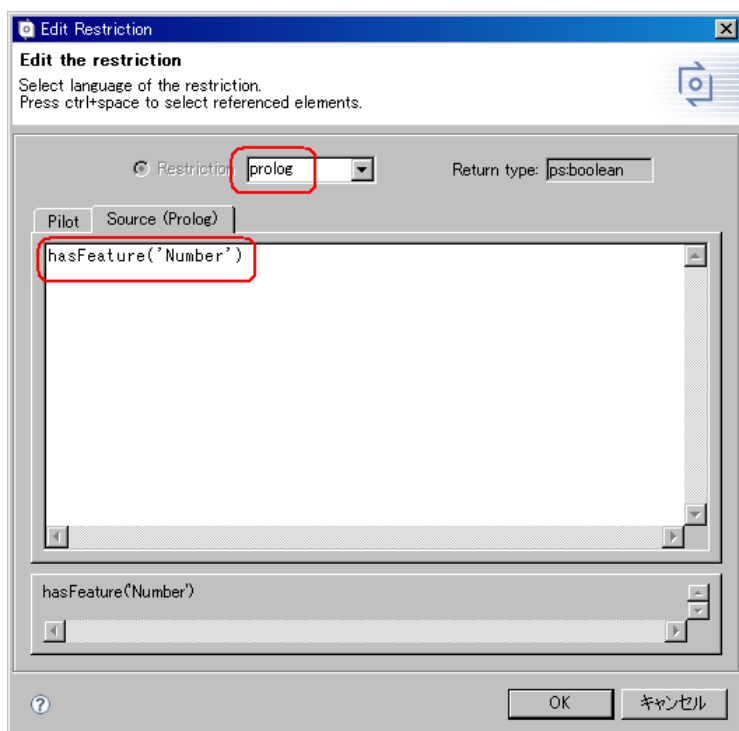
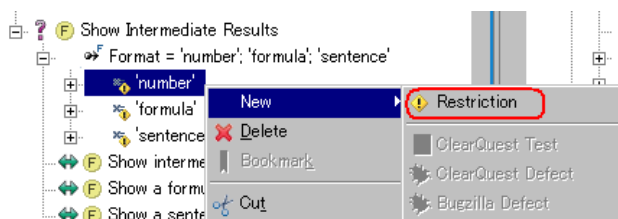


フィーチャ “ Show Intermediate Results ” に、Format という名前の属性を追加し、それに対して Add value ボタンを押して、3 つの値 “ number、formula、sentence ” を追加します。



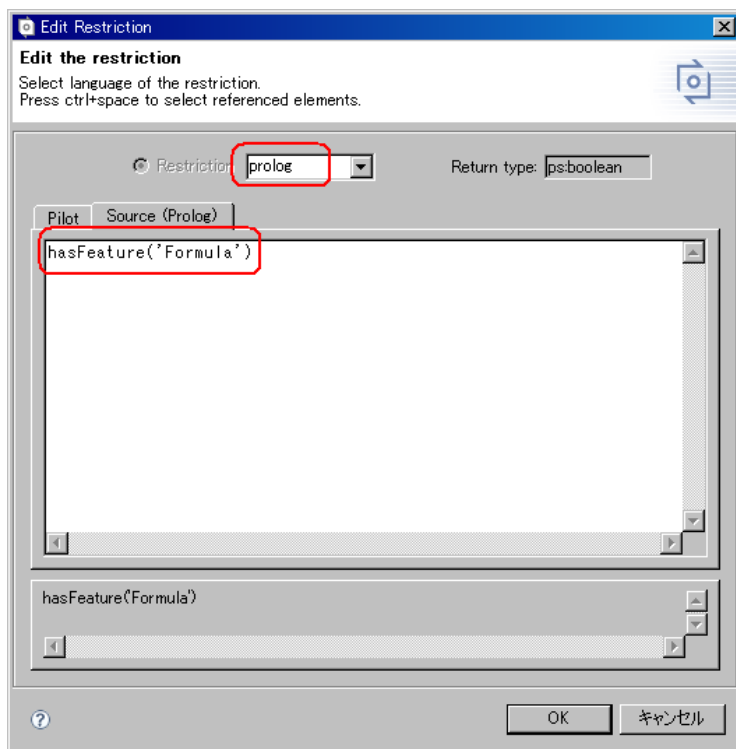
OK ボタンをクリックします。

フィーチャモデル上の値 “ number ” で右クリックし、コンテキストメニューから New -> Restriction を選択することで、制約 (`hasFeature('Number')`) を追加します。



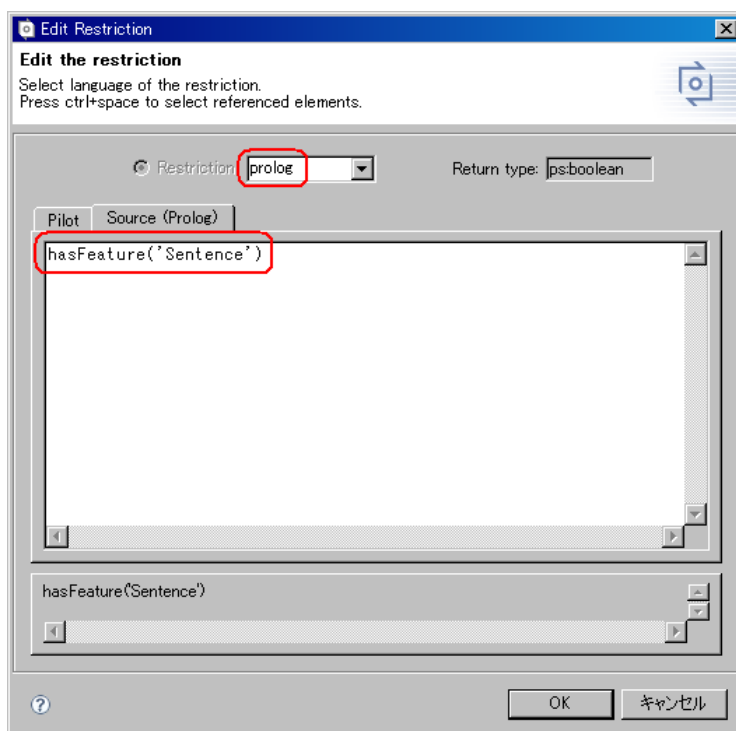
OK ボタンを押します。

これは、フィーチャ “ Number ” が選択された場合、属性 Format の値が、 “ number ” になることを示します。同様に、“ formula ” に対しても、制約 (`hasFeature('Formula')`) を追加しましょう。



OK ボタンを押します。

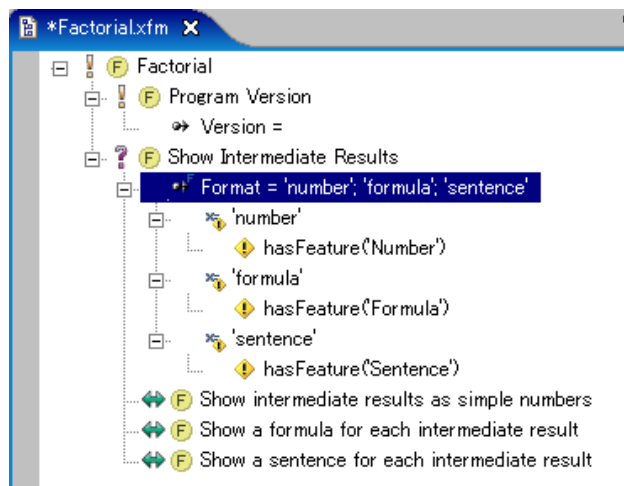
最後に、“ sentence ” に制約 (`hasFeature('Sentence')`) を追加しましょう。



OK ボタンを押します。フィーチャモデルを保存しましょう。

これらのフィーチャと属性 Format は、アプリケーションが中間結果をどんなフォーマットで出力するかを設定するために使用されます。

フィーチャモデルは、以下のようになります。



5. ファミリーモデルの設定

アプリケーションの機能をフィーチャにマップした後は、その部品をファミリーモデルにマップします。ファミリーモデルは、ファイルに対応するファミリーモデルの要素を選択することで、アプリケーションを構成するファイルを定義するだけでなく、アプリケーションファイルのトランスフォーメーション（変換）を定義することにも使用されます。ファミリーモデルにおいて、テキスト変換エレメント（*ps:context*, *ps:condxml*, *ps:fragment*）が作用します。

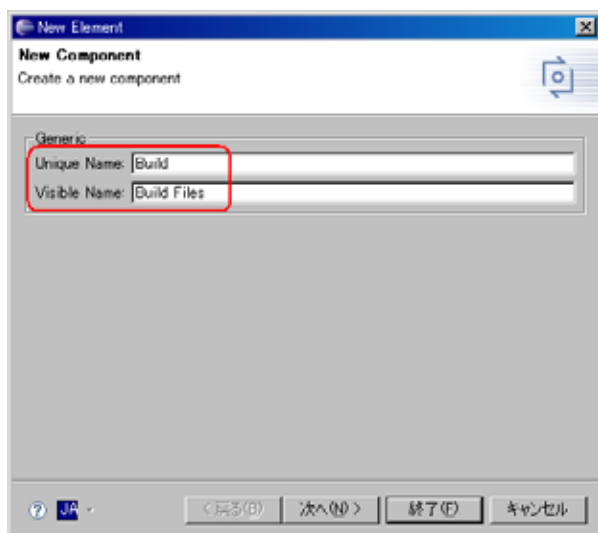
アプリケーションの部品は、Source ディレクトリ内で、異なったファイルタイプであるビルドファイル、ソースファイル、そしてドキュメントファイルの3つのグループに分類されます。

5.1. ビルドファイル


最初にビルドファイルについて説明します。この例では、アプリケーションのトランスフォーメーション/コンフィグレーションによって、これらの中身は変換されません。その代わりに、最終的に設定されたアプリケーションファイルを含むディレクトリに単純にコピーされます。したがって、ビルドファイルを指し示す、*ps:file* ソースエレメントを使用します。

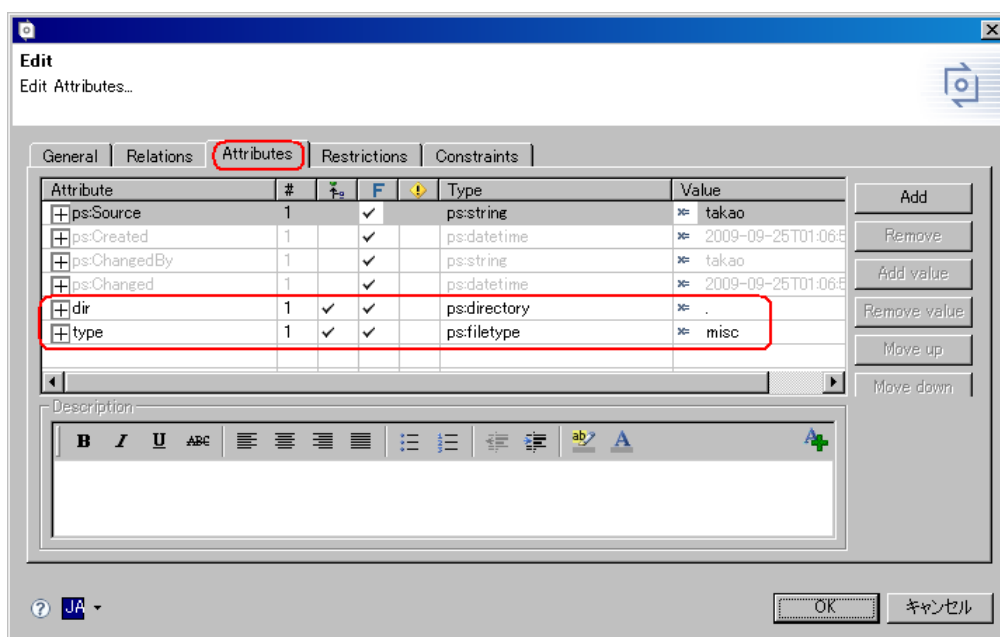
（ここでは主に、ビルド用のファイルを出力させるディレクトリの指定方法などを紹介。バリエーションごとにビルド内容を変えるような場合は、以下のテキストやソースファイルの例を参考に、コンフィグ用の *make* を作成することができます）

ファミリーモデル“ Factorial.ccfm ”を開き、モデルのルートエレメント (Factorial) で右クリックし、コンテキストメニューから New -> Component を選択します。以下のように、Unique Name に“ Build ”、Visible Name に“ Build Files ”を入力し、終了ボタンをクリックします。“ Build Files ”コンポーネントを作成します。



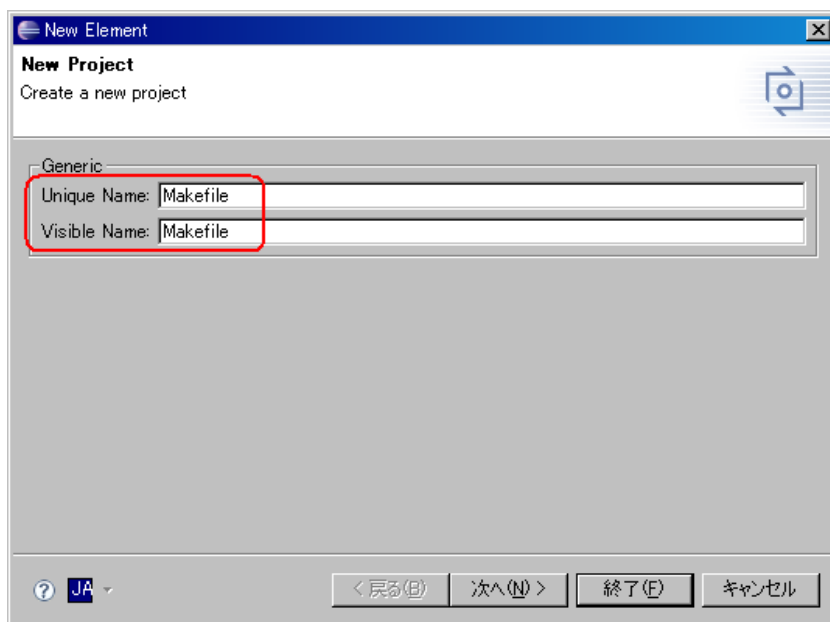
いくつかの子エレメントに対して、dir や type 属性を同じ値で使用したいので、Build Filesコンポーネントのこれらの属性をinheritable（継承）として設定します。Build Filesコンポーネント上で右クリックし、コンテキストメニューから New -> Attributeを選択します。ウィザードが開くので、Attributeに“ dir ”を入力し、Typeとして ps:directory を選択し、Value には“ . ”を設定します。

この新しい属性に inheritable オプションを設定します(3列目  をクリックしチェックを入れます)。これで、全ての子エレメントにこの値が継承され、子エレメントで定義する必要がなくなります。同様に、Add ボタンを押して、属性名が“ type ”で、Type が“ ps:filetype ”の別の属性を追加します。Value として“ misc ”(*.c、*.cc、*.cpp、*.h 以外のファイル)を設定し、inheritable オプションを設定します。以下のようになります。

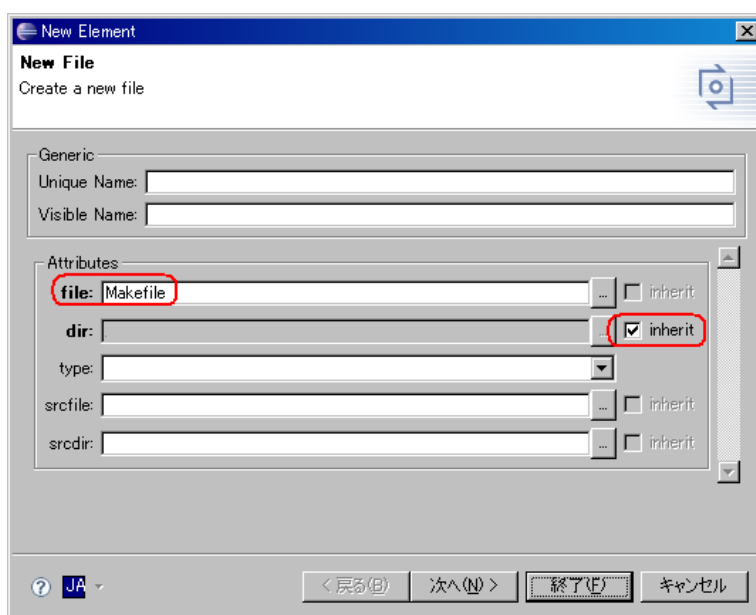


OK ボタンを押します。

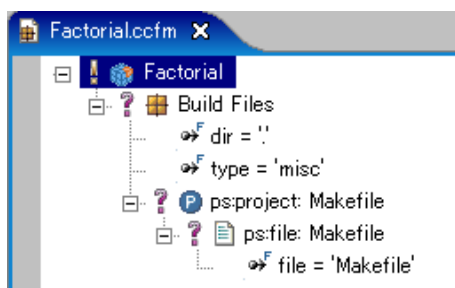
このプロジェクトには2つのビルドファイル (fact.vcproj、Makefile) があります。Build Files コンポーネント上で右クリックし、コンテキストメニューから New -> Project を選択します。Unique Name と Visible Name の両方に “ Makefile ” を入力し、終了ボタンをクリックします。



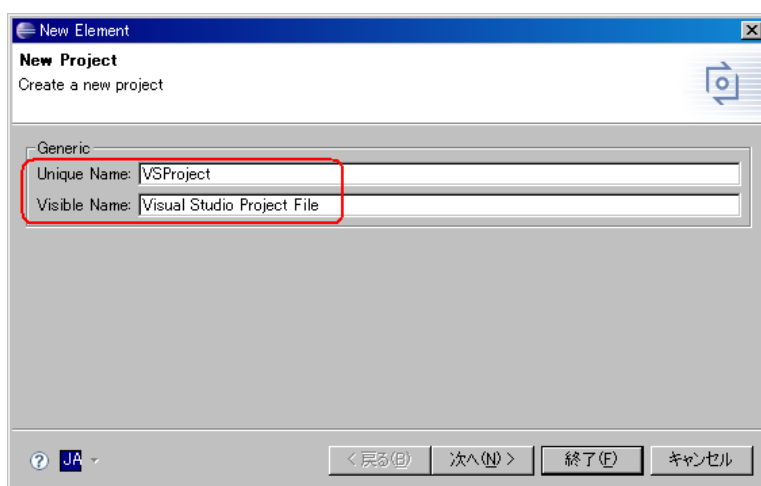
作成したエレメント “ Makefile ” で右クリックし、コンテキストメニューで New -> File を選択すると、New File ウィザードが開きます。Attributes の file に “ Makefile ” を入力します。属性 dir の inherit チェックボックスを ON にします。



終了ボタンをクリックすると、make ファイルを表す新しいエレメントが作成されます。
ファミリーモデルの空白部分で右クリックし、コンテキストメニューから Tree Layout -> Restrictions と Attributes にチェックを入れ、フィーチャモデル上に Restriction と Attribute が表示されるように設定しましょう。以下のようになります。

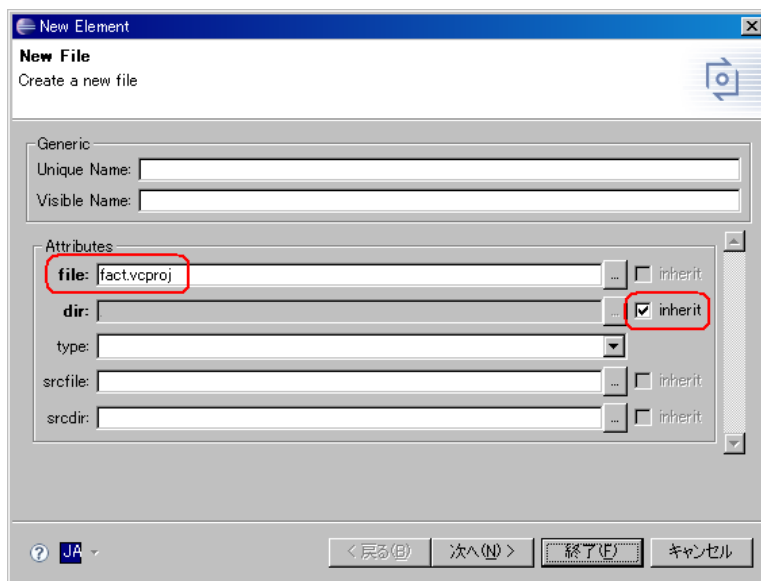


2 つ目のビルドファイルに対するモデルエレメントは、Visual Studio プロジェクトファイルで、1 つ目の make ファイルと同様の手順で作成します。Unique Name に“ VSProject ”、Visible Name に“ Visual Studio Project File ”を入力し、新しく Project エレメントを作成します。



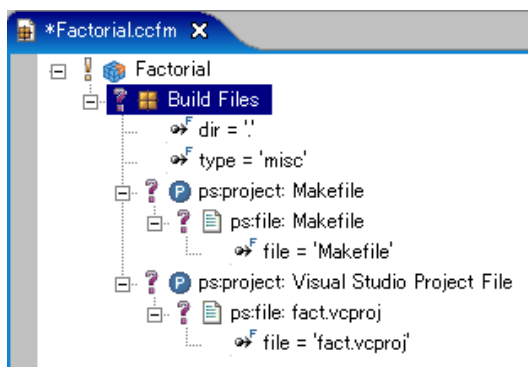
終了ボタンをクリックします。

作成したエレメント“ Visual Studio Project File ”で右クリックし、コンテキストメニューで New -> File を選択すると、New File ウィザードが開きます。Attributes の file に “ fact.vcproj ” を入力します。属性 dir の inherit チェックボックスを ON にします。



終了ボタンをクリックします。

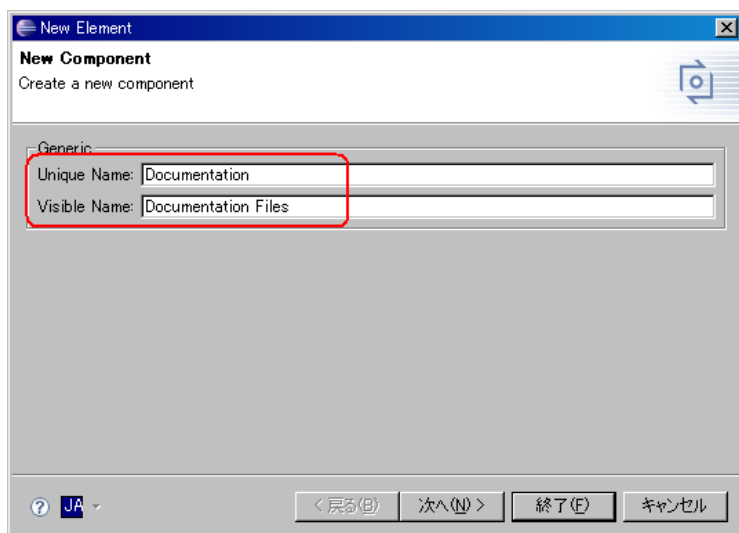
ファミリーモデルは、以下のようになります。



5.2. ドキュメントファイル

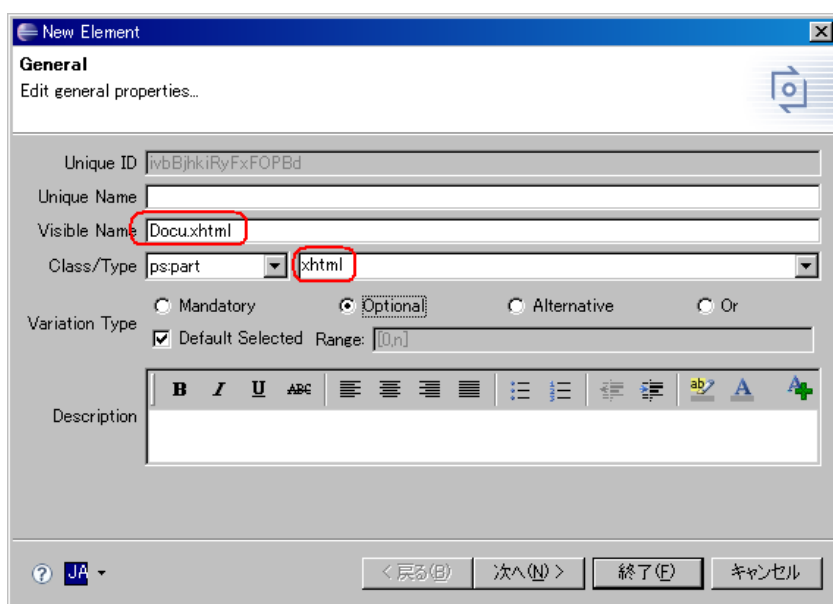
次の手順では、ドキュメントファイル“Docu.xhtml”をファミリーモデルにマップします。この例では、ビルドファイルとは違い、このファイルは、トランスフォーメーション Output ディレクトリに置かれる前に、変換されます。トランスフォーメーションの結果、選択されたフィーチャに応じたテキストのみを含むドキュメントファイルが生成されます。これは上述した XHTML ドキュメント用の、タグに出力条件を持つ特別な属性により実現できています。これら条件は、*ps:condxml* ソースエレメントによって評価されます。もし条件が False であれば、対応するタグは、ドキュメントから除外されます。

ファミリーモデルのルートエレメント配下に、新しくコンポーネントを追加し、Unique Name に “Documentation”、Visible Name に “Documentation Files” を入力します。(New -> Component)



終了ボタンをクリックします。

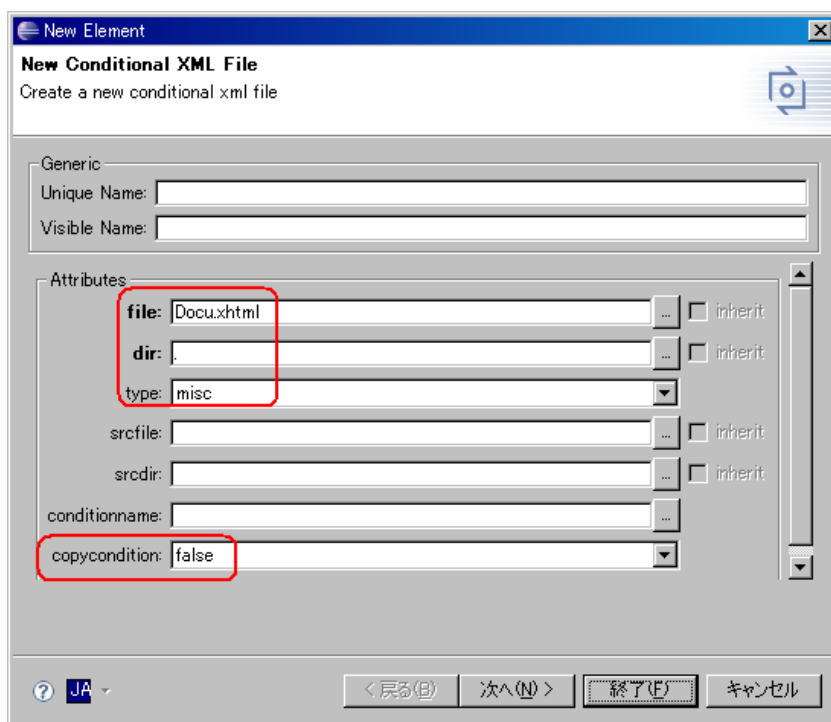
“ Documentation Files ” コンポーネントが作成され、使用できるドキュメントをまとめて管理できるようになりました。“ Documentation Files ” コンポーネント配下に、新しく Visible Name が “ Docu.xhtml ”、Type が “ xhtml ” であるファミリーエレメントを作成します。“ Documentation Files ” コンポーネントで右クリックし、コンテキストメニューから New -> Family Element を選択して作成します。



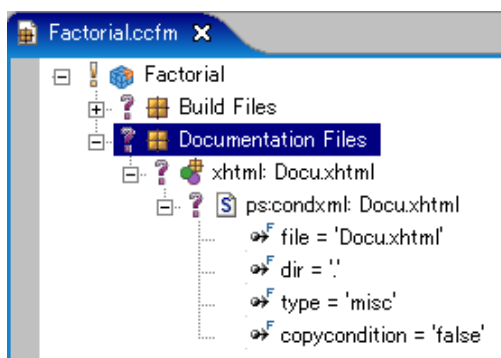
終了ボタンをクリックします。

上記で作成したファミリーエレメント (Docu.xhtml) 配下に、新しく Conditional XML エlementを追加します。“ xhtml: Docu.xhtml ” で右クリックし、New -> Conditional XML を選択し、ウィザードを開きます。Attributes の file に “ Docu.xhtml ”、dir に “ . ”、type に “ misc ” (*.c、*.cc、*.cpp、*.h 以外のファイルを表す)、copycondition に “ false ” を設定します。copycondition に “ false ” を設定す

ることで、条件を含む XHTML ドキュメントのタグにある特別な属性は、条件が評価された後、除外されます。これらの属性は、有効な XHTML ではないため、除外されることは重要です。



終了ボタンをクリックします。ファミリーモデルは、以下のようにになります。



ps:condxml ソースエレメントに関する詳細は、pure::variants ユーザーズガイドのセクション“ Chapter 9. Reference ”を参照してください。

5.3. ソースファイル

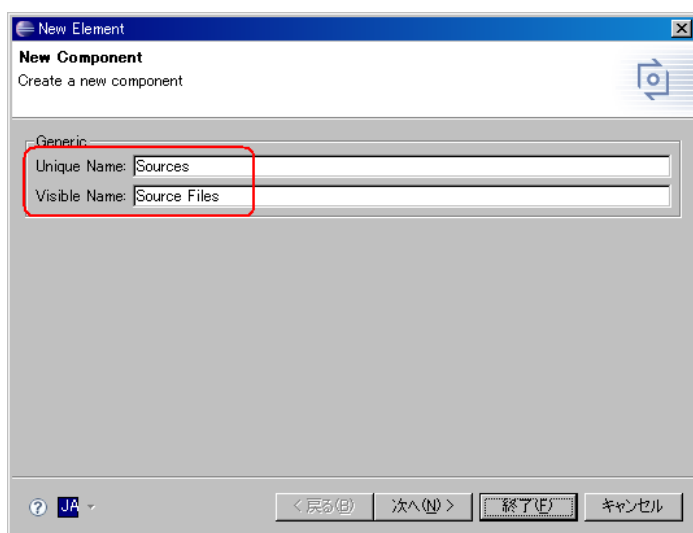
最後に、ソースファイル (fact.cc、 Factorial.h) をファミリーモデルにマップします。両方のファイルは、変換されます。

“ fact.cc ”には、関数 info の呼び出しが含まれます。この関数は、トランスフォーメーション中に生成


され、アプリケーションの名前、バージョン、ソースファイルの生成日時(トランスフォーメーション実行日時)を出力します。関数 info は、*ps:fragment* ソースエレメントを使って生成されます。

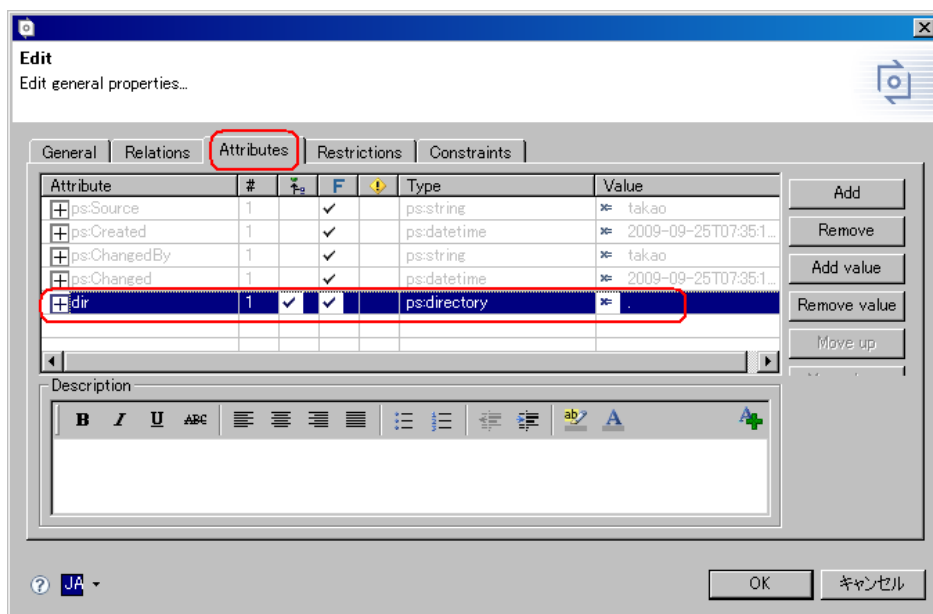
“Factorial.h”には、中間結果を出力するために、コードに対するガードとして、条件を使った特別なマクロが含まれています。これらのマクロは、*ps:condtext* ソースエレメントを使うことで評価されます。

Unique Name が “Sources”、Visible Name が “Source Files” である新しいコンポーネントを作成します。



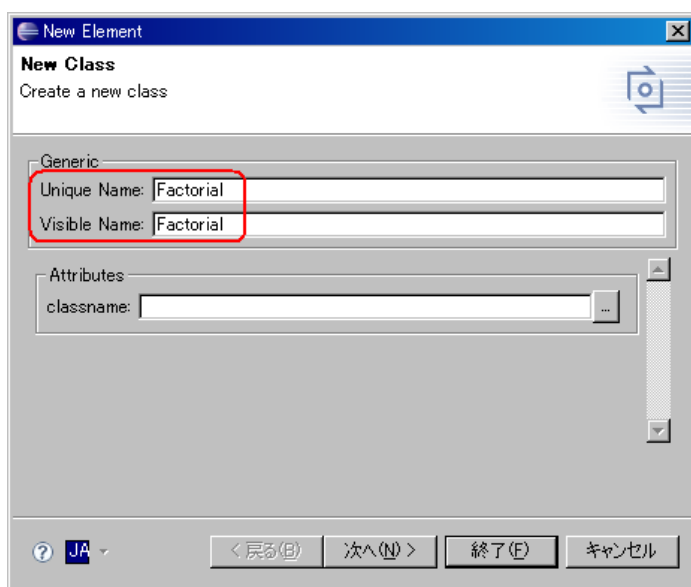
終了ボタンをクリックします。

そのコンポーネントに、type が “ps:directory”、Value が “. ”、inheritable オプションを設定(3列目  をクリックしチェックを入れます)した属性 dir を新規に作成します。“Source Files” コンポーネントで右クリックし、コンテキストメニューから New -> Attributes を選択し作成します。

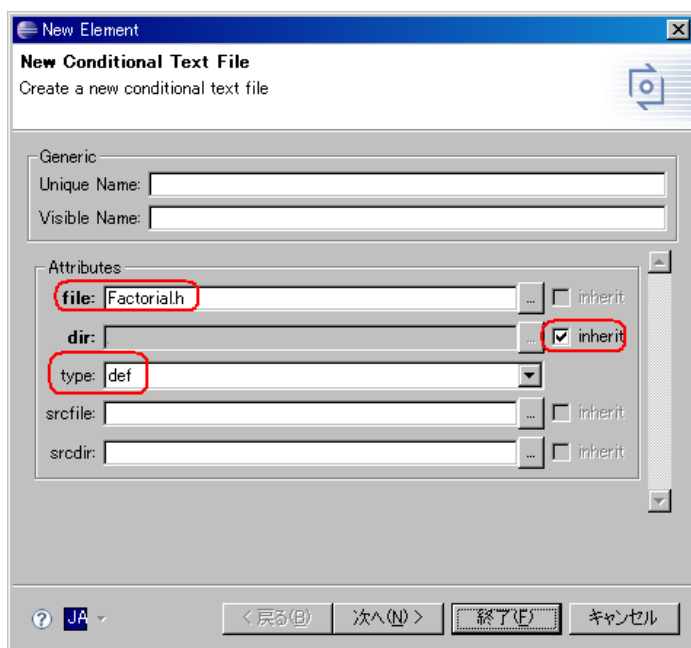


OK ボタンをクリックします。

クラス Factorial を持つヘッダファイルを追加します。コンポーネント “ Source Files ” の配下に、新たに Class エlementを作成します。Source Files で右クリックし、コンテキストメニューから New -> Class を選択し、New Class ウィザードを開きます。Unique Name と Visible Name に “ Factorial ” を入力し、終了ボタンをクリックします。

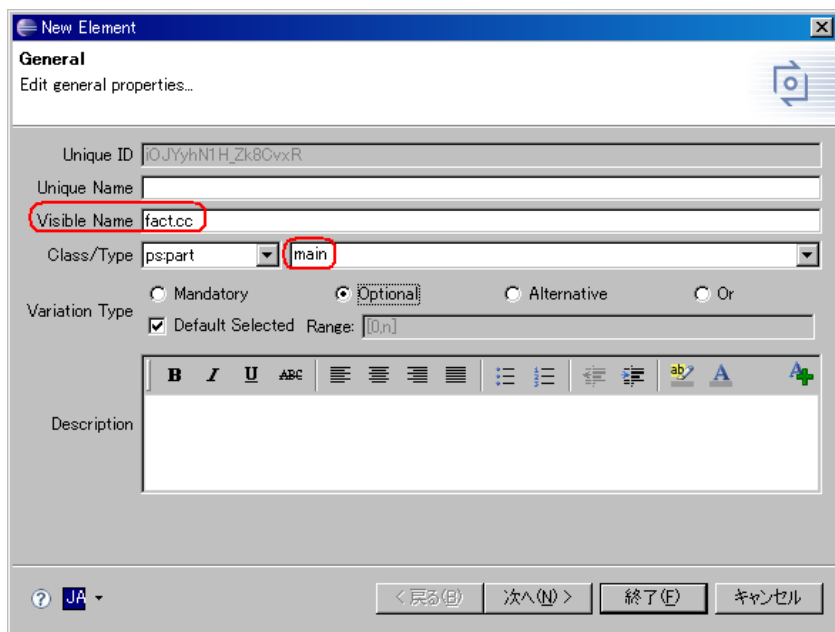


作成した Factorial エlement配下に、新たに Conditional Text ソースElementを作成します。Factorial で右クリックし、New -> Conditional Text を選択します。New Conditional Text ウィザードで、file に “ Factorial.h ”、type に “ def ” (*.h のヘッダファイルを表す)、dir の inherit ボタンにチェックを入れ、終了ボタンをクリックします。




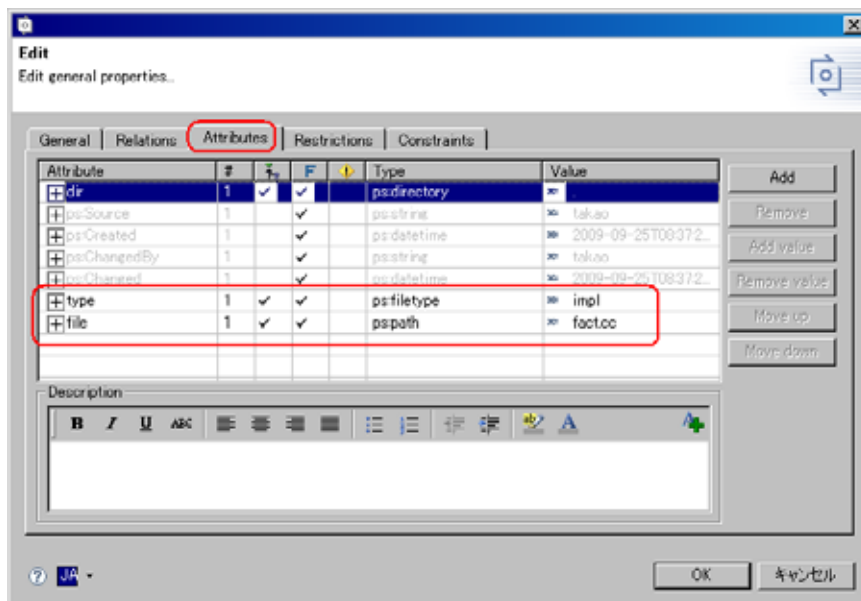
ps:condtext ソースエレメントに関する詳細は、*pure::variants* ユーザーズガイドのセクション“ Chapter 9. Reference ”を参照してください。

2 つ目のソースファイル“ *fact.cc* ”に対して、コンポーネント“ Source Files ”配下に、*type* が“ *main* ”、Visible Name が“ *fact.cc* ”である新しいファミリーエレメントを作成します。Source Files で右クリックし、New -> Family Element を選択します。



終了ボタンをクリックします。

作成したエレメント *main* に、*Type* が“ *ps:filetype* ”、*Value* が“ *impl* ”（*.c、*.cc、*.cpp といったファイルを表す）である属性“ *type* ”と、*Type* が“ *ps:path* ”、*Value* が“ *fact.cc* ”である属性 *file* を追加します。共に *inheritable* オプションを設定します（3 列目  をクリックしチェックを入れます）。“ *main* ”エレメントで右クリックし、コンテキストメニューから New -> Attributes を選択し作成します。これらの属性は、次に作成する *ps:fragment* エレメントで必要になります。



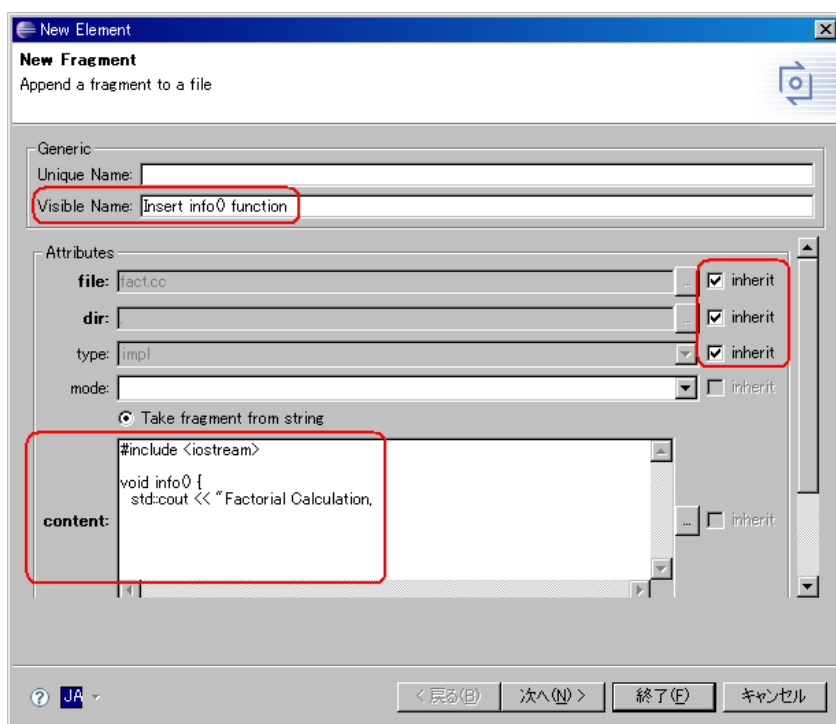
main エLEMENTの配下に、Fragment エLEMENTを新たに作成します。main エLEMENTで右クリックし、New -> Fragment を選択します。New Fragment ウィザードで、Visible Name に “ Insert info() function ”を入力し、Attributes の file、dir、type の inherit チェックボックスにチェックを入れます。content フィールドに以下のテキストを入力し、終了ボタンをクリックします。

content フィールド :

```
#include <iostream>

void info() {

    std::cout << "Factorial Calculation,
```

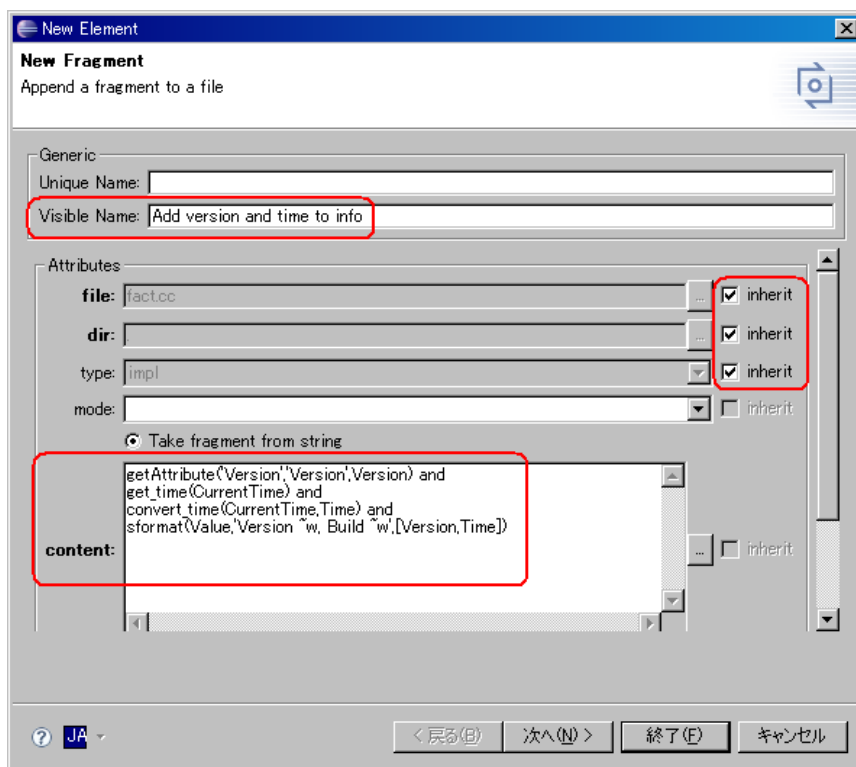


これは、関数 info の最初の部分です。トランスフォーメーション出力ディレクトリ内で、ファイル fact.cc へ書き込まれます。次の Fragment では、ファイル fact.cc にバージョン番号とトランスフォーメーション実行日時を付加します。

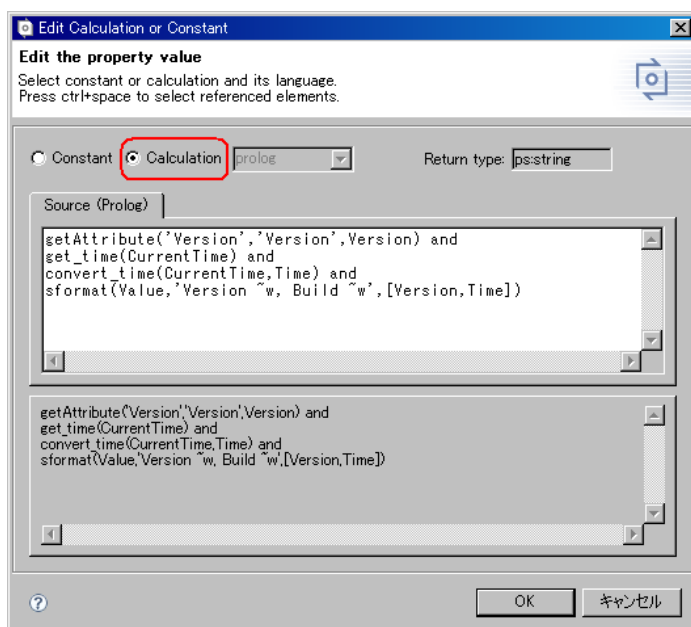
main エLEMENT配下に、2 つ目の Fragment エLEMENTを作成し、Visible Name を “ Add version and time to info ” と入力します。Attributes の file、dir、type の inherit チェックボックスにチェックを入れ、content フィールドに以下のテキストを入力し、終了ボタンをクリックします。

content フィールド:

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



content の値は、単なるテキストではなく計算処理です。作成した Fragment エlement でダブルクリックし、Properties ダイアログを開きます。Attributes タブに切り替え、content の Value フィールドをクリックします。右端の “...” ボタンを押して、ダイアログで “Calculation” に変更し、OK ボタンをクリックします。



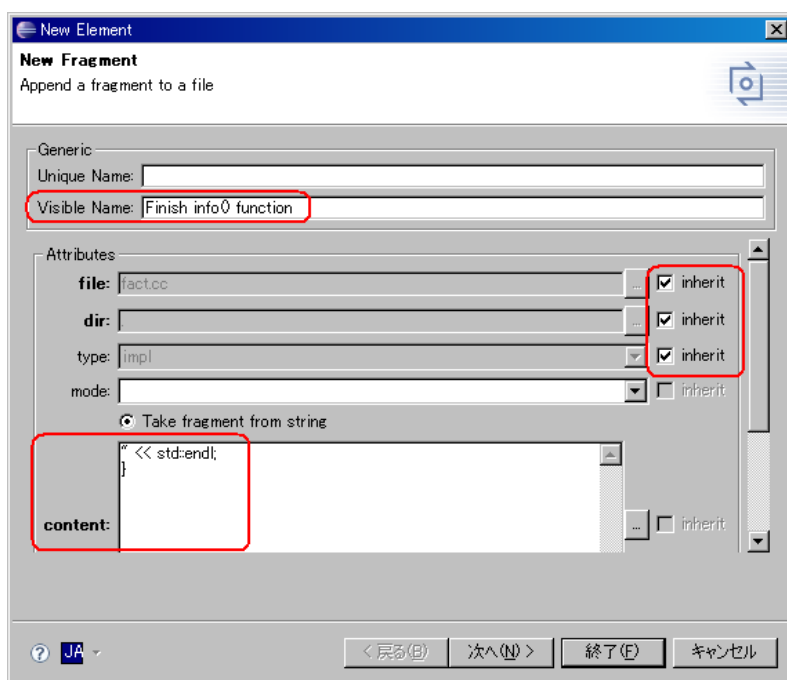
この計算処理は、フィチャモデルの Element “Version” の属性である Version の値を取得することから始まり、この値を変数 Version に保存します。それから現在の日時を取得し、変数 CurrentTime に保存します。変数 Time には、現在の日時に対応する、人が読める形式の日時が記入されます。最後に、変数 Value に保存された計算結果は、バージョンや現在の日時が挿入された文字列で、これは、ト

ランスフォーメーション中に “ fact.cc ” に付加されます。

次の Fragment には、関数 info の残りの部分が含まれます。main エlement 配下に、3 つ目の Fragment エlement を作成し、Visible Name を “ Finish info() function ” と入力します。Attributes の file、dir、type の inherit チェックボックスにチェックを入れ、content フィールドに以下のテキストを入力します。

content フィールド:

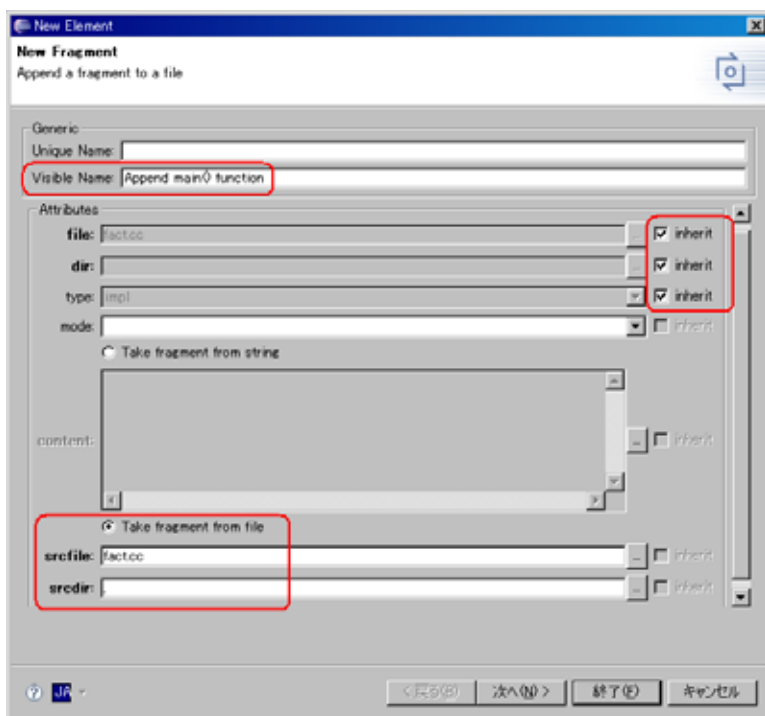
```
" << std::endl;
}
```



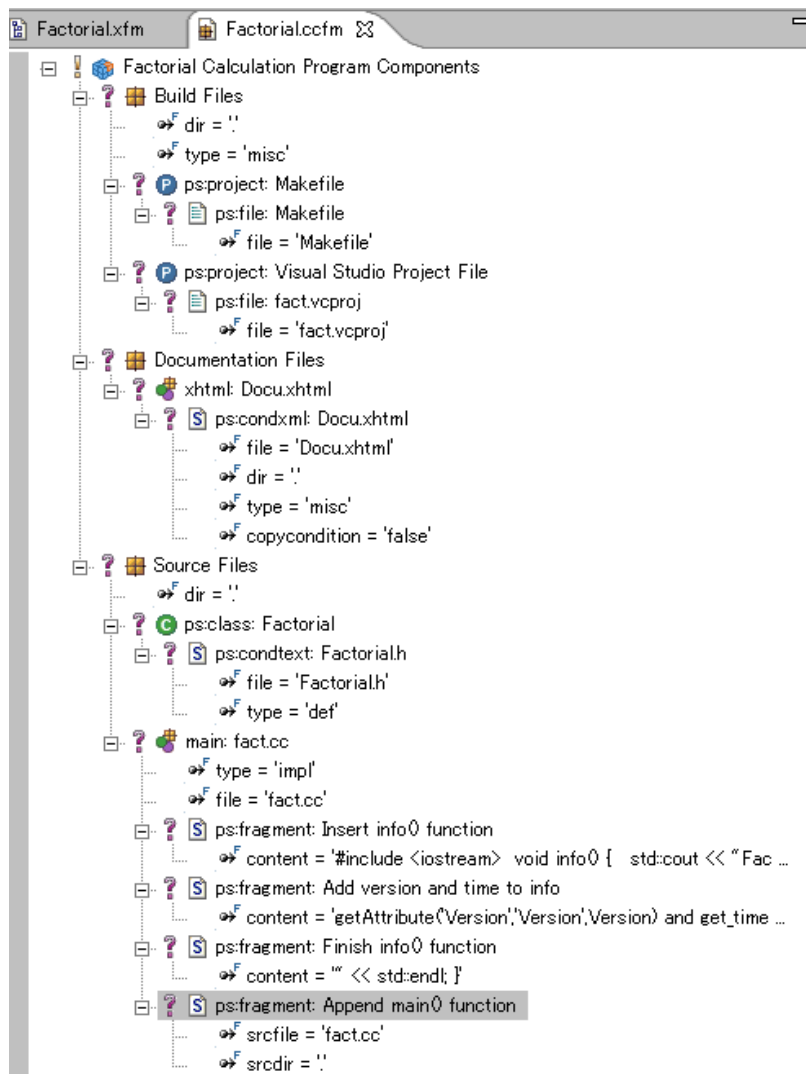
終了ボタンをクリックします。

関数 info は、完成しました。最後の Fragment では、ランスフォーメーション Output ディレクトリのファイル “ fact.cc ” に、fact.cc オリジナルの内容を付加します。

main エlement 配下に、Fragment エlement を新規作成し、Visible Name を “ Append main() function ” と入力します。Attributes の file、dir、type の inherit チェックボックスにチェックを入れます。他の Fragment とは違い、fragment のテキストは、content ではなく、ファイルから設定します。“ Take fragment from file ” ボタンをクリックし、srcfile に “ fact.cc ”、srcdir に “ . ” を入力し、終了ボタンをクリックします。

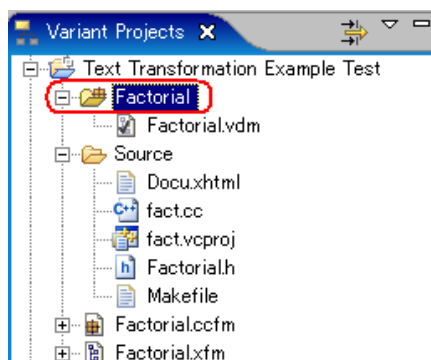


最終的にファミリーモデルは、以下ようになります。

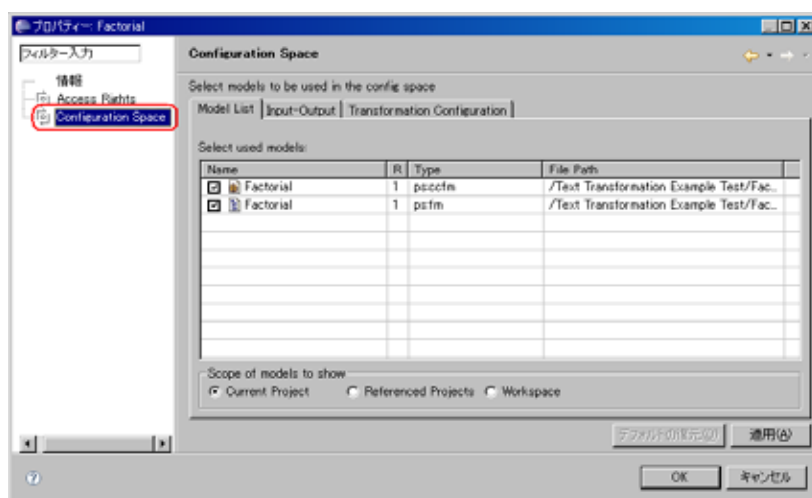


6. トランスフォーメーションの設定

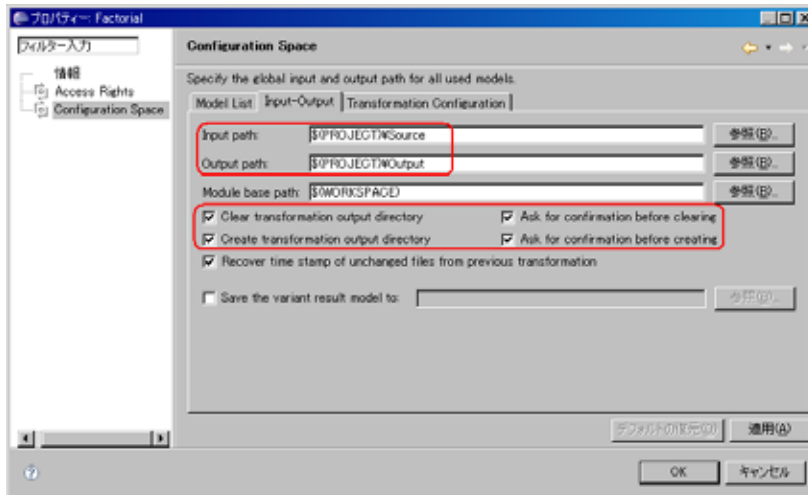
トランスフォーメーションに対応するために、いくつかのコンフィグレーションオプションを変更する必要があります。Variant Projectビューの Text Transformation Example Test プロジェクトのConfiguration space (Factorial) を選択します。



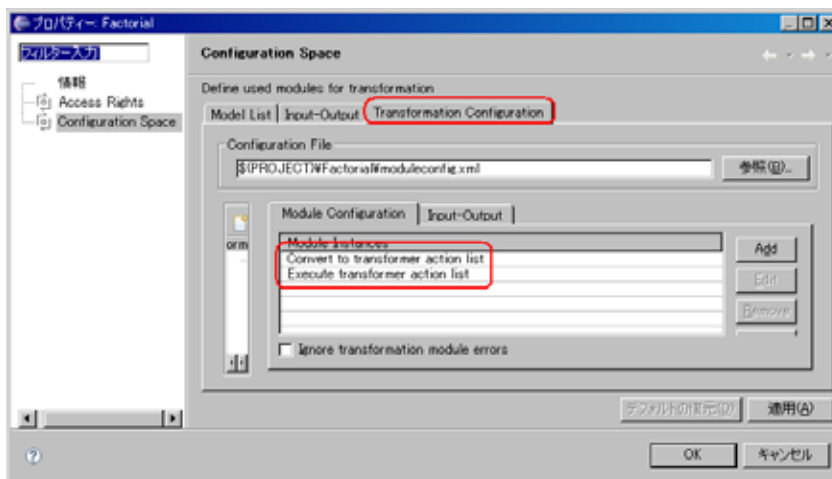
右クリックでConfiguration spaceのコンテキストメニューから、Propertiesを選択し、ダイアログでConfiguration Spaceに切り替えます。



ダイアログの Input-Output タブを開き、トランスフォーメーションに対して、“ Input path ” にインプットディレクトリとして “ \$(PROJECT)¥Source ” を、“ Output path ” にアウトプットディレクトリとして “ \$(PROJECT)¥Output ” を入力します。“ Clear transformation output directory ” と “ Create transformation output directory ” チェックボックスにチェックを入れます。

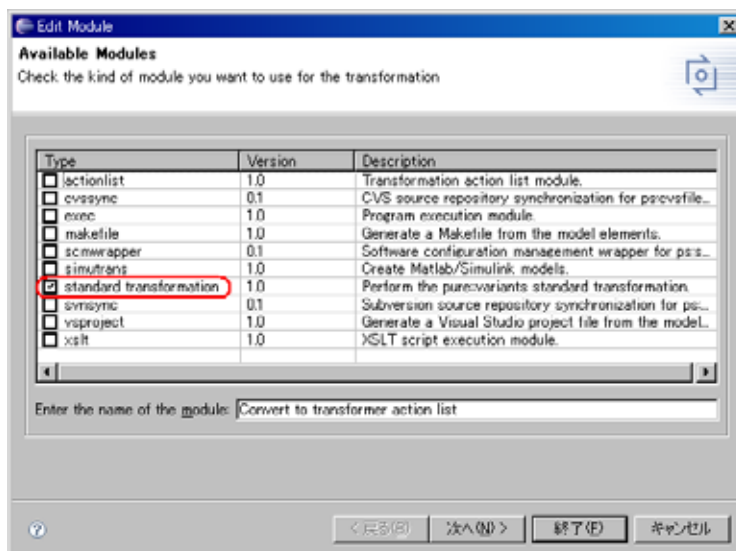


Transformation Configuration タブに切り替えます。

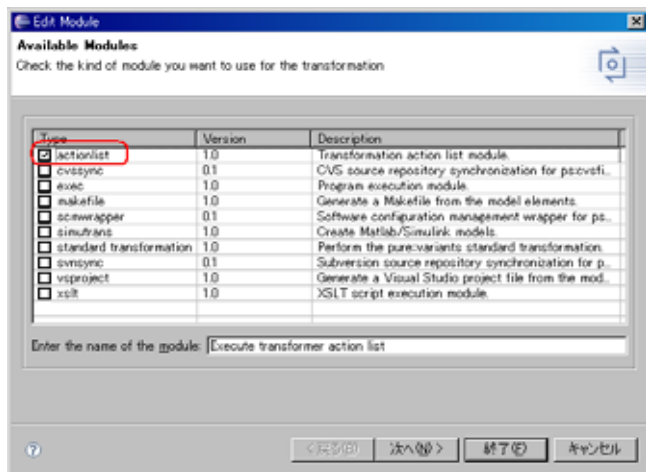


ここでは、トランスフォーメーションを実行する時に行う動作設定の確認を行います。

Module Configuration にある “ Convert to transformer action list ” を選択し、Edit ボタンをクリックします。開いたダイアログで、“ standard transformation ” モジュールにチェックが入っていることを確認し、終了ボタンをクリックしましょう。“ standard transformation ” モジュールは、コピーするファイルのリストを作成します。



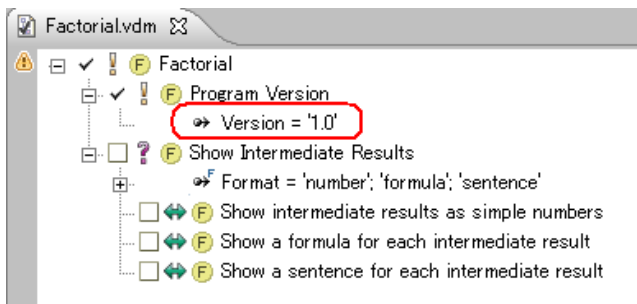
同様に、Module Configuration にある “Execute to transformer action list” を選択し、Edit ボタンをクリックします。開いたダイアログで、“actionlist” モジュールにチェックが入っていることを確認し、終了ボタンをクリックしましょう。“actionlist” モジュールは、上記で作成したリストにあるファイルのコピーを実行します。




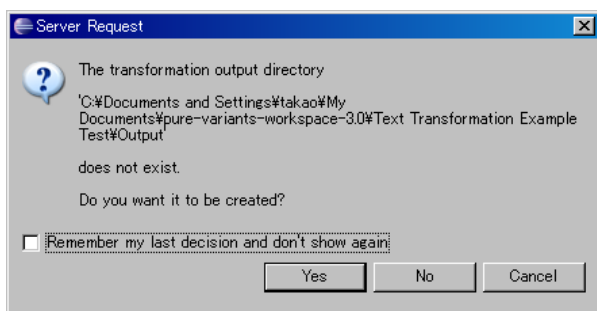
OK ボタンをクリックします。

7. バリエント生成

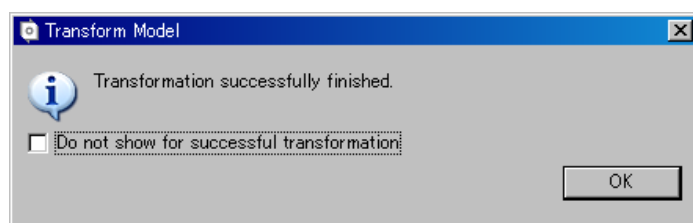
pure::variants プロジェクトとアプリケーションファイルにより、最初のトランスフォーメーションを開始する準備ができました。Configuration space フォルダにある “Factorial.vdm” をダブルクリックすることでバリエントモデルが開きます。フィーチャ “Program Version” の属性 “Version” の値に、“1.0” を入力します (Version をダブルクリックして入力します)。ここでは、アプリケーションのバージョンを “1.0” と設定しましょう。



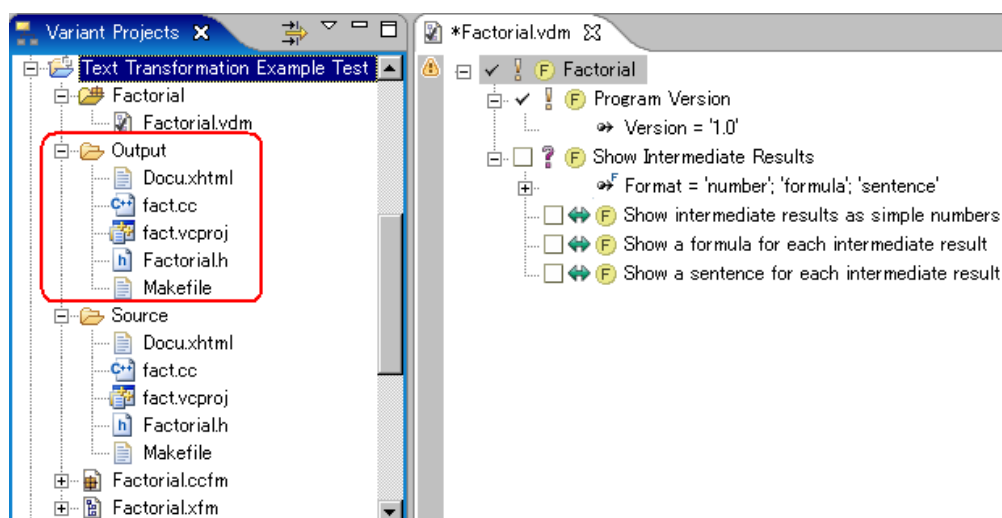
トランスフォーメーションを開始するために、ツールバーの Transform Model ボタン () をクリックします。Output ディレクトリを作成するかどうかを促すダイアログが表示されるので、Yes をクリックして作成します。



トランスフォーメーション完了のダイアログが表示されるので、OK をクリックします。



トランスフォーメーション完了後、“Text Transformation Example Test”プロジェクトを選択し、F5 ボタンを押すことで、Variant Projects ビューのプロジェクトが更新され、変換されたアプリケーションファイルを含む新しいディレクトリ“Output”が現れます。



Output ディレクトリ内の“fact.cc”は、下記コードになります（日時の違いは除きます）。

[fact.cc]

```
#include <iostream>

void info() {
    std::cout << "Factorial Calculation, Version 1.0, Build Mon Sep 28 12:57:28 2009" << 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;
}
```

クラス Factorial に対するコード (Factorial.h) とドキュメントファイル (Docu.xhtml) は、トランスフォーメーション後、同様に変更されます。Output ディレクトリ内のこれらのファイルは、以下のようになります。

[Factorial.h]

```
//
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;
}
};
```

[Docu.xhtml]

Factorial Calculation Program

Usage

The only argument of the program is a number for which the factorial is calculated.

Result

The result of invoking the program is a formula like:

$3! = 6$

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!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>
</body>
</html>

```

バリエーションモデルで、フィーチャ “ Show Intermediate Results ” とその配下の “ Show a sentence for each intermediate result ” 選択します。Transform Model ボタンを押してトランスフォーメーションを実行し、プロジェクトを更新しましょう。“ Factorial.h ” と “ Docu.xhtml ” は、以下のようになります。(赤字は、最初のトランスフォーメーションの実行結果と異なる箇所です)

[Factorial.h]

```

//
#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;
}
};

```

[Docu.xhtml]

Factorial Calculation Program

Usage

The only argument of the program is a number for which the factorial is calculated.

Result

The result of invoking the program is a formula like:

$3! = 6$

Intermediate Results

Intermediate results are shown as sentences, e.g. Factorial of 3 is 6.

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!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>
Intermediate Results
</h2>
<p>
<p>
Intermediate results are shown as sentences, e.g. Factorial of 3 is 6.
</p>
</p>
</body>
</html>

```

例えば、下記のように、それぞれで生成されたドキュメントファイルを並べてみると違いがわかります。左側は、最初に生成されたドキュメント、右側は、フィーチャ “ Show Intermediate Results ” とその配下の “ Show a sentence for each intermediate result ” を選択した時のドキュメントです。

<p>Factorial Calculation Program</p> <p>Usage</p> <p>The only argument of the program is a number for which the factorial is calculated.</p> <p>Result</p> <p>The result of invoking the program is a formula like:</p> <p>3! = 6</p>	<p>Factorial Calculation Program</p> <p>Usage</p> <p>The only argument of the program is a number for which the factorial is calculated.</p> <p>Result</p> <p>The result of invoking the program is a formula like:</p> <p>3! = 6</p> <p>Intermediate Results</p> <p>Intermediate results are shown as sentences, e.g. Factorial of 3 is 6.</p>
--	---

このように、ソースエレメント (*ps:fragment*, *ps:condtext*, *ps:condxml*) を使って、選択されたフィーチャに対応したファミリーモデルの部品 (ソースファイル、ドキュメントファイルなど) を生成することができるようになっています。

以上