pure::variantsによるテキスト変換チュートリアル

目 次

1. 概要

このチュートリアルでは、pure::variants でバリアントごとにドキュメントを生成する方法を紹介します。ここでのドキュメントは、英語や日本語など言語によらずにテキストや XML などあらゆるフォーマットのものであり、仕様書やソースコード、make用のファイルなどが想定されます。

バリアントの定義に従ってこれらのドキュメントを生成させるためは、ファミリーモデルで（*ps:fragment*, *ps:pvscltext*, *ps:pvsclxml*）型のソースエレメントを使用します。それらをファミリーモデルで設定し、バリアント定義したモデルをモデル変換（Transformation）すると、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が生成されます。

- *ps:fragment* 型のソースエレメントは、あるファイルに対して、テキストや別のファイルを追加するために使用します。
- *ps:pvscltext* 型のソースエレメントは、テキスト文書をコピーしたり、そのコピーをファイルに保存するために使用します。フィーチャの選択に応じてテキスト文書内にある「条件」を評価して、文書の一部分を出力ファイルにコピーしたり、追加のテキストを挿入するために使用します。この「条件」は、コメントのような形式を使用して指定することで表現されます。
- *ps:pvsclxml* 型のソースエレメントは、XML 文書をコピーしたり、コピーをファイルに保存するために使用します。フィーチャの選択に応じてXML文書のノードにある「条件」が評価され、そのノード（や

以下のサブノード）を出力ファイルにコピーするかどうかが制御されます。この「条件」は特別な属性をXML文書のタグに設定することで表現されます。

以下、例題プロジェクト「入力された数値の階乗を求めるCアプリケーション」を用いて、これらのソースエレメントの使用法について説明します。このアプリケーションでは、計算結果の出力フォーマットを指定したり、実行時にバリアントに依存するバージョン番号を表示できます。すなわち、表示フォーマットの違いやバージョン情報がバリアントとなります。

このプロジェクトには、Cアプリケーションのソースコードだけでなく仕様説明ドキュメントやアプリケーションをビルドするためのファイルも含まれます。それらのドキュメントもバリアントに対応したものが生成されます。

本チュートリアルではプロジェクトを一から作る手順を紹介していますが、実施結果として得られるプロジェクトをサンプルプロジェクトとして弊社サイトで公開しています[1]。

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のプロジェクトを作成します。

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）を作成して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. 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モデル変換の設定を行う

[1] 同内容のプロジェクトが pure::variants のサンプルの Variant Management Tutorials で Text Transformation として提供されており、本資料はそれに基づき一部変更して構成したものです。

なお、本資料のスクリーンショットは pure::variants バージョン 5 のものですが、バージョン 6 でも動作を確認しています。

2. チュートリアルについて

このチュートリアルを進めるには pure::variants と標準モデル変換（Standard Transformation）の動作についての理解が必要です。次でそれらの概要と例題プロジェクトの説明を行いますが、本チュートリアルを実施する前に pure::variants 導入資料を参照してください。

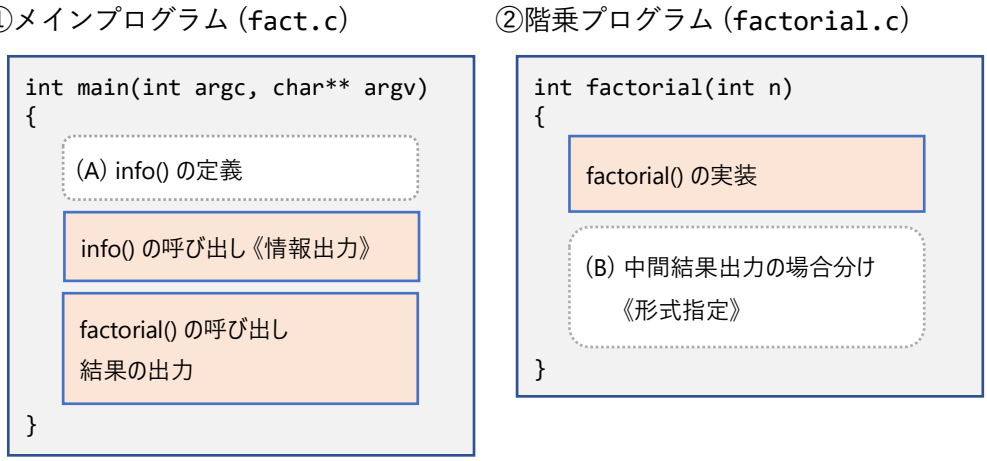
各種チュートリアルや導入資料は、弊社サイトから参照できます。
https://www.fuji-setsu.co.jp/products/purevariants/index.html より、説明資料：「ソースコード、フラグのバリアント管理」もご覧ください。

アプリケーション素材の概要

3 章以降での詳細説明の前提となるよう、以下で本例題プロジェクトの概略を述べます。

（1）階乗プログラム

階乗を計算する C プログラムで、メインプログラム、階乗の実装、の 2 ファイルからなります。（下図で、《情報出力》や《形式指定》がバリアントに対する部分となります）



（2）バリアントと、そのソースやドキュメントへの反映

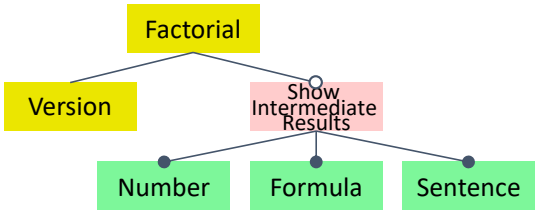
このアプリケーションのバリアントは以下のものを想定します。

■ プログラムのバージョン番号（バリアントごとに設定できます）

■ 計算の中間結果（階乗計算の途中結果）を出力する際の形式の違い
これはオプショナルな機能であり、以下のいずれかが選択されます

- 中間結果の値だけ（たとえば「6」だけが表示される）
- 中間結果の計算式を示す（たとえば「3! = 6」のように表示される）
- 中間結果を文の形で示す（たとえば「Factorial of 3 is 6.」のように表示される）

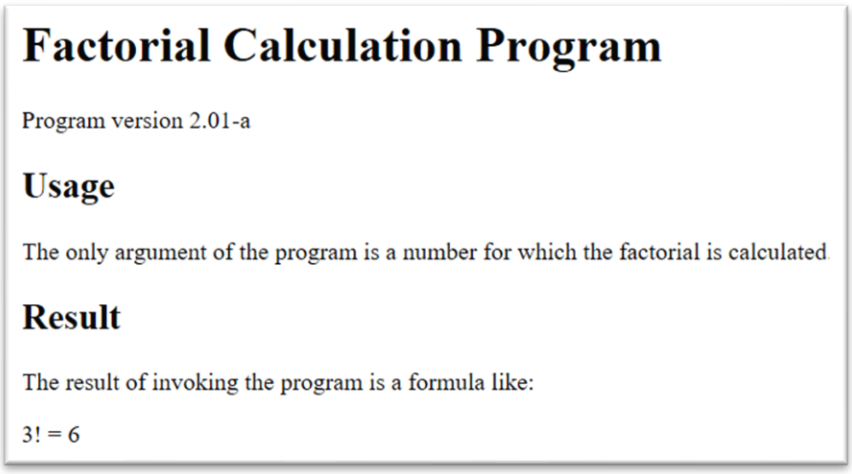
フィーチャモデルのツリー表現は下図のようになり、上記のバリアントはフィーチャ Version と Number、Formula、Sentence で制御されます。



これらのバリアントは、pure::variants のモデル変換の過程で ① や ② のソースファイル中にコードテキスト（(A) や (B)）を挿入することで実現します[2]。

（3）概略仕様ドキュメント

ドキュメントは以下のように、このプログラムの概要を示す HTML 文書です。



Factorial Calculation Program

Program version 2.01-a

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

（4）ビルド用ファイル　Makefile です。

（5）バリアント生成のイメージ

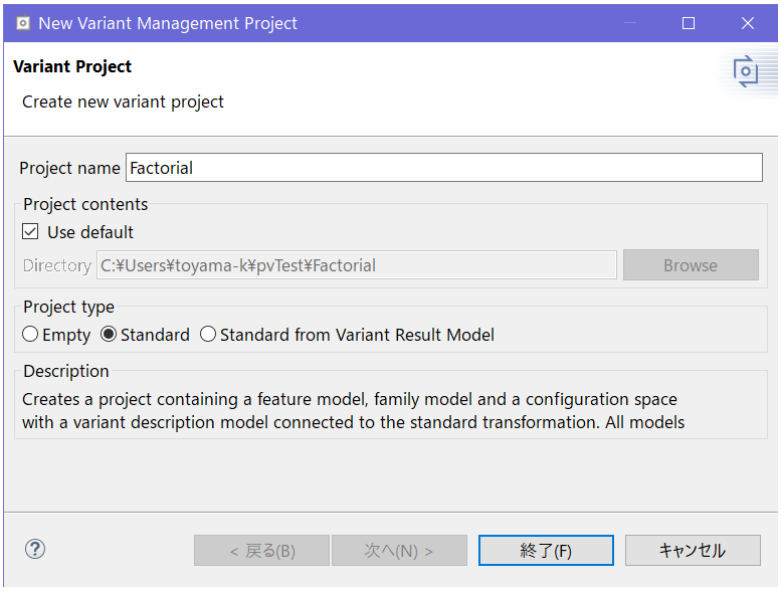
フィーチャモデルで想定されるすべてのフィーチャを実装するソースコードからバリアントで特定された各可変要素に対応するソースコードが、テキスト変換によって実現されます。概略仕様ドキュメントは vdm での指定によって、すべての場合を含む HTML ファイルから生成します。

上記の情報を前提として、以下の章でこれらが設定される様子を説明します。

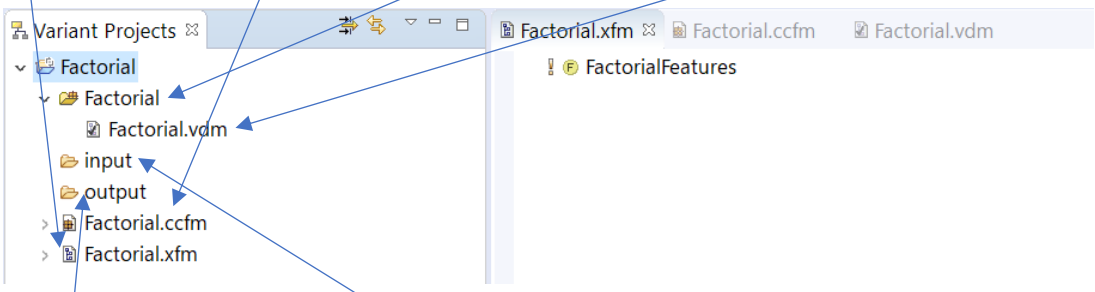
[2] バリアントを実現するには、本資料での方法以外にマクロ定義ファイルを使用した設定や、ビルド時の Makefile でのオプションによる設定などがあります。それらについては、弊社サイトで導入資料として紹介している「ソースコード、フラグのバリアント管理」などを参照ください。

3. pure::variants プロジェクトの作成

pure::variants プロジェクトを新規作成します。pure::variants を起動すると、バリアント管理パースペクティブが既定で開きますので、Variant Projects 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[3]。



ウィザードで Project name に「Factorial」と入力し、Project type で「Standard」を選択して 終了 すると、下図のように、必要なモデルすべてを含んだ初期プロジェクトが作成されます。プロジェクト内にはフィーチャモデル、ファミリーモデル、Configuration Space（中にバリアントモデルが 1 つ）があり、すべてが自動的にオープンされます。



以降で、アプリケーションファイル（ソースファイル、ドキュメントファイル、ビルドファイル）を作成します。これらは、プロジェクト内の input フォルダに置きます。

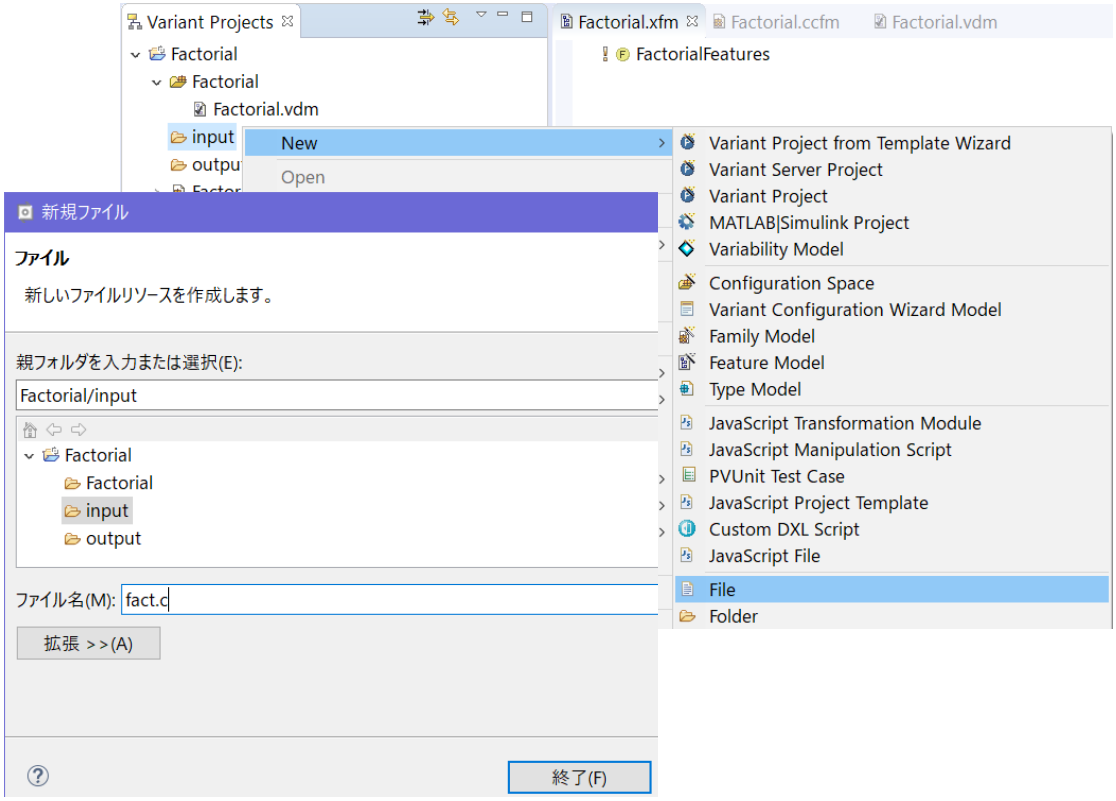
なお、output はバリアントが出力されるフォルダです。

[3] pure::variants のメニューで ファイル > 新規 > プロジェクト から 新規プロジェクト ウィザードで Variants Project を選択してもできます。

3.1. ソースファイル

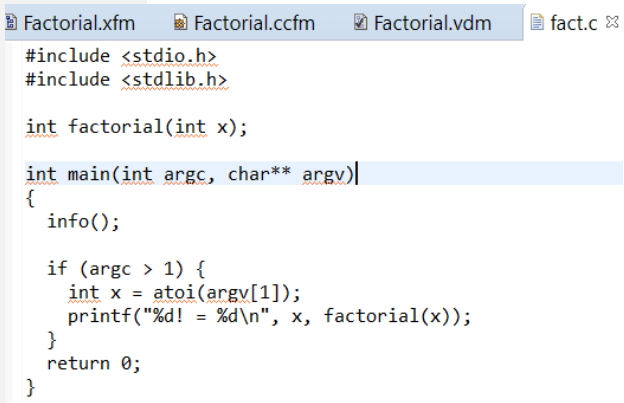
本例では、ソースファイルとして `fact.c`（アプリケーションのメインプログラム）と `factorial.c`（階乗計算プログラム）を用います。

まず、input フォルダに `fact.c` を作成します。ファイルの新規作成には input フォルダを右クリックし New > File を選択します。



ファイル名 に fact.c と入力して 終了 し、Variant Project ビューの fact.c で右クリックして Open with > テキスト･エディター を選択し、`fact.c` を開いてコードを以下のコードを入力します[4]。

```
#include <stdio.h>
#include <stdlib.h>

int factorial(int x);

int main(int argc, char** argv)
{
  info();

  if (argc > 1) {
    int x = atoi(argv[1]);
    printf("%d! = %d\n", x, factorial(x));
  }
  return 0;
}
```



[4] デフォルトのテキストエディタが設定されている場合は、それが起動して `fact.c` の入力画面となります。

ここで main()は、最初に info()を呼び出してプログラム名とバージョン番号を出力した後、factorial()を呼び出して実行時引数で与えられた数値の階乗を計算し、結果を出力します。

info()の内容はモデル変換の実行時に *ps:fragment* 型のソースエレメントを使って生成され、アプリケーションの名前とバージョンを出力するためのコードです（モデル変換の項で説明します）。

同様の手順で factorial.c を以下の内容で input フォルダに作成します。factorial.c では階乗計算のアルゴリズムを実装しています。

```
// PVSCL:IFCOND(IntermediateResults)
#include <stdio.h>
// PVSCL:ENDCOND

static int factorialOf(int x);

int factorial(int n)
{
  return factorialOf(n);
}

static int factorialOf(int x)
{
  int result;

  if (x <= 1) {
    result = 1;
  } else {
    result = x * factorialOf(x-1);
  }

  // PVSCL:IFCOND(IntermediateResults)
  // PVSCL:IFCOND(IntermediateResults->Format='number')
  printf("%d\n", result);
  // PVSCL:ELSEIFCOND(IntermediateResults->Format='sentence')
  printf("Factorial of %d is %d.\n", x, result);
  // PVSCL:ELSECOND
  printf("%d ! = %d\n", x, result);
  // PVSCL:ENDCOND
  // PVSCL:ENDCOND

  return result;
}
```

フィーチャ IntermediateResults が選択されている時この条件が true となり、以下 PVSCL:ENDCOND までの部分の内容が結果ファイルに出力されます

条件はネストできます

上図で printf …の部分はオプション機能で、階乗の途中結果を次々（たとえば 3!の場合、2!、1!、と順次）表示するもので、フィーチャ **Number**、**Formula**、**Sentence** に従って、どの書式で出力するかを切り替えます。この選択動作は「条件」をコメントとしてコード内に挿入することで実現します。

このコメント内の PVSCL:xxx が「条件」マクロ指令で、ソースファイルを条件付きテキスト（*ps:pvscltext*）型のエレメントとしてファミリーモデルに設定すると、モデル変換実行時にファイル内の「条件」に応じて内容が無効となる部分を削除し、有効部分だけを出力してソースコードファイルが生成されます。（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）

「条件」は pvscl 式で表現され、「比較」や「計算」を含めることもできます。これによって、たとえば **IntermediateResults** の属性 Format などフィーチャがもつ値に応じて、プログラムの計算結果を様々な形式（本例では、文章 / 式 / 数値だけ）で出力します。

3.2. ドキュメントファイル

本例では、プログラムの概略仕様を説明するドキュメントとして HTML ファイルを使用します。上記の他のソースファイルと同様に以下の内容で、input フォルダに `document.html` ファイルを作成します。

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8"></meta>
  <title>Factorial Calculation Program</title>
</head>

<body>
  <h1>Factorial Calculation Program</h1>
  <p>Program version <pv:eval>Version->Version</pv:eval></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 pv:condition="IntermediateResults">
  Intermediate Results
  </h2>
  <p pv:condition="IntermediateResults">
    <p pv:condition="(IntermediateResults->Format='number')">
    Intermediate results are shown as simple numbers.
    </p>
    <p pv:condition="(IntermediateResults->Format='formula')">
    Intermediate results are shown as formulas, e.g. 3! = 6.
    </p>
    <p pv:condition="(IntermediateResults->Format='sentence')">
    Intermediate results are shown as sentences, e.g. Factorial of 3 is 6.
    </p>
  </p>
</body>

</html>
```

`document.html` ファイルは「条件」用の属性を記述したタグをもち、条件付き XML (*ps:pvsclxml*) 型のソースエレメントとしてファミリーモデルに設定します。

`factorial.c` の場合と同様に、フィーチャ **IntermediateResults** の属性 Format の値に応じてこのファイル内の部分が選択されます。これは HTML 文書のタグとして特別な属性（上図の `pv:condition`）を使い、そこに pvscl 式で記述した「条件」を設定することで実現します。

モデル変換中に *ps:pvsclxml* 型のソースエレメントに対し、この特別なタグで規定される部分において「条件」が true の場合に、対応する部分が結果のドキュメントに出力されます。

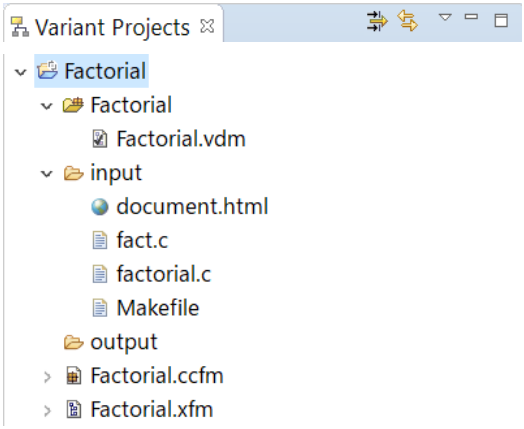
3.3. ビルド用ファイル

アプリケーションのビルドは、本例では UNIX 系プラットフォームで動作する GNU make を使用します。以下の内容で、プロジェクトの input フォルダに Makefile という名前でファイルを作成します。

アプリケーションのバリアント用のファイルが output フォルダ内のそれぞれのフォルダに生成されて、そこにある .c ファイルを使用してコンパイル・リンクするものとしています。(各バリアント用の Makefile もそれぞれ出力されます)

```
PROGRAM = fact
OBJS = fact.o factorial.o
CC = gcc

$(PROGRAM): $(OBJS)
	$(CC) $(OBJS) -o $(PROGRAM) -I.
```

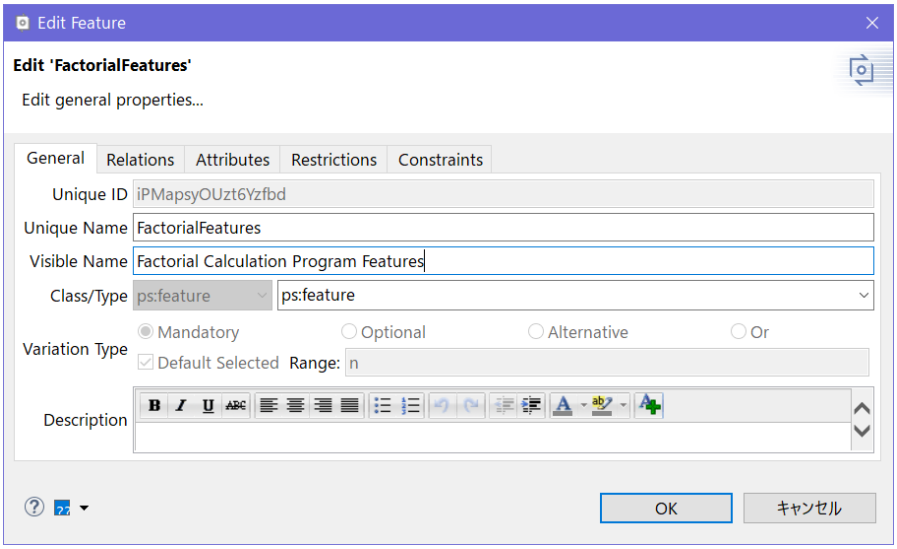
最終的に、この階乗計算アプリケーションの pure::variants プロジェクトの構造は、下図のようになります。



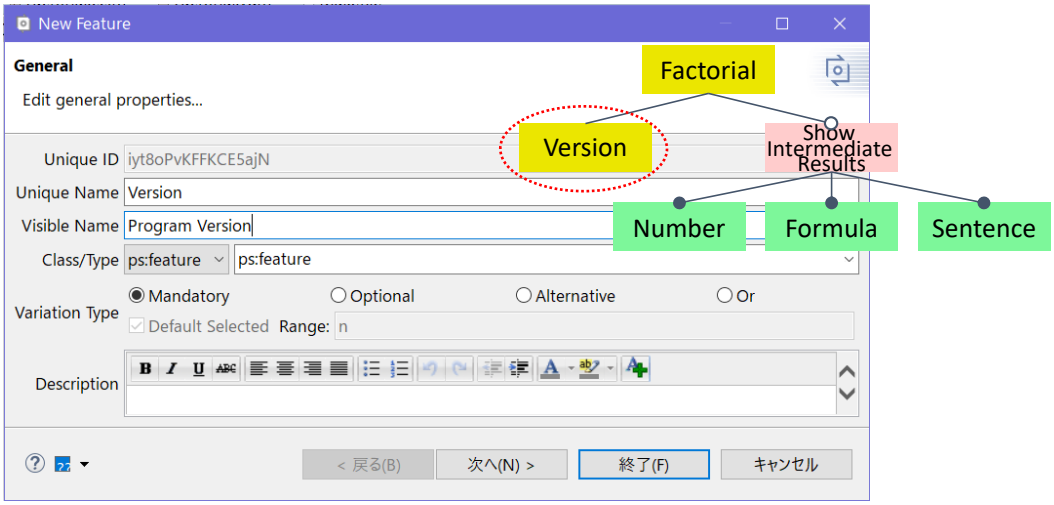
4. フィーチャモデルの設定

p.4 のモデルに従って、構成されるアプリケーションの機能に対する pure::variants の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、ルートフィーチャ（**FactorialFeatures**）をダブルクリックして Visible Name を「Factorial Calculation Program Features」とし、OK します。



Factorial Calculation Program Features を右クリックし、コンテキストメニューから New > Feature を選択します。New Feature ウィザードで、Unique Name に「Version」、Visible Name に「Program Version」を入力し、Variation Type は Mandatory をチェックします。（バリエーションのタイプは、先述のフィーチャモデルで決定しているものです）

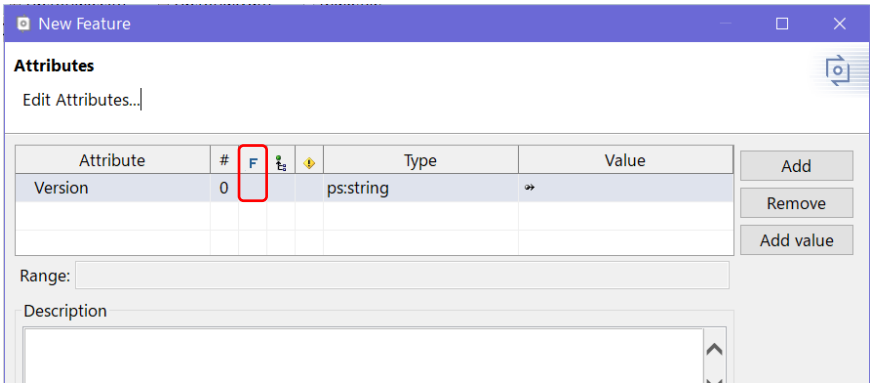


次へ のクリックで 2 画面進めると Attributes ウィザードが開きます[5]。Add し、以下のような値を持たない Attribute「Version」を追加します（この値はモデル変換時に設定するので、3 列目 F [6]のチェックを外しておきます）。

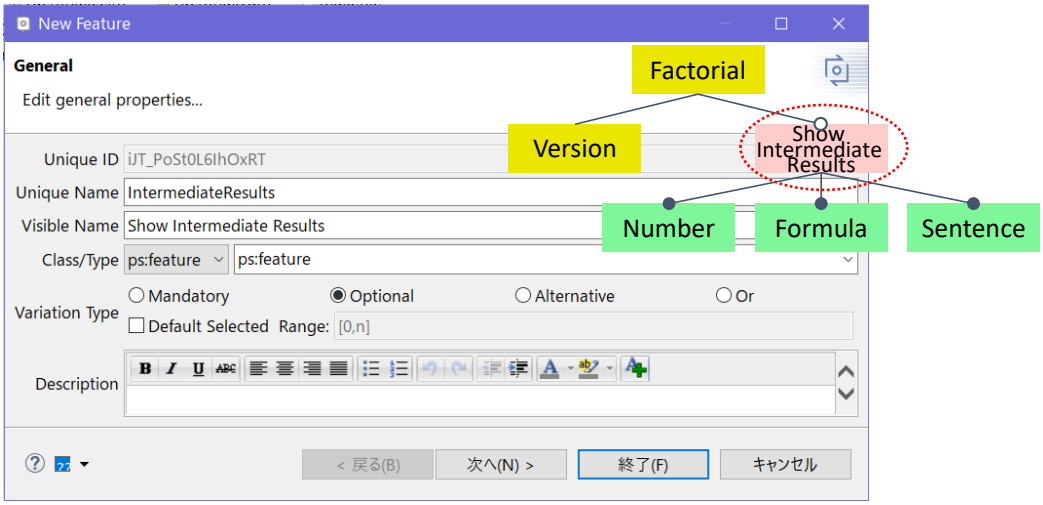
[5] フィーチャウィザードを終了後、ダブルクリックして Attributes タブからでも設定できます。

[6] このアイコンは、当該属性が「固定」や「読み取り」であることを示します。

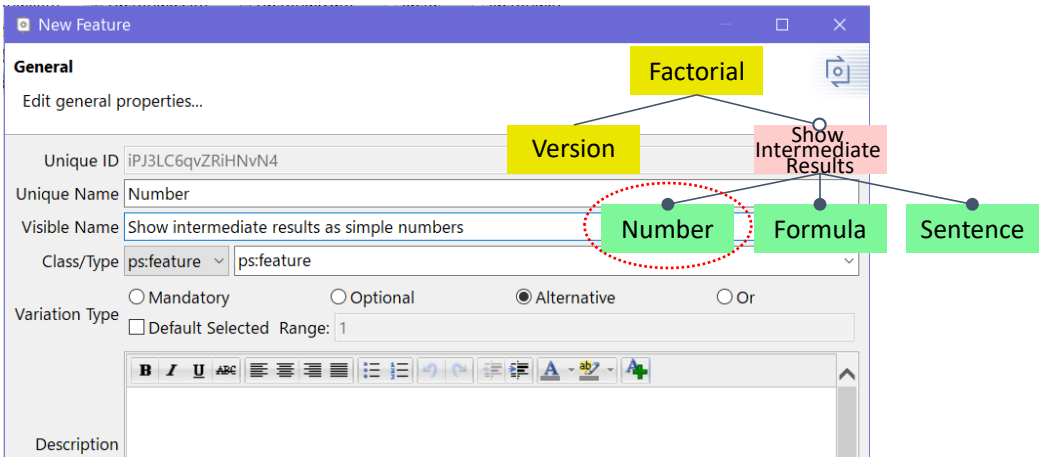
この Version 属性によってアプリケーションのバージョン番号を設定します。



ルートの子として Unique Name が「IntermediateResults」、Visible Name が「Show Intermediate Results」、Variation Type が Optional であるフィーチャを作成します。

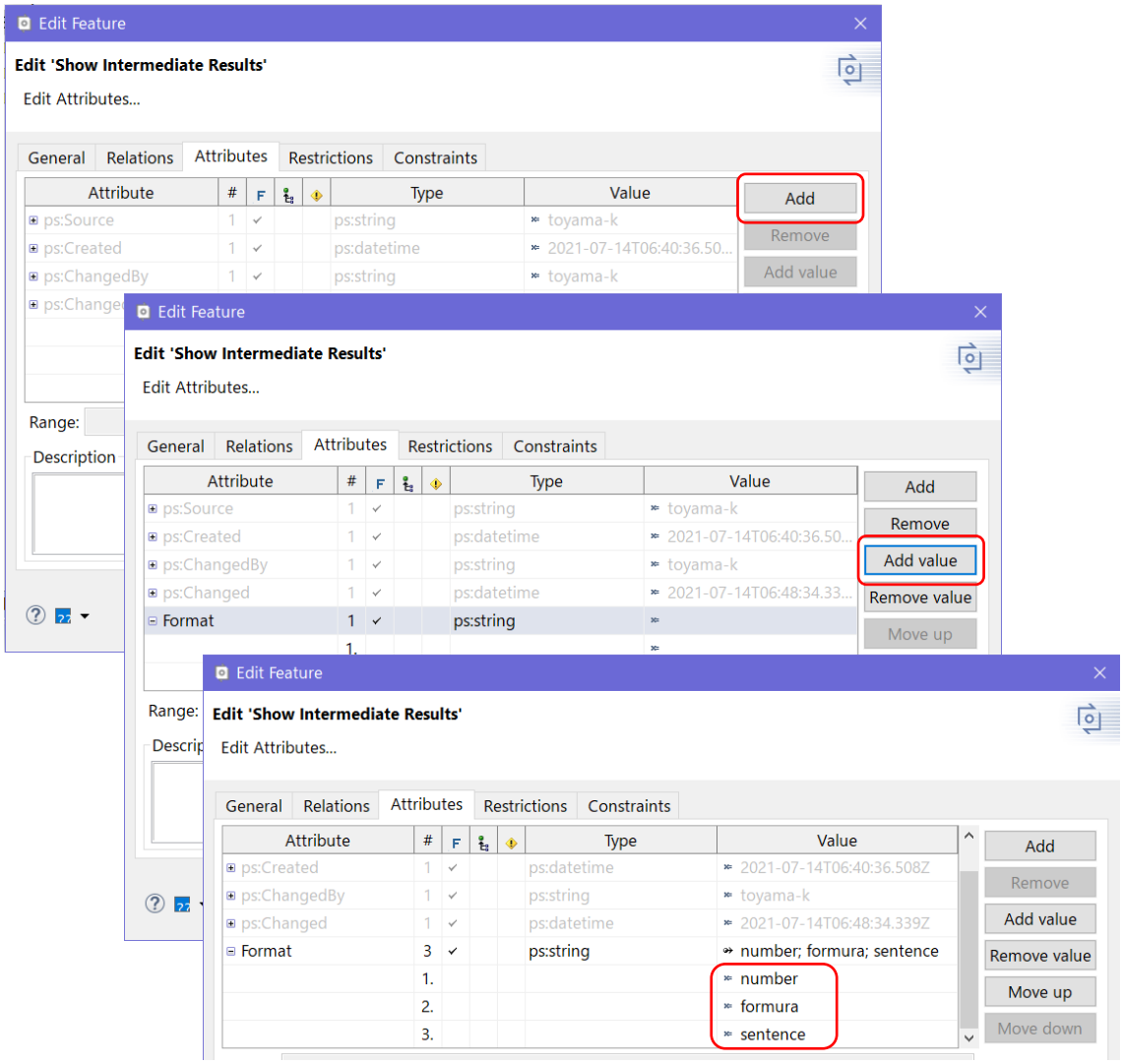


フィーチャ **Show Intermediate Results** の子として Variation Type が Alternative であるフィーチャを 3 つ作成します。1 つ目は Unique Name が「Number」、Visible Name が「Show intermediate results as simple numbers」のものです。



同様にして、2 つ目：Unique Name が「Formula」、Visible Name が「Show a formula for each intermediate result」である Alternative フィーチャ、3 つ目：Unique Name が「Sentence」、Visible Name が「Show a sentence for each intermediate result」である Alternative フィーチャを作成します。

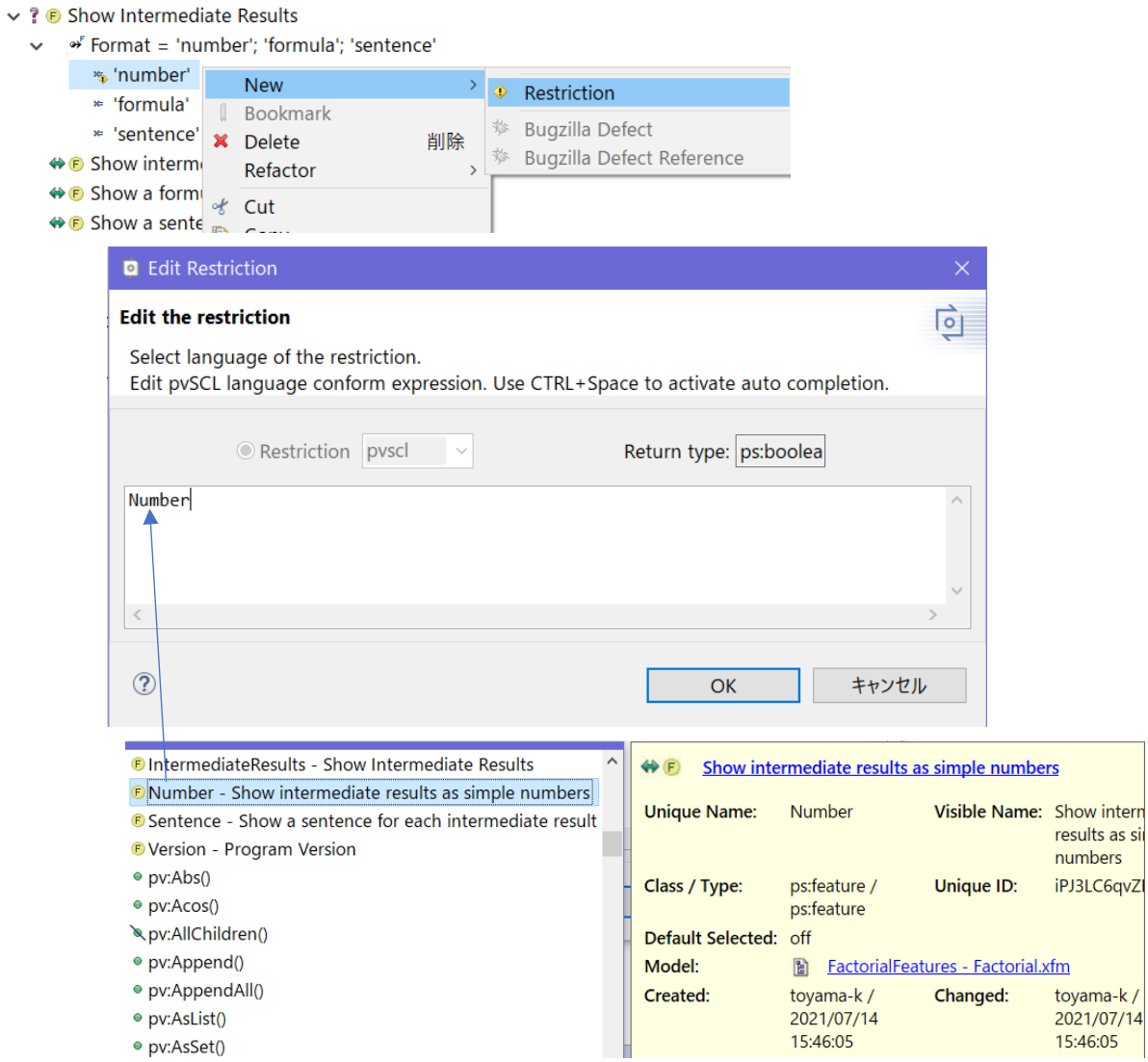
フィーチャ **Show Intermediate Results** をダブルクリックし、下図のようにウィザードの Attributes タグで Add して「Format」という名前の属性を追加し、それに対して Add value を押し、値「number」「formula」「sentence」を Value 欄に追加します。



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のフォーマットで出力するかを設定するために使用します[7]。

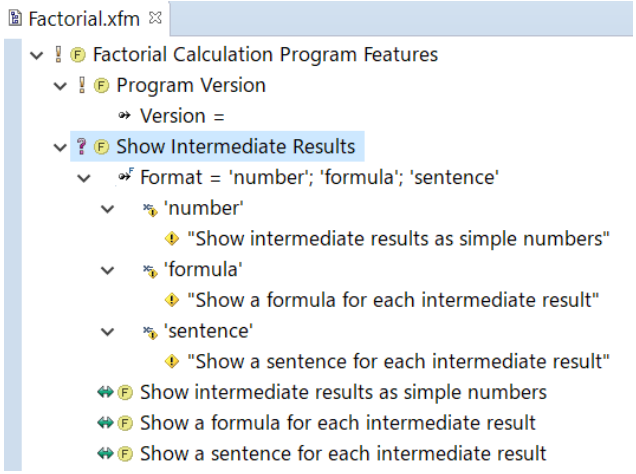
そのため、フィーチャモデル上で Format の Value である number を右クリックしコンテキストメニューから New > Restriction を選択して pvscl 制約を追加します。子ノードの Unique Name である Number を直接入力するか、CTRL+space で候補が出ますので、Number に対する行をダブルクリックして選択します。

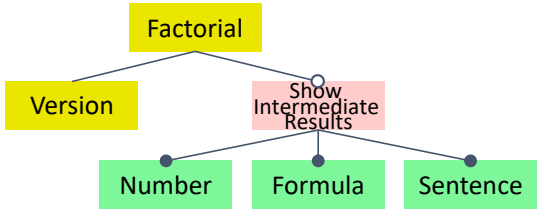
[7] Show Intermediate Results の子フィーチャを使用しても特定できますが、ここではそれら子フィーチャを属性として結びつけることで処理しています。



この制約は、フィーチャ **Number** が選択された場合に属性 Format の値が number になることを示します。同様に、formula と sentence に対してもそれぞれ Formula と Sentence となる制約を設定します。

完成したフィーチャモデルは以下のようになります。



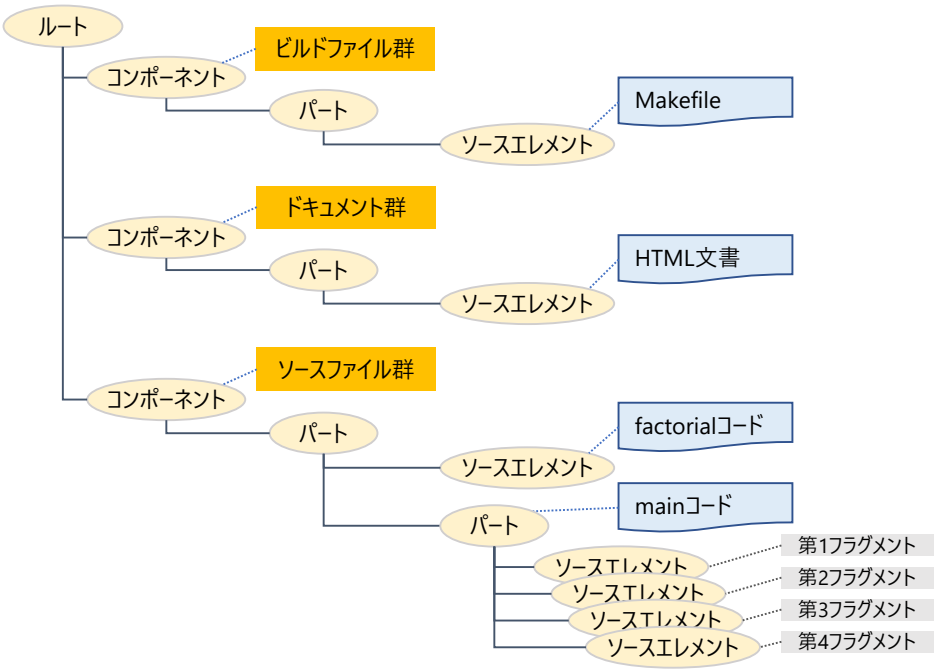


5. ファミリーモデルの設定

アプリケーションのコンポーネント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エレメント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変換を定義することにも使用されます。

この変換は前述したように、ソースエレメントの型（*ps:pvscltext*, *ps:pvsclxml*, *ps:fragment*）に対応して、指定された条件によって一部を削除したり、テキストを追加したりすることで実施されます。

アプリケーションのコンポーネントは、input フォルダで異なったファイルタイプであるビルドファイル、ドキュメントファイル、そしてソースファイルの 3 グループに分類され、ファミリーモデルは以下のような構成となります。



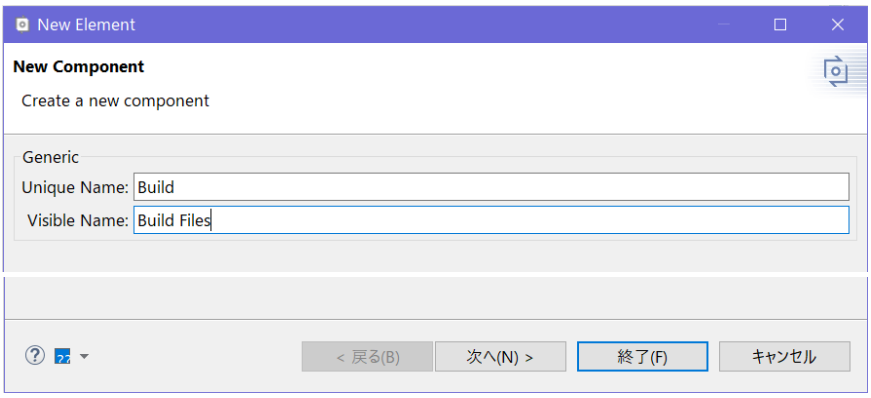
5.1. ビルドファイル

本例で作成するビルドファイルは、モデル変換によってファイルの内容が変更されず、アプリケーションのファイルを含む output フォルダに単純にコピーされます[8]。したがって *ps:file* 型のソースエレメントを使用します。

`Factorial.ccfm` のルートフィーチャ（FactorialFamily）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」を入力して 終了 し、**Build Files** コンポーネントを作成します[9]。

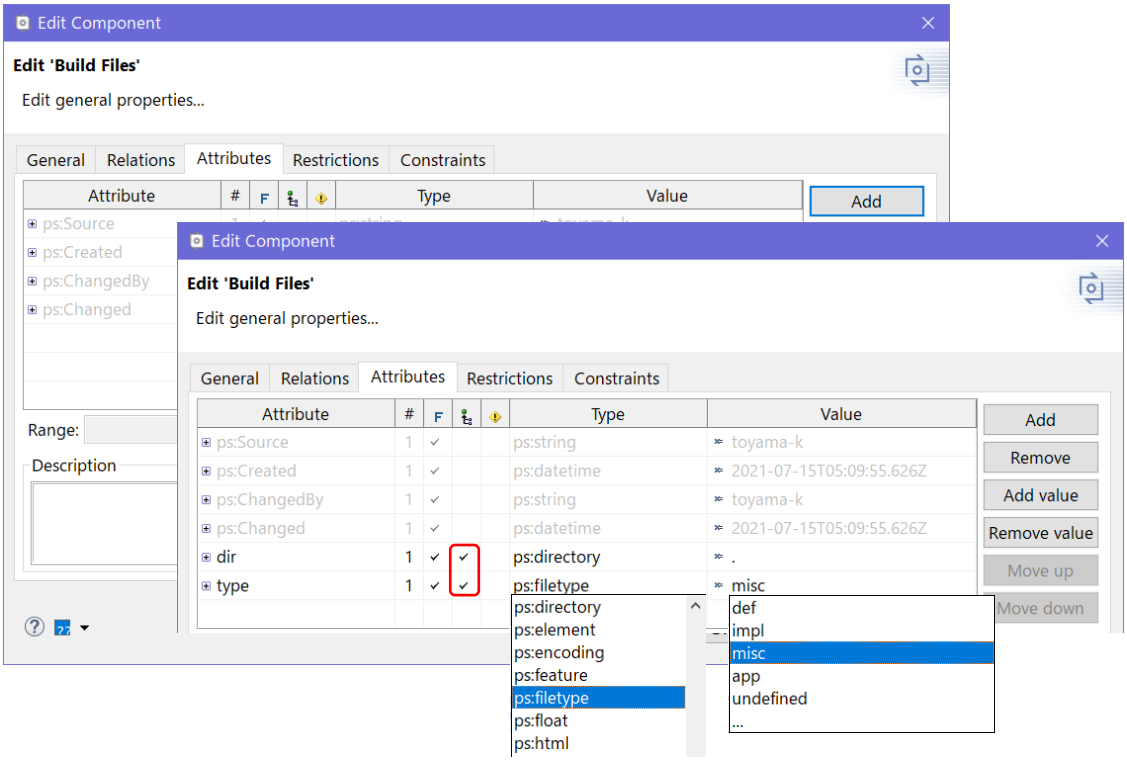
[8] バリアントごとでオプション設定などビルドファイルの内容を変更する場合は、本資料のテキストやソースファイルの例を参考にして Makefile を作ることができます。資料「ソースコード、フラグのバリアント管理」も参照願います。

[9] ここでの説明は複数のビルドファイルがあることを想定しているものですが、本例では一つだけ作成します。



Build Files コンポーネントをダブルクリックし、ウィザードで Attributes タブを選択します。Add して[10] Attribute に「dir」と入力し、Type として *ps:directory* を選択、Value には「`.`」を設定し、この属性に *inheritable* オプションを設定します (4 列目の  をクリックし ✓ を入れます)。これにより、すべての子エレメントに値が継承され、子エレメントでは設定せずに dir 属性を同じ値で使用できます。

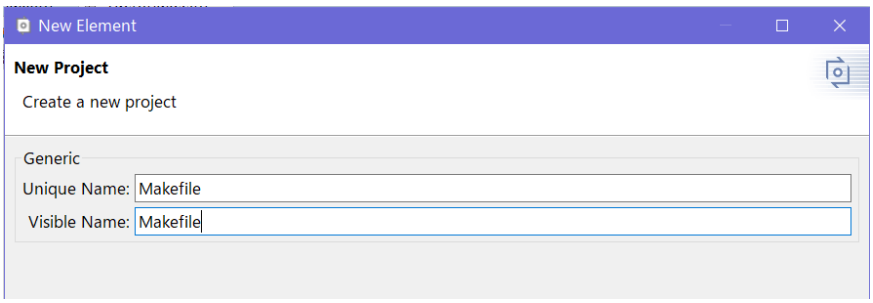
同様に Add して Attribute が「type」、Type が *ps:filetype* の別の属性を追加します。Value として「misc」(`.c .cc .cpp .h` 以外のファイルであることを表します) を設定し、*inheritable* オプションを設定し、OK します。



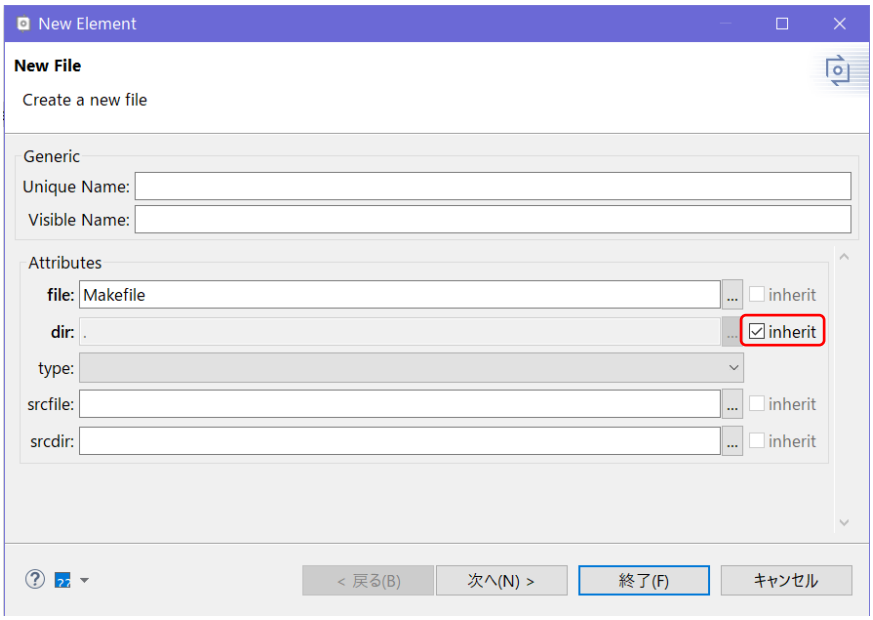
ビルドファイルは既定の *ps:project* 型 (*ps:class* や *ps:flag* など他のどの型にも相当しないもの) のパートとして作成します。

[10] 右クリックし、コンテキストメニューから New > Attribute を選択すると、Add した状態になります。

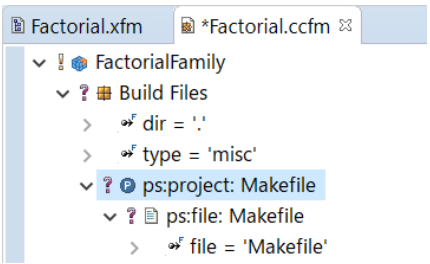
Build Files コンポーネントで右クリックしコンテキストメニューから New > Project を選択します。Unique Name と Visible Name 両方に「Makefile」と入力し、終了 します。



作成したソースエレメント **Makefile** で右クリックしコンテキストメニューから New > File を選択します。ウィザードで Attributes の **file** に「Makefile」と入力し、**dir** の inherit をチェックします。



終了 で Makefile を表すソースエレメントが作成され、ファミリーモデルは下図となります。

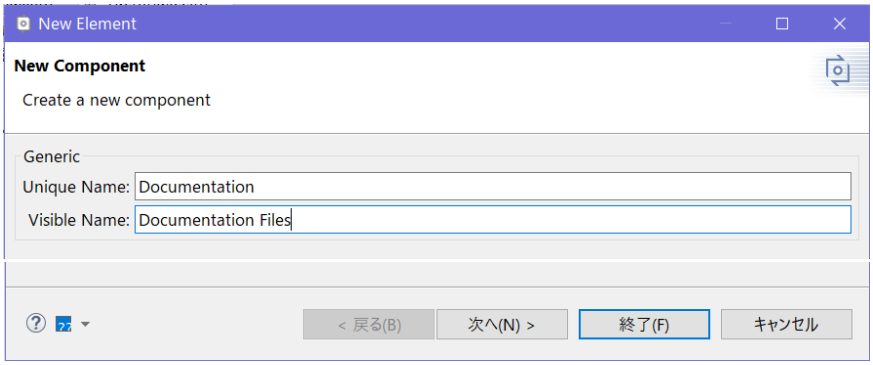


5.2. ドキュメントファイル

ドキュメントファイル `document.html` をファミリーモデルにマップします。モデル変換を行うと、このファイルから選択されたフィーチャに応じた部分のテキストだけを含むドキュメントファイルが生成されて output フォルダに置かれます。これは先述した HTML 文書用のタグに出力のための「条件」を指定する属性を記述することで実現しています。

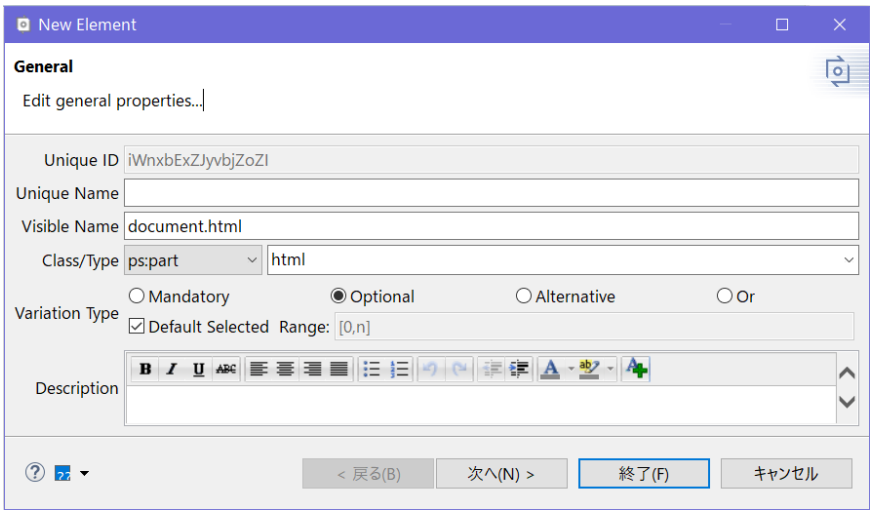
これらの「条件」は *ps:pvsclxml* ソースエレメントで評価され、その「条件」の評価結果が false であれば、対応するタグで定義される部分がドキュメントから除外されます。

ファミリーモデルのルートにコンポーネントを追加し、Unique Name に「Documentation」、Visible Name に「Documentation Files」と入力します。



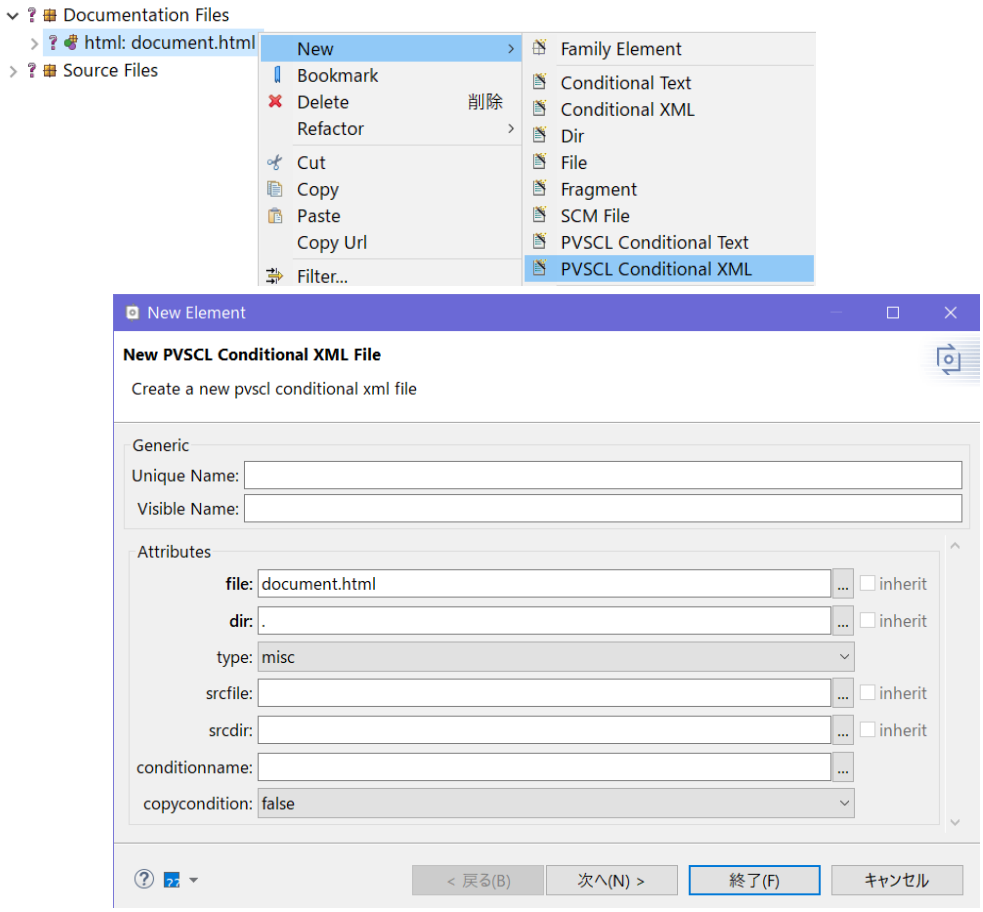
終了 で **Documentation Files** コンポーネントが作成され、使用するドキュメントをまとめて管理できるようになります[11]。

Documentation Files の子として Visible Name が「document.html」、Type が「html」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 を選択します。



[11] ビルドファイル同様、以下の説明は複数のドキュメントファイルがあることを想定しているものですが、本例では一つだけ作成します。

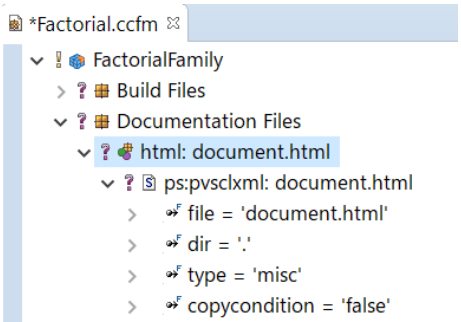
作成したファミリーエレメント（document.html）の子として、PVSCL 条件付き XML エレメント（*ps:pvsclxml* 型のソースエレメント）を追加します。ファミリーモデルの **html: document.html** で右クリックしコンテキストメニューから New > PVSCL Conditional XML でウィザードを開きます。



Attributes の **file** に「document.html」、**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 HTML ドキュメントのタグにある特別な属性は、その「条件」が評価された後に削除されます。それらの属性は HTML で有効ではないので削除することを指定しています。

ps:pvsclxml 型のソースエレメントに関する詳細は pure::variants User's Guide の 9.5.6 項を参照してください。

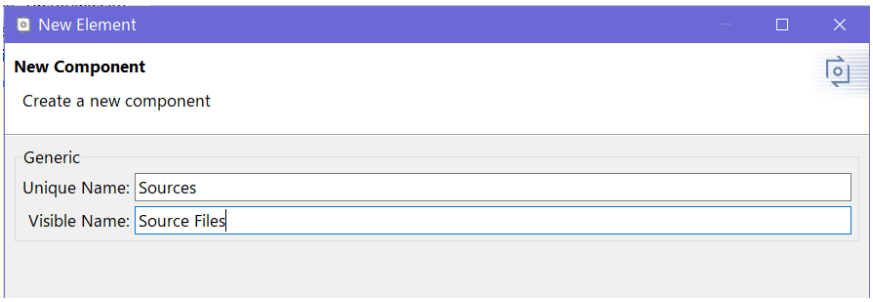
ファミリーモデルは、以下のようになります。



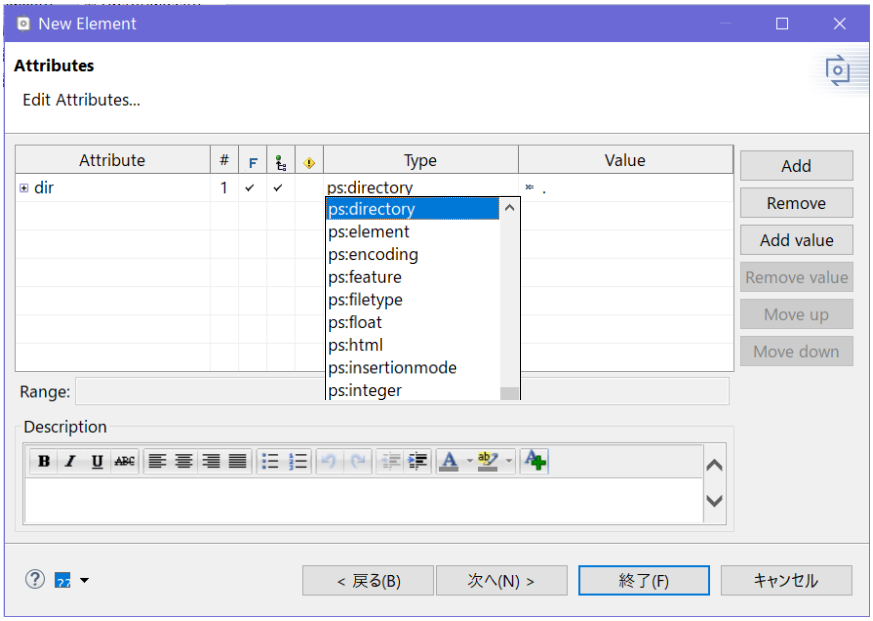
5.3. ソースファイル

ソースファイル（`fact.c` と `factorial.c`）をファミリーモデルにマップします。

まず、ファミリーモデルのルートに Unique Name が「Sources」、Visible Name が「Source Files」であるコンポーネントを作成します。



次へ を 3 回で Attributes 設定ウィザードに進み、Add して Attribute に「dir」を入力し、Type として *ps:directory* を選択、Value には「`.`」を設定し、*inheritable* オプションを設定します（4 列目の をクリックし ✓ を入れます）[12]。



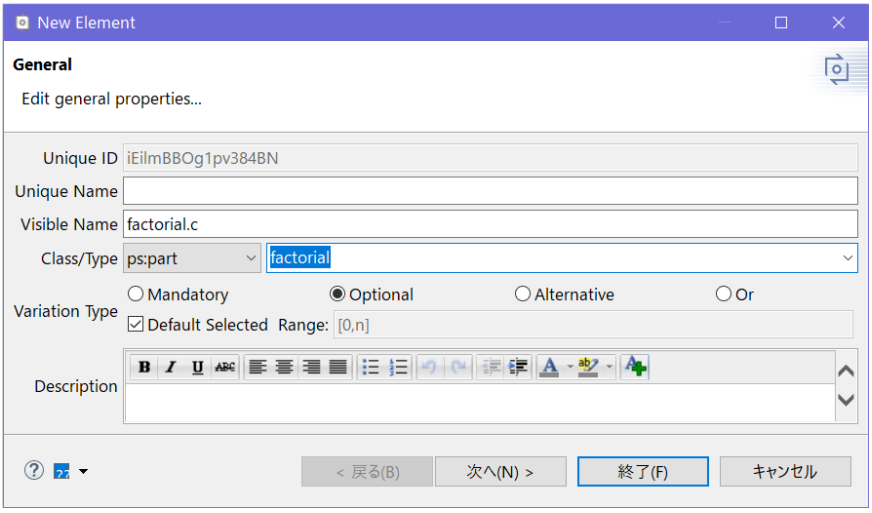
この **Source Files** コンポーネントの子として `factorial.c` と `fact.c` を作成します。

`[factorial.c]`

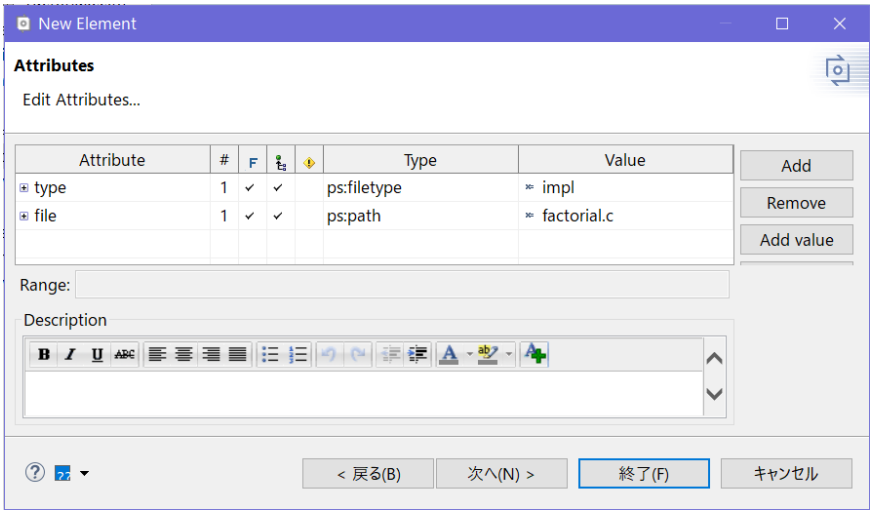
`factorial.c` は **IntermediateResults** の値に応じて中間結果を出力するため、コードに対するガードとなる「条件」マクロ指令が含まれた *ps:pvscltext* 型のソースエレメントに対して評価されます。

Source Files の子として、Type が「factorial」、Visible Name が「factorial.c」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）

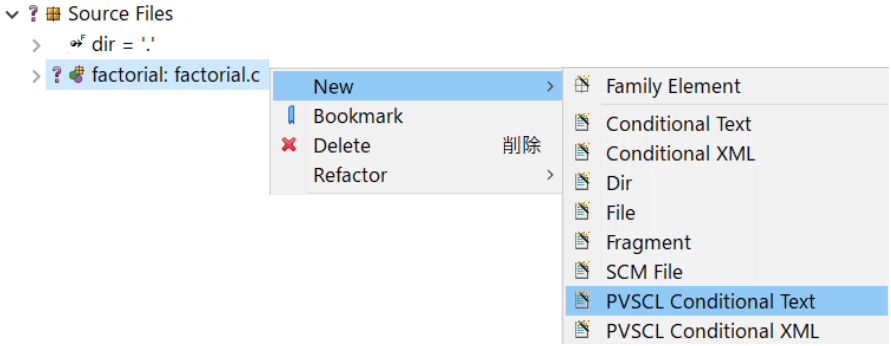
[12] p.14 の「5.1 ビルドファイル」の設定と同じ手順です。



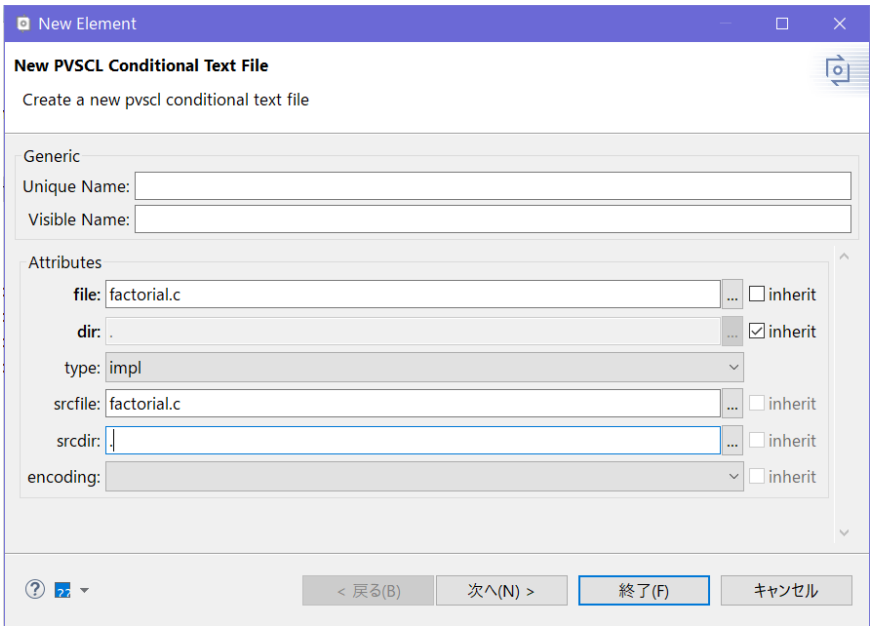
次へ を 2 回で Attributes ウィザードに進み Add して、Type が *ps:filetype*、Value が「impl」である Attribute「type」と Type が *ps:path*、Value が「factorial.c」である Attribute「file」を追加し、それぞれ *inheritable* オプションを設定して[13]、終了 します。



factorial:factorial.c の子として PVSCL 条件付きテキスト（*ps:pvscltext* 型のソースエレメント）を作成します。（右クリックしコンテキストメニューから New > PVSCL Conditional Text を選択します）



[13] ビルドファイルやドキュメント同様、複数のファイルがあることを想定して、設定する属性を子に継承することを指定しているものですが、本例ではファイルを一つだけ作成します。

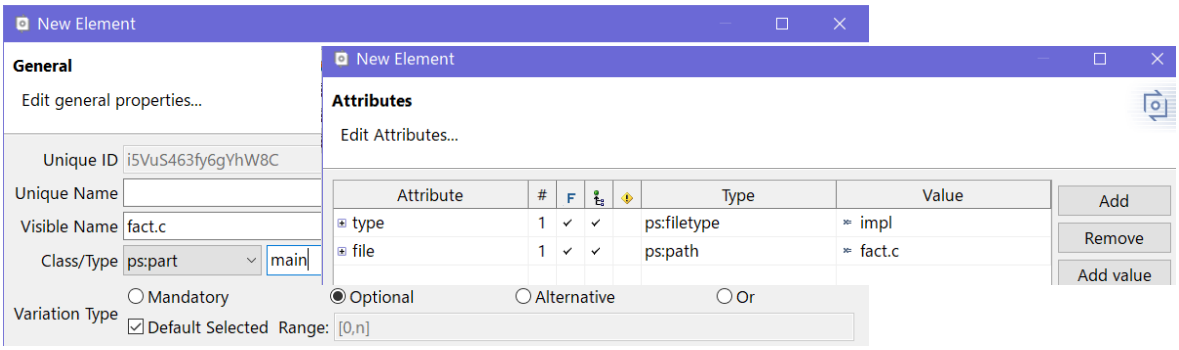


ウィザードで **file** を「factorial.c」、type を「impl」と設定します。**dir** は inherit をチェックし、srcfile は「factorial.c」、srcdir は「.」として 終了 します。

ps:pvscltext 型のソースエレメントに関する詳細は pure::variants User's Guide の 9.5.7 項を参照してください。

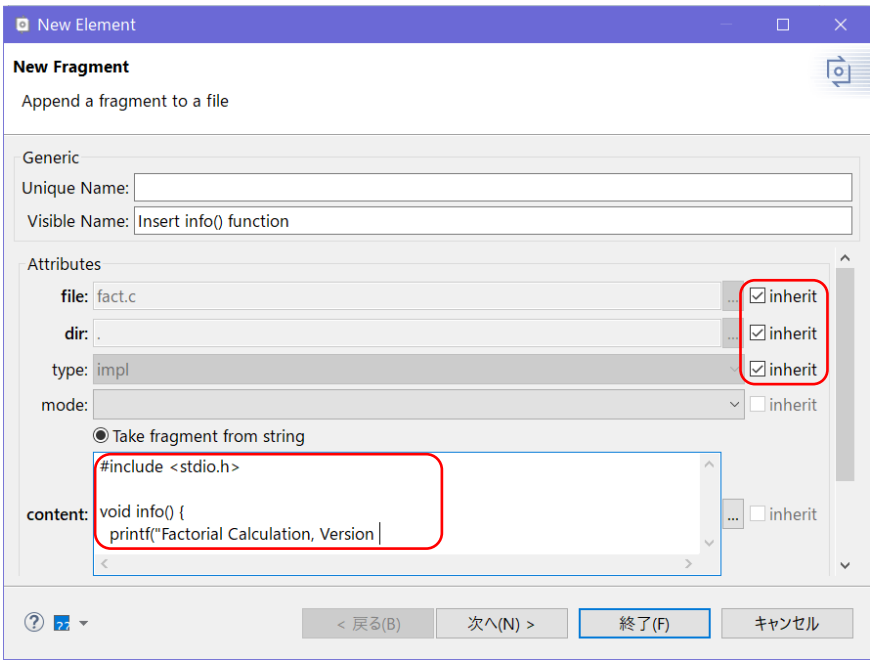
[fact.c]

`fact.c` は *ps:fragment* 型のソースエレメント 4 つのパートによって構成します。まず、全パートをまとめて表現するファミリーエレメントを `fact.c` として、`factorial.c` と同じ手順で作成します。type は「main」、Visible Name は「fact.c」とします。



inheritable オプションの設定は、ここで定義するファイルに対して、名前が `fact.c` で C や C++のソースコードであるという属性を子に継承することを指定しています。

以下、**main:fact.c** の子として 4 つのフラグメントを順に作成します。第 1 フラグメントは `info()` の定義部分で、*ps:fragment* 型のソースエレメントを作成（コンテキストメニューから New > Fragment を選択します）し、Visible Name を「Insert info() function」とします。Attributes の **file**、**dir**、type の inherit をチェックします。**content** に下図のように入力して 終了 します。

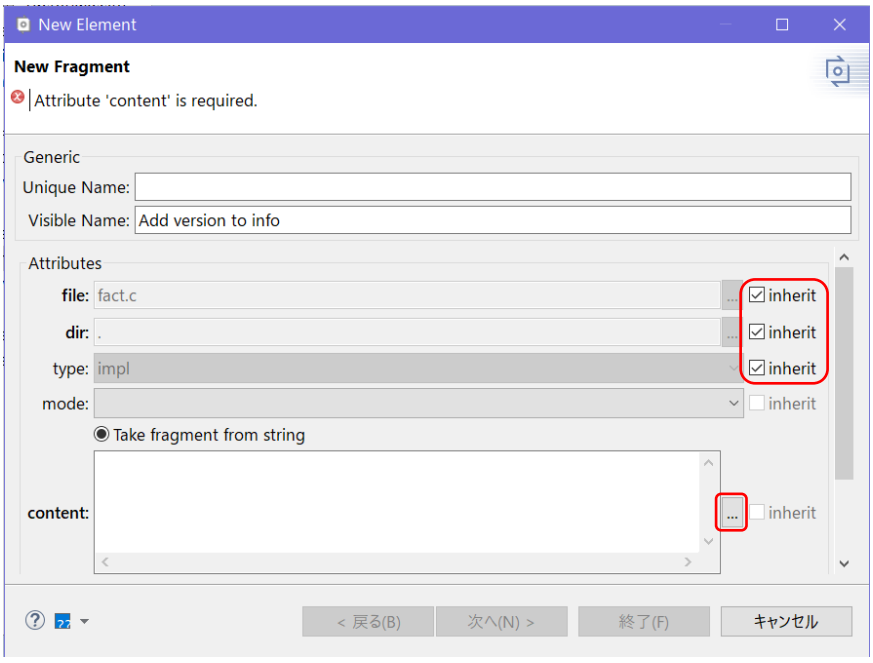


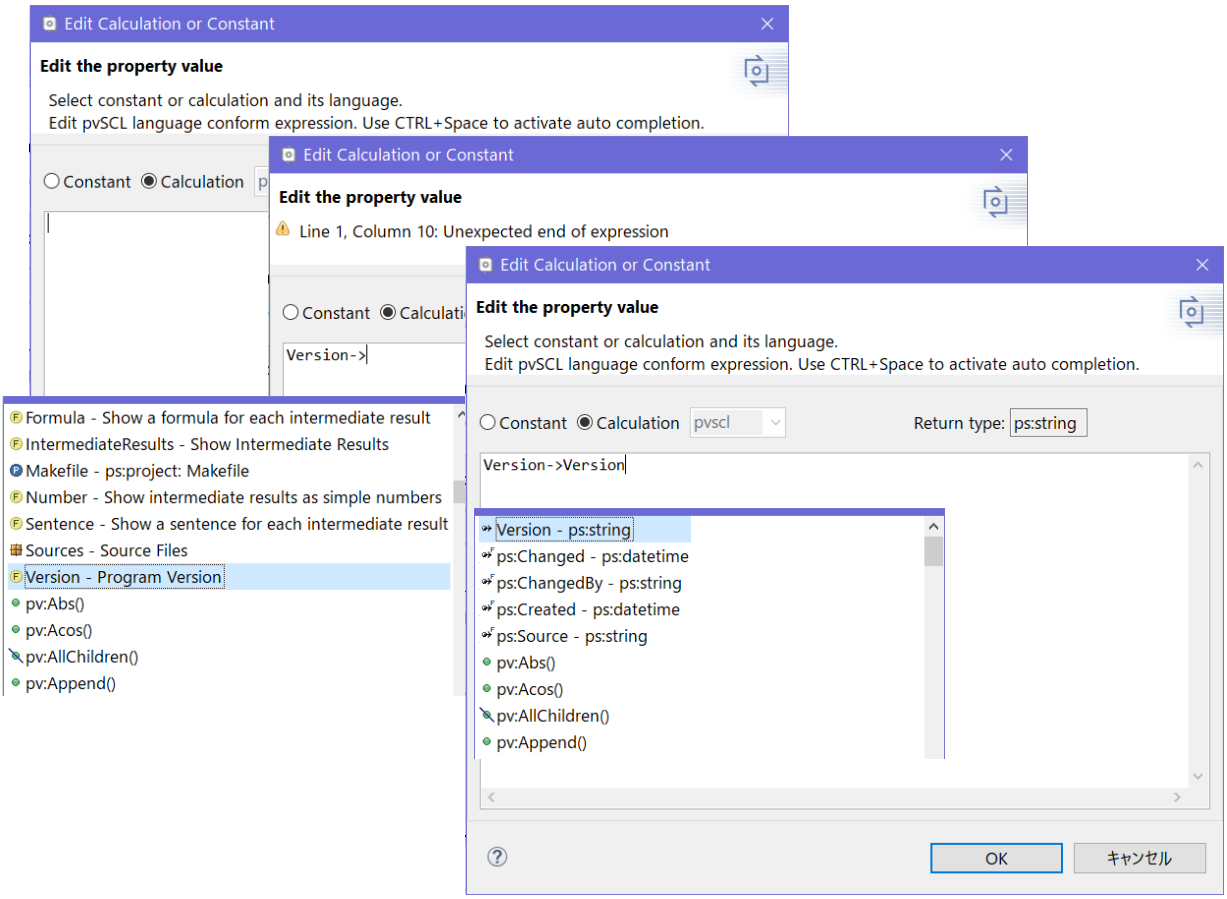
これで、`printf()`で出力する文字列の途中までを作成しています。**content** の内容がモデル変換の出力フォルダ（output）内に生成する `fact.c` に追加されます。

この後、フラグメント 2 つで `info()` の定義を追加して行きます。それらがモデル変換時に `fact.c` 内に追加され、本アプリケーションの名前とバージョンを出力するソースコードとなります。

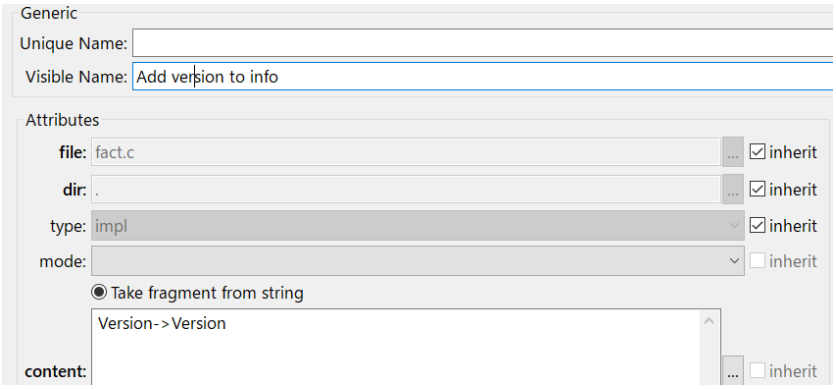
`fact.c` の第 2 フラグメントとして *ps:fragment* 型のソースエレメントを作成します。ウィザードで Visible Name を「Add version to info」、Attributes の **file**、**dir**、type で inherit をチェックします。

ここで、フィーチャの属性を参照するコードを生成するため、**content** フィールドの右の ... をクリックします。





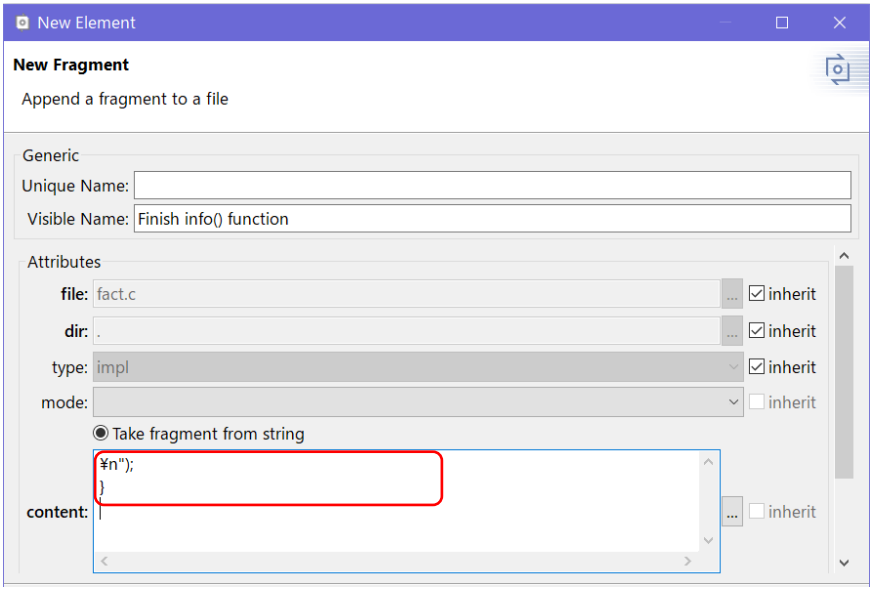
pvscl のエディット画面が開きますので、pvscl 式を入力します。CTRL+space で候補が出ますので、ダブルクリックして選択します。最初に CTRL+space で候補を出して「Version – Program Version」を選択し、その直後に「->」を入力すると Version の候補が出るので「Version – ps:string」を選択します。OK して戻り、終了 します。



このフラグメントは「計算」(Calculation のラジオボタンが ON) であり、フィーチャモデルの Version (Visible name は **Program Version**) の属性 Version の値を取得して出力すべきバージョン番号としています。この値は文字列です。

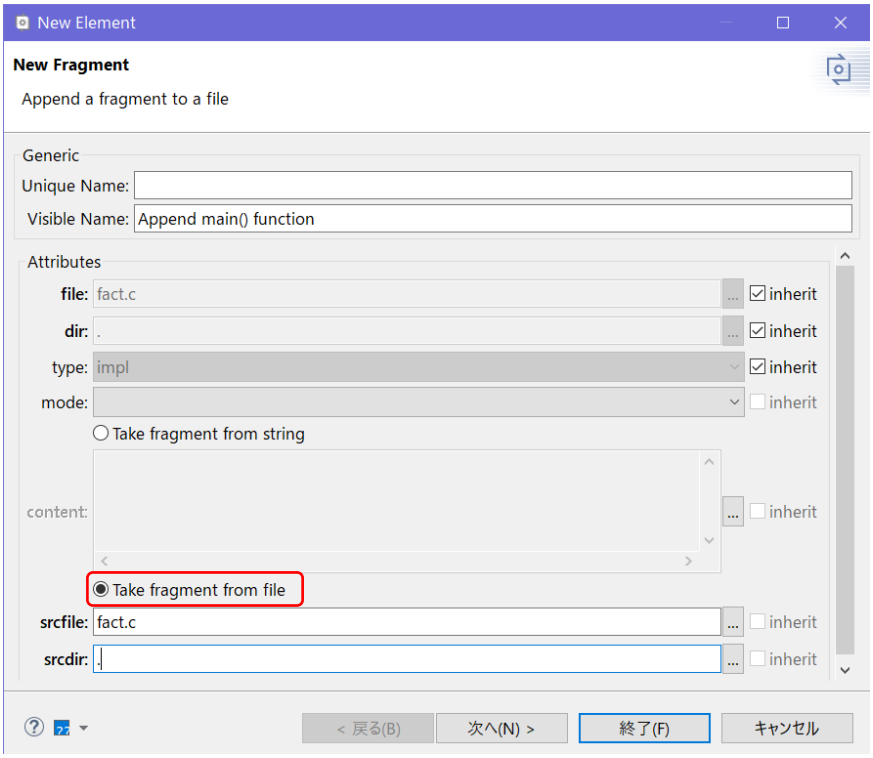
なお、このパート以外では通常のテキストをそのまま出力します。(Constant のラジオボタンが ON となります)

第 3 フラグメントは `info()` の残りの部分です。*ps:fragment* 型のソースエレメントを作成し、Visible Name を「Finish info() function」と入力します。Attributes の **file**、**dir**、type で inherit をチェックし、**content** フィールドに下図のようにテキストを入力します。

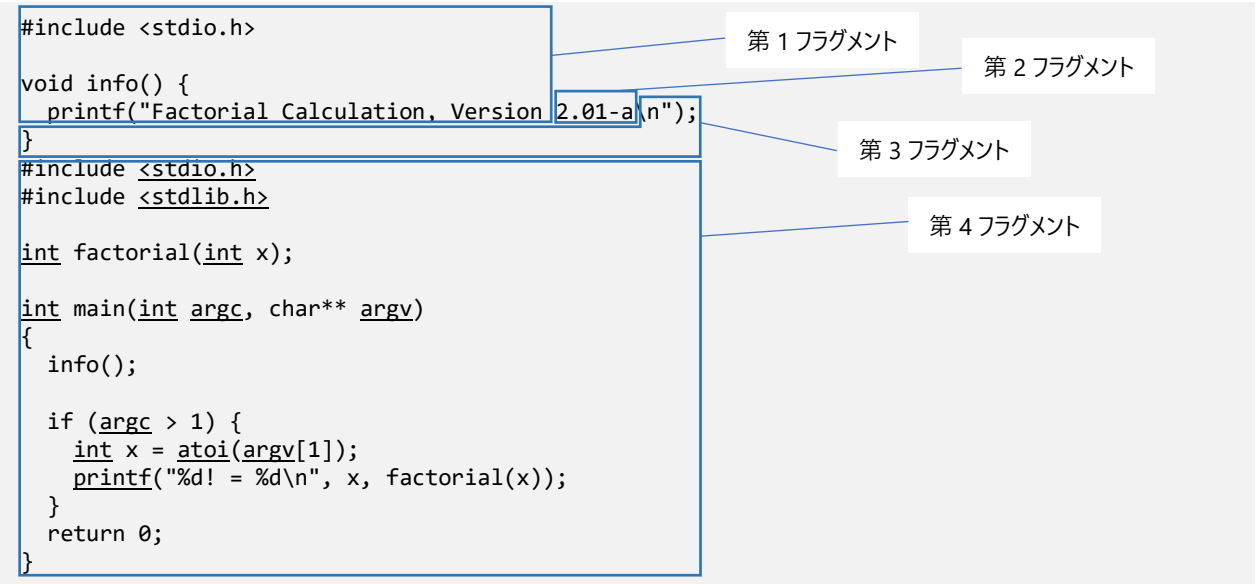


最後の第 4 フラグメントを *ps:fragment* 型のソースエレメントで作成します。Visible Name を「Append main() function」とし、Attributes の **file**、**dir**、type で inherit をチェックします。

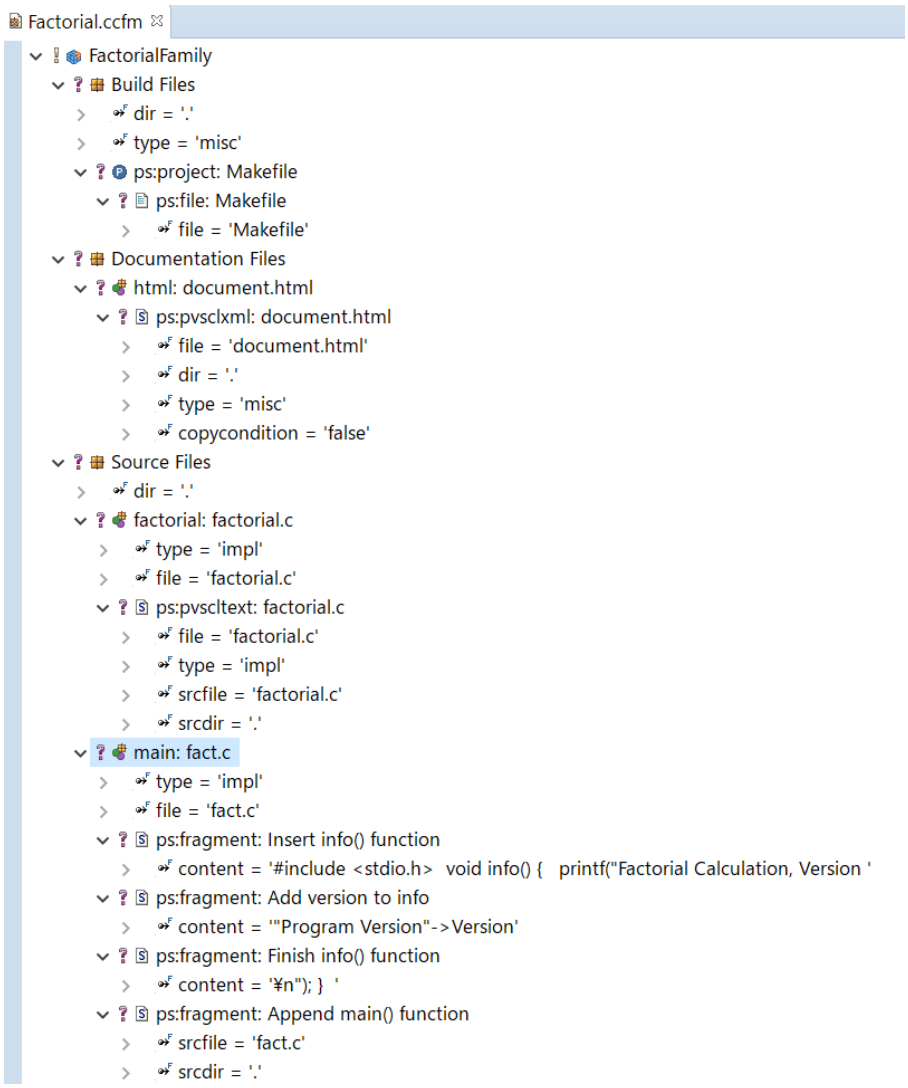
他のフラグメントと違い、このフラグメントのテキストは、**content** に記述するのではなく、元の `fact.c` の内容をそのままファイルから入力します。そのため Take fragment from file をチェックし、srcfile に「fact.c」、srcdir に「.」を入力し、終了します。



生成されるソースコードのイメージは下図のようになります。



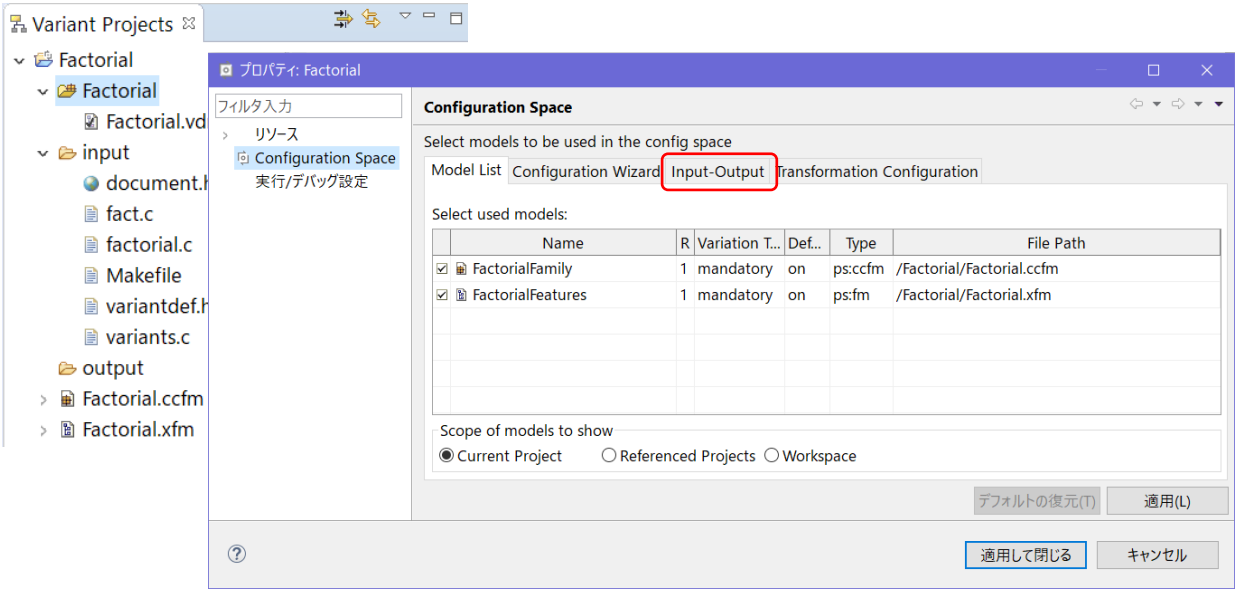
最終的にファミリーモデルは、以下のようになります。



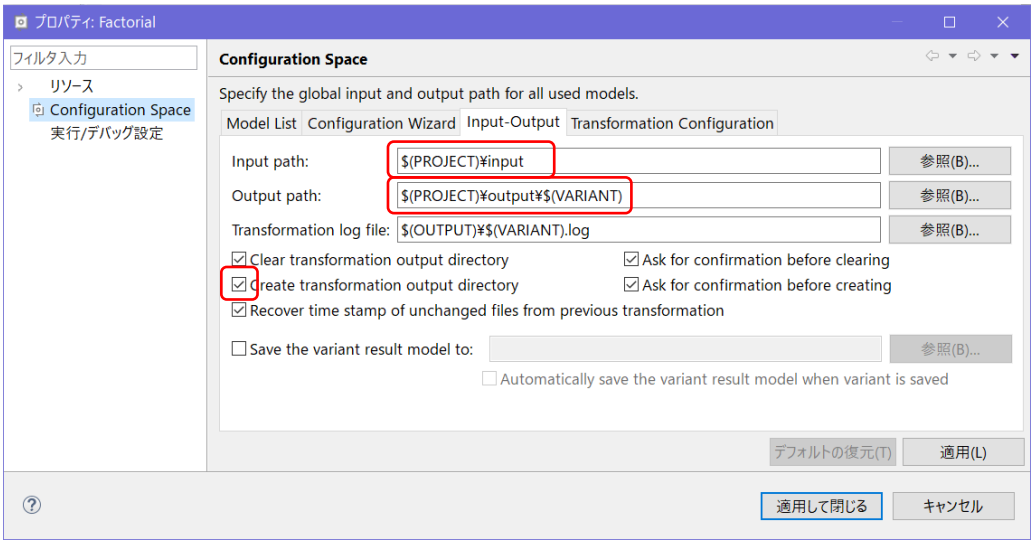
6. モデル変換の設定

モデル変換を行うため、コンフィグレーションオプションを変更します。Variant Project ビューで Configuration Space (Factorial) を選択し、右クリックでコンテキストメニューから プロパティ を選択します。

ウィザードで Configuration Space に切り替え、このテキスト変換を実施する際の入出力となるファイルの場所を設定します。

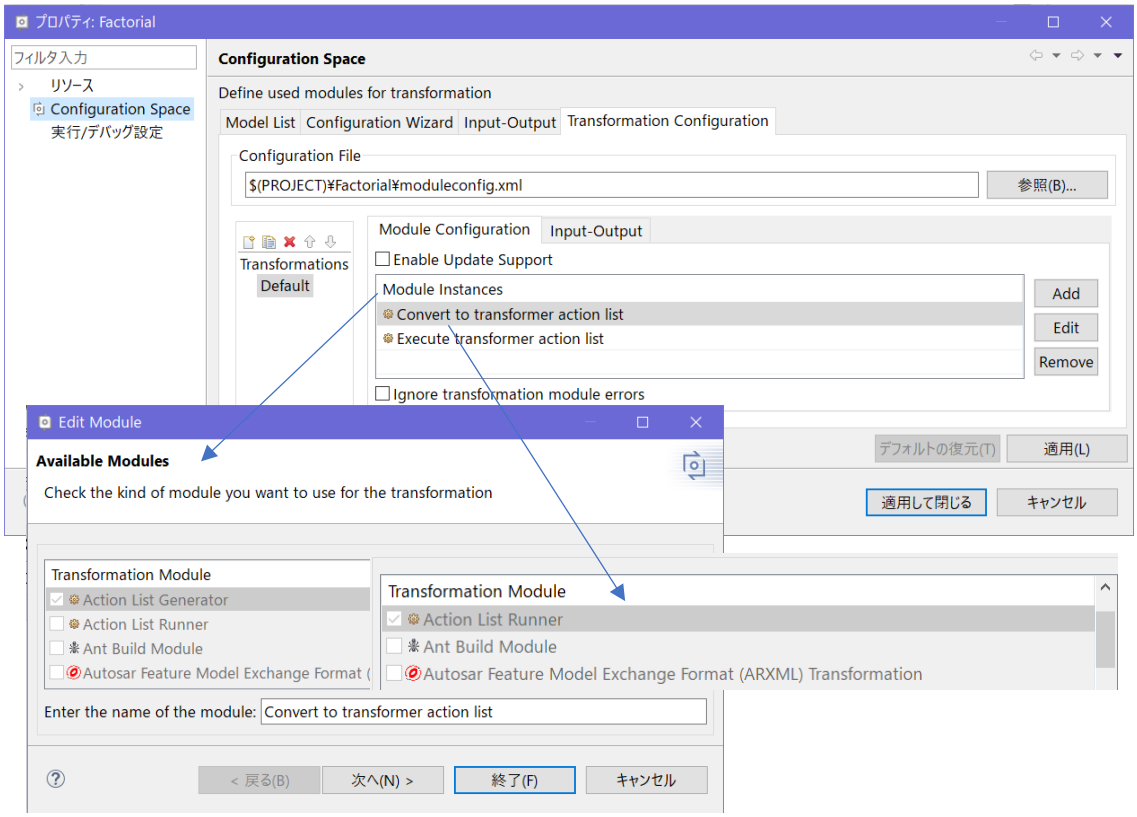


ウィザードの Input-Output タブで、Input path に入力フォルダとして $(PROJECT)¥input、Output path に出力フォルダとして $(PROJECT)¥output$(VARIANT)、Clear transformation output directory と Create transformation output directory がチェックされていることを確認します。



Transformation Configuration タブで、Module Instances に登録されているモジュールが Action List Generator と Action List Runner であるか確認します。(下図のように、登録されているモジュールをダ

ブルクリックしたウィザードで Action List Generator や Action List Runner がチェックされているかどうかを確認します)



そうでない場合、すでに設定されているモジュールがあれば削除した後、両者を追加します。

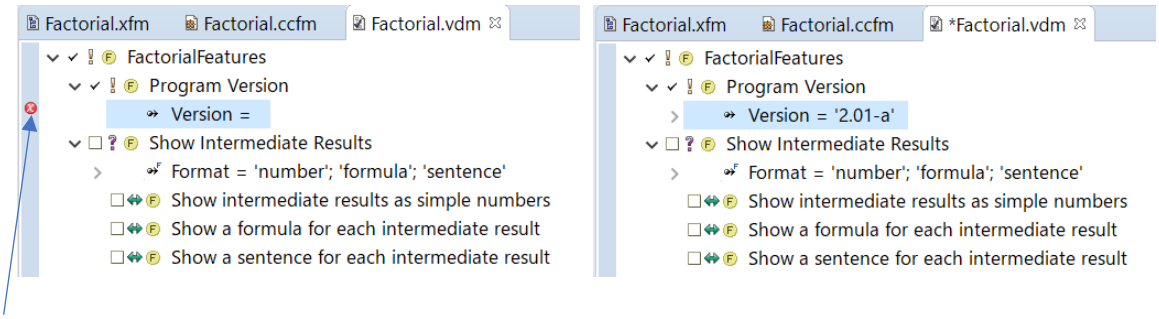
Add で開くウィザードで、Action List Generator をチェックして選択し、終了 します。(モジュール名称は、デフォルトで入力されている Convert to transformation action list のままとします)

これで新たに Convert to transform action list モジュールがモデル変換のモジュール構成に追加されました。このモジュールは、ファミリーモデルの要素にしたがって、次に追加するモジュールが実行する変換アクションのリストを生成します。

再度 Add して同様に Action List Runner を選択し、終了 します。

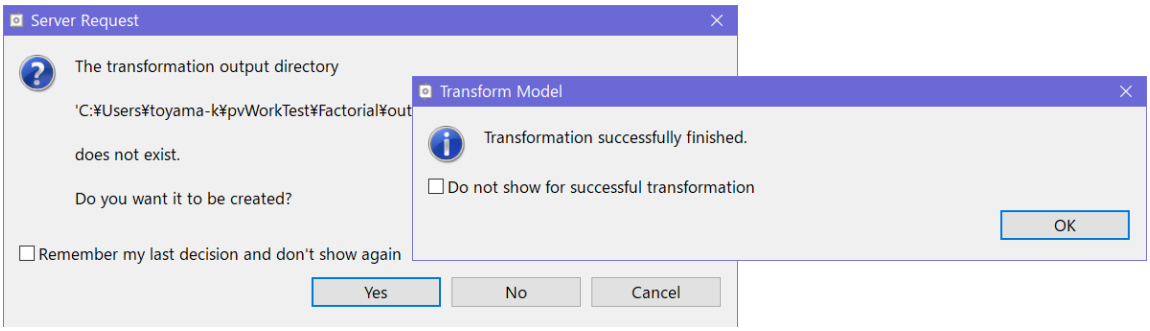
7. バリアントの生成

ここまでの設定で、pure::variants プロジェクトと input フォルダのアプリケーションファイルからモデル変換を行う準備ができました。Configuration Space (Factorial) にある `Factorial.vdm` をダブルクリックしてバリアントモデルを開き、フィーチャ Program Version の属性 Version をダブルクリックして値に「2.01-a」を入力します (アプリケーションのバージョン番号を 2.01-a として設定することを意味します)。

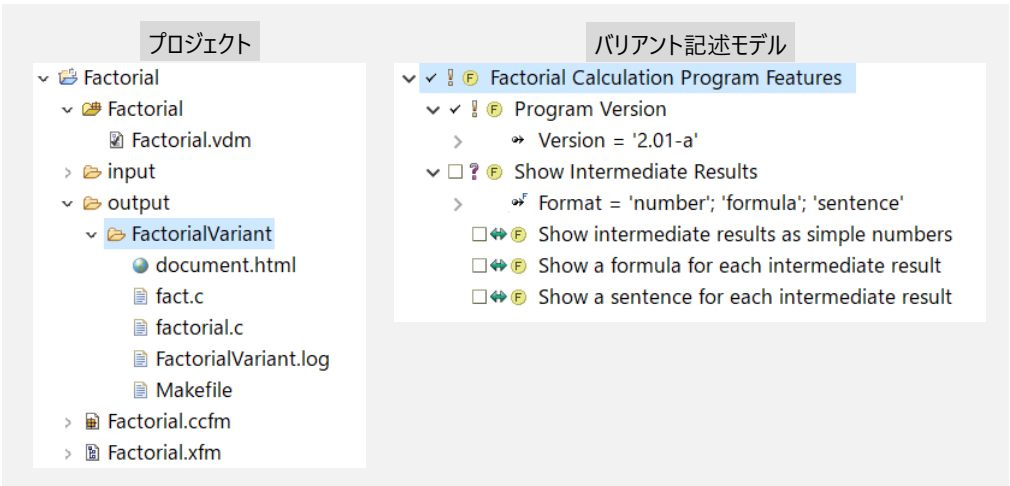


Version に値が設定されていないというエラーが報告されているものです。

ツールバーの Transform Model ボタン ( ) をクリックしてモデル変換します。output フォルダを作成するかどうかのダイアログでは Yes で進みます。変換成功のダイアログは OK します。



`output` フォルダ (の `FactorialVariant` フォルダ) に、変換後の `fact.c`、`factorial.c`、`Makefile`、`document.html`、とログが生成されます。



output フォルダに生成される変換後の fact.c の内容は以下のようになります。p.25 の生成コードイメージと同じであることを確認してください。

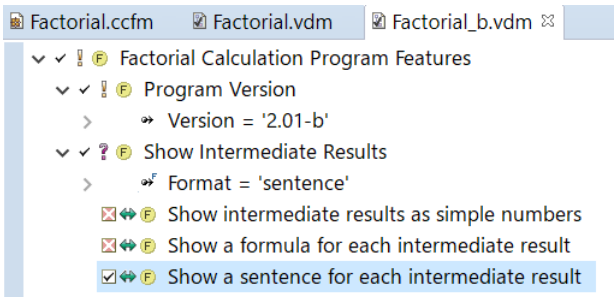
```
#include <stdio.h>

void info() {
  printf("Factorial Calculation, Version 2.01-a\n");
}
#include <stdio.h>
#include <stdlib.h>

int factorial(int x);

int main(int argc, char** argv)
{
  info();

  if (argc > 1) {
    int x = atoi(argv[1]);
    printf("%d! = %d\n", x, factorial(x));
  }
  return 0;
}
```

同様に factorial.c の内容は以下です。PVSCL 条件付きテキストの「PVSCL:～」の部分が除去されています。

```
//

static int factorialOf(int x);

int factorial(int n)
{
  return factorialOf(n);
}

static int factorialOf(int x)
{
  int result;

  if (x <= 1) {
    result = 1;
  } else {
    result = x * factorialOf(x-1);
  }

  //

  return result;
};
```

別のバリアントを生成するため、Configuration Space (Factorial) のコンテキストメニューから New > Variant Model を選択し、ウィザードで Variant model name を「Factorial_b」として作成します。Factorial_b.vdm が作成されます。

この Factorial_b.vdm で、フィーチャ **Show Intermediate Results** とその子の **Show a sentence for each intermediate result** を選択します。また、Program Version の属性 Version を「2.01-b」とします。



Transform Model でモデル変換を実行し、プロジェクトを更新すると output フォルダに新たにフォルダ Factorial_b が生成され、このバリアントに対する factorial.c がそこに生成されます。内容は以下のようになり、2 行目の「#include～」と 24 行目の「printf～」が除去されずに残っています。

```
//
#include <stdio.h>
//

static int factorialOf(int x);

int factorial(int n)
{
  return factorialOf(n);
}

static int factorialOf(int x)
{
  int result;

  if (x <= 1) {
    result = 1;
  } else {
    result = x * factorialOf(x-1);
  }

  //
  //
  printf("Factorial of %d is %d.\n", x, result);
  //
  //

  return result;
}
```

二つの vdm でそれぞれ生成されたドキュメントファイルのバリアントは下図のようになります。左側は Factorial.vdm で生成されたドキュメント (バージョンが 2.01-a、中間結果を表示しない) で、右側は Factorial_b.vdm フィーチャで生成されたドキュメント (バージョンが 2.01-b、中間結果の表示が文による) です。

Factorial.vdm

Factorial Calculation Program

Program version 2.01-a

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

Factorial_b.vdm

Factorial Calculation Program

Program version 2.01-b

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:

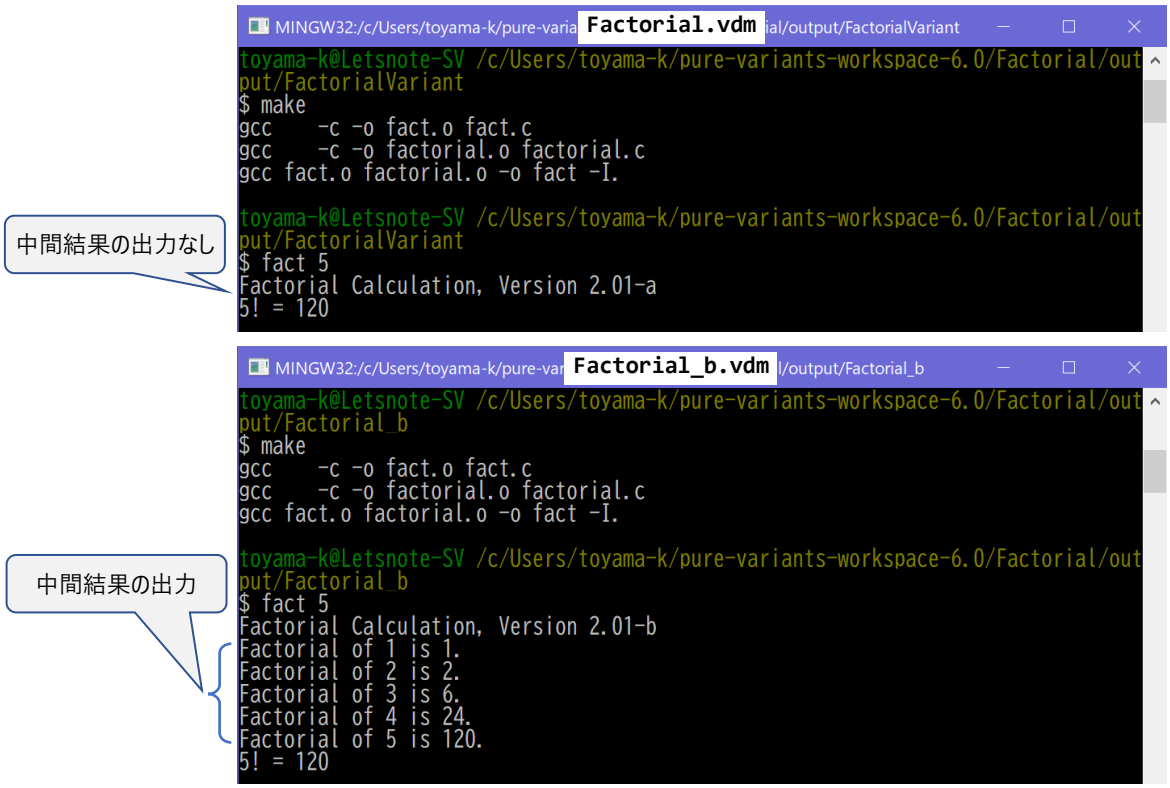
3! = 6

Intermediate Results

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

バリアントのアプリケーションを実行するには、output のバリアントそれぞれのフォルダにソースと Makefile が生成されていますので、make します。(本例では MinGW と MSYS を利用します[14])

make 後、生成された実行ファイル fact.exe に引数を与えて実行すると結果が出力されます。



FUJI SETSUBI

富士設備工業株式会社 電子機器事業部
〒591-8025 大阪府堺市北区長曽根町1928-1
Tel: 072-252-2128 www.fuji-setsu.co.jp

[14] コンパイル・実行環境については、資料「ソースコード、フラグのバリアント管理」も参照ください。