「Omni XcalableMPコンパイラのC++対応内部表現とコード変換器の開発」報告書

2017年3月30日

株式会社トランス・ニュー・テクノロジー

内容

[1 本報告書の構成 3](#_Toc478566868)

[2 納品物の構成 4](#_Toc478566869)

[2.1 docs/ ディレクトリ 4](#_Toc478566870)

[2.2 CtoXcodeML/ ディレクトリ 6](#_Toc478566871)

[2.3 CXXtoXML/ ディレクトリ 6](#_Toc478566872)

[2.4 XcodeMLtoCXX/ ディレクトリ 7](#_Toc478566873)

[2.5 tests/ ディレクトリ 8](#_Toc478566874)

[2.6 scripts/ ディレクトリ 8](#_Toc478566875)

[2.7 schema/ ディレクトリ 9](#_Toc478566876)

[3 ビルド手順 10](#_Toc478566877)

[3.1 Clangのビルド 10](#_Toc478566878)

[3.2 ClangXcodeMLの各ツールのビルド 12](#_Toc478566879)

[3.2.1 CtoXcodeMLのビルド 12](#_Toc478566880)

[3.2.2 CXXtoXMLのビルド 12](#_Toc478566881)

[3.2.3 XcodeMLtoCXXのビルド 13](#_Toc478566882)

[3.3 ビルドされたツールの利用方法 13](#_Toc478566883)

[3.3.1 CtoXcodeMLの利用方法 13](#_Toc478566884)

[3.3.2 CXXtoXMLの利用方法 15](#_Toc478566885)

[3.3.3 XcodeMLtoCXXの利用方法 16](#_Toc478566886)

[4 作業報告 17](#_Toc478566887)

[4.1 ツール実装とXcodeML仕様拡張の設計について 17](#_Toc478566888)

[4.1.1 C++対応のための正変換ツールの設計変更 17](#_Toc478566889)

[4.1.2 XcodeMLの仕様の不備に対する対応 18](#_Toc478566890)

[4.1.2.1 独自属性 18](#_Toc478566891)

[4.1.2.2 clangDecl要素とclangStmt要素 20](#_Toc478566892)

[4.1.2.3 独自要素 21](#_Toc478566893)

[4.1.2.4 その他 22](#_Toc478566894)

[4.1.3 CXXtoXMLを用いた正変換の処理手順 22](#_Toc478566895)

[4.2 納品ソフトウェアの内部実装に関する解説 22](#_Toc478566896)

[4.2.1 CtoXcodeMLの各ソースコードについて 22](#_Toc478566897)

[4.2.2 CXXtoXMLの各ソースコードについて 23](#_Toc478566898)

[4.2.3 XcodeMLtoCXXの各ソースコードについて 23](#_Toc478566899)

[4.3 テスト用ソースコードによる評価 25](#_Toc478566900)

[4.3.1 access001 から access055 まで 27](#_Toc478566901)

[4.3.2 class001からclass065まで 27](#_Toc478566902)

[4.3.3 derive001からderive065まで 27](#_Toc478566903)

[4.3.4 member001からmember058まで 28](#_Toc478566904)

# 本報告書の構成

本報告書は、本案件に関する解説書を兼ねた文書であり、下記の内容からなる。

* 納品されたソフトウェア（ClangXcodeML）の利用方法について  
  下記の各パートからなる。
  + gitリポジトリの構成
  + ビルド方法
  + ビルドされたツールの利用方法
* 作業報告書一式  
  下記の各パートからなる。
  + 納品ソフトウェアの内部実装に関する解説
  + テスト用ソースコードによる評価

# 納品物の構成

本案件で開発されたソフトウェアは、omni-compiler.org上に設置されたgitレポジトリでバージョン管理されており、納品物はここに置かれている。

gitリポジトリには下記のディレクトリが置かれている。

* docs/ ディレクトリ
* CtoXcodeML/ ディレクトリ
* CXXtoXML/ ディレクトリ
* XcodeMLtoCXX/ ディレクトリ
* tests/ ディレクトリ
* scripts/ ディレクトリ
* schema/ ディレクトリ

## docs/ ディレクトリ

* CtoXcodeMLreport-20170330.docx  
  本報告書である。
* XcodeML\_CXX.md  
  現段階でのXcodeML C++ 仕様書である。 Pandoc 形式で書かれている。Pandoc がインストールされていれば、このディレクトリの Makefile を用いて make を実行することで、 XcodeML\_CXX.docx という Microsoft Word 形式のファイルを得ることができる。また、 make XcodeML\_CXX.html を実行することで、 HTML 形式のファイルを得ることもできる。
* reference.docx  
  上記の XcodeML\_CXX.md を Pandocを用いて Microsoft Word 形式に変換する際に、見出しなどのスタイルはこのファイルのものが用いられる。（文章内容は意味を持たない）
* Makefile  
  上記の XcodeML\_CXX.md を Pandocを用いて変換する手順を記載している。単に make (あるいは make XcodeML\_CXX.docx ) と入力すれば Microsoft Word 形式のファイル、 make XcodeML\_CXX.html と入力すれば HTML 形式のファイルを得ることができる。
* extension.md  
  XcodeML仕様書にはないが、本報告書の段階で対象としたベンチマークソースコードの正変換・逆変換を扱うために仕様外の実装をした部分についての説明であり、本報告書の4.1.2節の内容として取り込まれている。
* XcodeMLtoCXX-src.md  
  逆変換ツールのファイルについての説明であり、本ドキュメントの4.2.3節の内容として取り込まれている。
* doxygen/ サブディレクトリ  
  空ディレクトリである。  
  CtoXcodeML/src/および XcodeMLtoCXX/ 内でdoxygenを実行したときに、このディレクトリ内にドキュメントが生成される。
* 上記の他、過去のドキュメントがas-isで含まれている。
  + CtoXcodeMLreport-20150925.docx  
    記述内容のうち、正変換ツール（現在は CtoXcodeML/ ディレクトリに収められているもの）についての基本設計（実装に用いているC++デザインパターンについての記述）については現状でも有効である。
  + CtoXcodeMLreport-20160330.docx  
    記述内容のうち、正変換ツールの動作テスト（現在はCtoXcodeML/testcases/ ディレクトリに収められているもの）についての記述は現状でも有効である。
  + CtoXcodeMLreport-20160929.docx  
    CtoXcodeMLのビルド方法・利用方法については現状でも有効であるが、その他の記述（とくにテスト結果の部分）は当時の実装水準に基づくものであるため、現状では有効ではない。
  + Compilability\_test.md  
    CtoXcodeMLreport-20160929.docx時点での逆変換ツールテストに関する情報であり、同報告書内に「テスト用ソースコードによる評価」の節の内容として含んでいるものに対応する。
  + XcodeML\_CXX\_0.2.docx, XcodeML\_CXX\_1.0.pdf, XcodeML\_CXX\_1.1.docx, XcodeML\_CXX\_1.2.docx の各ファイル  
    過去の各段階でのXcodeML C++ 仕様書である。

## CtoXcodeML/ ディレクトリ

旧版の正変換（C/C++ソースコードを入力にとり、対応するXcodeMLを出力する操作）ツールであるCtoXcodeMLに関するファイル一式を収めている。下記のファイル・サブディレクトリからなる。

* Makefile  
  下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make clean と make check にのみ対応。
* src/ サブディレクトリ  
  CtoXcodeMLのソースコード一式を収めている。ここにある Makefile を用いてビルドできる。手順は3.2.1節で解説する。
* testcases/ サブディレクトリ  
  CtoXcodeMLの動作テスト用のC/C++ソースコードと、それぞれについてのCtoXcodeMLによる変換結果を収めている。テスト結果についての報告は前述のCtoXcodeMLreport-20160330.docxに含まれている。

## CXXtoXML/ ディレクトリ

従来の正変換ツールを簡素化し、 Clang AST の生の姿に近い XML を出力するための新しい正変換ツールであるCXXtoXMLと、その XML を基にして XcodeMLに変換するための XSLTファイル群を収めている。

* Makefile  
  下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make all と make clean に対応 (make check にも対応しているが、 tests/ サブディレクトリにテストが置かれていないので、 make checkは特に何もしない)。
* src/ サブディレクトリ  
  CXXtoXMLのソースコード一式を収めている。ここにある Makefile を用いてビルドできる。手順は3.2.2節で解説する。
* src/XSLTs/ サブディレクトリ  
  CXXtoXML の生成する XML を XcodeML に近づけるための XSLTファイル群を収めている。 CXXtoXML とこれらの XSLT ファイルを合わせて用いることで正変換ツールとして用いることができる（このための補助スクリプトとして scripts/CXXtoXcodeMLというスクリプトが準備されている。また、その処理の流れについては4.1.3節で解説する）。
* tests/ サブディレクトリ  
  CXXtoXML単独のテストを配置するためのディレクトリだが、現状ではそのようなテストを作成していない（Makefileのみ存在するが、何もしない）。

## XcodeMLtoCXX/ ディレクトリ

逆変換（XcodeMLを入力にとり、対応する C/C++ ソースコードを出力する操作）ツールである XcodeMLtoCXXに関するファイル一式を収めている。下記のファイル・サブディレクトリからなる。

* Makefile  
  下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make all, make clean と make check にのみ対応。
* src/ サブディレクトリ  
  XcodeMLtoCXXのソースコード一式を収めている。ここにある Makefile を用いてビルドできる。手順は3.2.3節で解説する。
* tests/ サブディレクトリ  
  XcodeMLtoCXX単独での動作テストを収めている。このテストについての報告は前期の報告書(CtoXcodeMLreport-20160929.docx) の4.3.1に収められている。

## tests/ ディレクトリ

正変換と逆変換の両方を用いた一連のテストをおこなうためのファイル群が置かれている。下記のファイル・サブディレクトリからなる。

* 下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make clean と make check にのみ対応。
* CCTest/ サブディレクトリ  
  本報告書での評価作業に用いたファイル群を収めている。テスト用ソースコード（C++ベンチマーク）はライセンスの都合上この中には含まれていないが、それらのソースコードを含む CppTest.tar.gzというファイルを所定の位置に配置しておくことにより、テストが実施される。テスト手順およびテスト結果については4.3節で解説する。
* Compatibility/ サブディレクトリ  
  正変換・逆変換ツールの動作テスト用のC/C++ソースコードと、それらに対して正変換→逆変換を通した結果を作成してそれぞれ通常のC/C++コンパイラでコンパイルできることを確認するテストを収めている。前期の報告書(CtoXcodeMLreport-20160929.docx)の4.3.2以降に収めたものに相当するが、用いるツールとして新しい正変換ツール（scripts/CXXtoXcodeMLによるCXXtoXML + XSLT の実行）を用いる形に変更してある。

## scripts/ ディレクトリ

CXXtoXMLとそれに付属するXSLTファイル群を用いて正変換をおこなうためのスクリプトである CXXtoXcodeMLが置かれている。このスクリプトの実行のためには、 xsltprocとxmllintがインストールされている必要がある。

## schema/ ディレクトリ

XcodeML の仕様を XML schema の記法で表したXcodeML\_CXX.xsdファイルを収めている。

# ビルド手順

## Clangのビルド

まず、Clangをソースコードからビルドし、 /usr/local 以下にインストールしておく必要がある。そのあと、ソースコードをビルドする。本開発の開始時点でベースとした Clangのバージョンは3.6.2であり、3.7.1を用いてもビルドと実行ができることは確認できている。3.8.0以降のclang はサポート対象外とする。

下記に Clang のインストール手順を述べる。また、下記のようにしてインストールするかわりに、 docker hub より koyama41/llvm-3.6 のイメージを取得することで同様の環境をdocker を用いて構築可能である。

Clangのビルドには、同バージョンのLLVMコアとcompiler-rtが必要である。いずれも<http://llvm.org/releases/download.html>から各バージョンのものが取得できる（バージョン3.6.2は同ページの<http://llvm.org/releases/download.html#3.6.2> にあり、バージョン3.7.1は同ページの<http://llvm.org/releases/download.html#3.7.1> にある）。また同じページの先頭のリンクでsubversionを用いたリポジトリも公開されている。しかしそれらを利用して取得するよりも、github のミラーを取得したほうが利便性が高い（git checkoutだけで各バージョンのブランチに切り替えることができる）ため、そちらを利用する方法でビルド手順を示す。

% cd 作業用ディレクトリ  
% git clone https://github.com/llvm-mirror/llvm  
% cd llvm  
% git checkout release\_36（またはrelease\_37）

% cd 作業用ディレクトリ/llvm/projects  
% git clone https://github.com/llvm-mirror/compiler-rt  
% cd compiler-rt  
% git checkout release\_36（またはrelease\_37）

% cd 作業用ディレクトリ/llvm/tools  
% git clone https://github.com/llvm-mirror/clang  
% cd clang  
% git checkout release\_36（またはrelease\_37）

% cd 作業用ディレクトリ  
% mkdir build  
 ※llvmのビルド用のディレクトリを準備する。名前は何でもよい。  
% cmake –G Ninja ../llvm  
 ビルド用のサブディレクトリ・ファイル群が生成される。  
　　また、この時点ですでにCコンパイラとしてclangを用いることができる  
　　のであればCC=clang CXX=clang++ という環境変数を設定してcmakeを  
　　おこなってもよい（gcc/g++を用いるよりも高速にビルドが完了する）。  
% ninja  
 ※（PATH環境変数で探せる位置に無いなら適宜フルパスで指定する）  
 ビルドが実行される。マルチコアの場合には自動的に並列ビルドとなる。  
 初回は4コア2スレッドのCPU（8並列）でも二時間ぐらいかかる。  
 明示的に並列度を指定する場合は-jオプションを用いる。ビルドの終盤  
　　でリンクをおこなう際にはメモリ不足になりやすいので、ビルドに失敗  
　　した場合には並列度を下げてやり直すと良い。また、前述のようにして  
　　clang/clang++を用いた方がメモリ消費も少なくて済むようである。  
　　一度ビルドが成功したあとは、再ビルドする際にcmakeから実行する  
　　必要はなく、ninjaコマンドのみを実行すればよい。

以上は通常のRelease版のビルド手順である。これ以外に、デバッグ版のビルドを下記のようにしておこなうことができる。このようにすると、LLVMの各種のコマンド(optなど)で –debug や –debug-only オプションが使えるようになり、コンパイラ内部のデバッグに役立つ情報が得られるようになる。

% cmake –G ninja –DCMAKE\_BUILD\_TYPE=Debug ../llvm  
% ninja  
 ※ビルドを実行（既に実行済みであれば、必要な部分だけリビルド）する。  
 Releaseビルドに比べてさらに多くのメモリを必要とするので、  
 メモリ不足で失敗するようなら-jオプションで並列度を下げて調整する。  
% ninja install  
 ※/usr/local以下にインストールされる。

CMakeのこの他のオプション指定については<http://llvm.org/docs/CMake.html> に書かれているので、そちらを参照されたい。

## ClangXcodeMLの各ツールのビルド

### CtoXcodeMLのビルド

ビルドのためにlibxml2が必要である。Ubuntuであれば  
 % sudo apt-get libxml2-dev  
のようにしてインストールする。

gitレポジトリから取得した CtoXcodeML/src/ ディレクトリ内で make を実行すればビルドが実行され、 CtoXcodeMLという名の実行ファイルが生成される。clang/LLVMとして3.6.2を用いている場合には警告が出ないが、3.7.1を用いている場合にはClang ASTの各種のenum型の定義において新規の値が追加されたために “…… not handled in switch” の警告が出る（現状ではそのようなASTを扱わないため無視して構わない）。

make distcleanをすると、ビルド結果が全て消される。

Make cleanをすると、ビルド結果のうち、もっともビルドに時間のかかるXcodeMlRAV.oを残して、その他を消す。

他にもMakefile内にいくつかターゲットが記述されているが、開発途上段階でのテスト用に準備されたものが多く、動作は保障されない。

### CXXtoXMLのビルド

ビルドのためにlibxml2が必要である。Ubuntuであれば  
 % sudo apt-get libxml2-dev  
のようにしてインストールする。

gitレポジトリから取得した CXXtoXML/src/ ディレクトリ内で make を実行すればビルドが実行され、 CXXtoXMLという名の実行ファイルが生成される。

make distcleanをすると、ビルド結果が全て消される。

make cleanをすると、ビルド結果のうち、もっともビルドに時間のかかるXMLRAV.oを残して、その他を消す。

また、 CXXtoXML/XSLTs/ サブディレクトリ内に、 CXXtoXMLが出力したXMLファイルを XSLT プロセッサで加工するためのサンプルが収められている。これらを利用するには、 xsltproc および xmllintをインストールする必要がある。Ubuntuであれば  
 % sudo apt-get xsltproc libxml2-utils  
のようにしてインストールする。

### XcodeMLtoCXXのビルド

ビルドのためにlibxml2が必要である。Ubuntuであれば  
 % sudo apt-get libxml2-dev  
のようにしてインストールする。

gitレポジトリから取得した XcodeMLtoCXX/src/ ディレクトリ内で make を実行すればビルドが実行され、 XcodeMLtoCXX/ ディレクトリ直下にXcodeMLtoCXXという名の実行ファイルが生成される。

make cleanをすると、ビルド結果がすべて消される。

## ビルドされたツールの利用方法

### CtoXcodeMLの利用方法

例として、

clang -I some-include-dir/ -D user\_defined\_macro=value foo.c

としてコンパイルするような foo.c を対象にする場合、

./CtoXcodeML foo.c -- -I some-include-dir/ -D user\_defined\_macro=value

のように「-- の前に対象ファイル名、 -- のあとにその他のオプション」として指定する。その他のオプションが不要なときも「--」が必要である（この -- が無いと ./compile\_commands.json というファイルが用いられる。これについての詳細は<http://clang.llvm.org/docs/JSONCompilationDatabase.html>を参照のこと）。

以下は、 -- よりも左側に指定することのできるCtoXcodeML独自のオプションについて解説する。

./CtoXcodeML -help

でオプションの一覧が出る。

-file - emit 'file'

-lineno - emit 'lineno'

-column - emit 'column'

…これらはそれぞれ、出力されるXcodeMLの要素にソースコードのファイル名情報・行番号情報・桁位置情報を付加するオプションである。 C\_Frontと同一の結果を得たい場合には –file と –lineno を指定する必要がある。

-disable-typeTable - disable <typeTable>

-disable-symbols - disable <globalSymbols>, <symbols>

-disable-declarations - disable <globalDeclarations>, <declarations>

…これらはそれぞれ、 XcodeMLの一部の出力を抑制するためのオプションである。主にデバッグ用に用いる。

-trace-typeTable - emit traces on <typeTable>

-fulltrace-typeTable - emit full-traces on <typeTable>

-trace-symbols - emit traces on <globalSymbols>, <symbols>

-trace-declarations - emit traces on <globalDeclarations>, <declarations>

-trace-rav - trace Recursive AST Visitor

…これらはそれぞれ、 XcodeMLの出力中にコメントとしてデバッグ情報を含めるためのオプションである。

-typenamemap=<string> - a map file of typename substitution

…これはXcodeMLのtypeTableで扱われる型名を置換するためのオプションである。置換したい型名と置換後の型名をそれぞれ空白区切りで交互に並べたファイルを準備し、そのファイル名を上記の<string>のところに指定する。ただしファイルの末尾に空白や改行があると正しく読み込めなくなるので、ファイルの最終行は改行無しで終わらなければならない。

これ以外にもいくつかのオプションがあるが動作保障はされない。

### CXXtoXMLの利用方法

このツールのコマンドラインオプションの指定は CtoXcodeMLに準じる。ただし、 -file, -lineno, -column オプションは無く、常にこれらすべて（ファイル名、ライン番号、カラム位置）の情報が出力に含まれる。

このツールは、ClangASTの情報をなるべくそのまま出力することを意図して作成されているため、出力結果を読むためにはClangASTの内部構造の知識が必要となるが、ClangASTそのものの詳細についてはここでは解説せず、ClangASTがどのようなXMLになるのかの対応関係のみを概説する。

* 全体は Program というXML要素となり、その子要素として clangASTという要素が生成され、そこにclangASTの構造が生成される。
  + clangのDeclクラスはclangDeclという要素で表現され、ノードの種類はclass属性に入り、種類ごとの細かい情報も属性値として表現される。
  + clangのStmtクラスはclangStmtという要素で表現され、ノードの種類はclass属性に入り、種類ごとの細かい情報も属性値として表現される。
  + clangのTypeLocクラスはclangTypeLocという要素で表現され、ノードの種類はclass属性に入り、xcodemLTypeTable（後述）でのデータ型識別名がtype属性として入る。
  + clangのDeclarationNameInfoクラスはclangDeclarationNameInfoという要素で表現され、ノードの種類はclass要素に入り、名前そのものはその要素のコンテンツとして表現される。
  + clangのNestedNameSpecifierクラスはclangNestedNameSpecifierという要素で表現され、xcodemlNnsTable（後述）でのNNS識別名がnns属性として入り、その他の各種の情報が属性値として表現される。
  + clangのConstructorInitializerクラスはclangConstructorInitializerという要素で表現され、各種の情報が属性値として表現される。
* clangASTの木構造そのものではないが、XcodeMLへの変換の際に必要となる情報として、xcodemlTypeTableとxcodemlNnsTableをそれぞれ出力する。
  + clangDecl の子要素としてxcodemlTypeTableが生成される。このうち、トップレベルのclangDeclの子がXcodeMLでのtypeTableとなる。それ以外の場所にあるものはlocalTypeTableなどとして扱われるべきものであるが、現状ではXSLTや逆変換で扱っていない。
  + clangDecl の子要素としてxcodemlNnsTableが生成される。このうち、トップレベルのclangDeclの子がXcodeMLでのnnsTableとなる。それ以外の場所にあるものはlocalNnsTableなどとして扱われるべきものであるが、現状ではXSLTや逆変換で扱っていない。

### XcodeMLtoCXXの利用方法

XcodeMLtoCXXコマンドにはコマンドラインオプションは無く、入力ファイル（XcodeMLファイル）の名前を一つだけ指定する。結果は標準出力に出される。

# 作業報告

## ツール実装とXcodeML仕様拡張の設計について

### C++対応のための正変換ツールの設計変更

2015年度より開始された本作業において、当初は正変換ツールとして「clangのライブラリを用いてClangASTをXcodeMLに変換する」という部分を一つのC++プログラムとして作成しており、これをCtoXcodeMLと呼んでいた。これは、Cの言語要素と少々のC++の言語要素を扱うという限定された部分においては、比較的うまく動いていた。しかし、その後さらなる C++ 対応を進めていくにあたり、この構造では下記のような困難が発生することがわかってきた。

* 正変換ツールでは、ClangASTの木構造をたどるために、clangライブラリのRecursiveASTVisitorクラスを用いて実装しているが、ClangAST の構造と XcodeML の構造の差が大きいため、(RecursiveASTVisitorが想定している)「ClangAST の構造に基づいて再帰処理をおこなう」という形では扱いきれない部分が多くなってきた（CtoXcodeMLにおいてもさまざまな工夫でこれを扱っていたが、構造がわかりにくく、バグの元となっていた）。
* ClangASTから「逆変換ツールにとって必要な情報」を取り出すためには、さまざまな ASTのノードの種類に応じてそれぞれ適切なメソッドを呼び出して情報を取得せねばならないが、従来の CtoXcodeML では「取得していない情報は単に無視されるので XcodeML に出現しない」ことになり、情報の欠落が発生していても気づきにくい。
* 逆変換に必要となる情報を扱うことを目的とするならば、「ClangASTが持っている情報を何らかの形式でファイルに出力したもの」を扱えば実装はできるが、この「何らかの形式」をXcodeMLそのものにしようと思うと、 XcodeMLの仕様の不足が問題となって実装が進みにくなる。

そこで、既存の CtoXcodeMLの実装のうち「XcodeMLの仕様に合わせるために苦労していたところ」をなくし、かわりに「ClangAST として持っている情報をなるべくそのままの形でXMLとして表現したもの」を出力するツールを作り、そのXML を別途XSLT変換することでXcodeMLに近づける、という設計に変えた。この「ClangASTとして持っている情報をなるべくそのままの形でXMLとして表現したもの」を作成するツールを CXXtoXML と名付けた。また、このXMLをXcodeMLに変換するためのXSLTファイル群も同時に開発し、これらを組み合わせたものを新たに「正変換ツール」と呼ぶことにした。

また、この実装を作ることを通じて、 XcodeML の仕様の不備について洗い出すことがやりやすくなった、という効果が得られた。次節ではそれについて述べる。

### XcodeMLの仕様の不備に対する対応

#### 独自属性

##### is\_access\_declaration属性

usingDecl要素はis\_access\_declarationを持ってもよい。 その属性値は、usingDecl要素がアクセス宣言(access-declaration)を表現するとき真("1"または"true")、 そうでないとき偽("0"または"false")である。 省略時の値は偽である。

##### is\_const属性、is\_volatile属性(functionType要素)

メンバー関数の型を表現するfunctionType要素はis\_const属性、is\_volatile属性を持ってもよい。 その属性値は、それぞれconst、volatileメンバー関数であるとき真("1"または"true")、 そうでないとき偽("0"または"false")である。 省略時の値は偽である。

##### is\_pure属性

非staticメンバー関数を表現するfunctionDecl要素はis\_pure要素を持ってもよい。 その属性値は、純粋virtualメンバー関数であるとき真("1"または"true")、 そうでないとき偽("0"または"false")である。 省略時の値は偽である。

##### is\_static\_data\_member属性

クラスのデータメンバーを表現するvarDecl要素はis\_static\_data\_member属性を持ってもよい。 その属性値は、staticデータメンバーを表現するとき真("1"または"true")、 そうでないとき偽("0"または"false")である。 省略時の値は偽である。

##### is\_this\_declaration\_a\_definition属性

classDecl要素はis\_this\_declaration\_a\_definition属性を持ってもよい。 その属性値は、クラス宣言が定義本体を持つとき真("1"または"true")、 そうでないとき偽("0"または"false")である。 省略時の値は偽である。

##### language\_linkage属性

functionDecl要素・functionDefinition要素・varDecl要素はlanguage\_linkage属性を持ってもよい。 その属性値は"C"または"C++"である。 省略時の値は"C++"である。

##### nns属性(memberRef要素)

修飾子付きの名前へのクラスメンバーアクセス(例: a->T::x)を表現するmemberRef要素はnns属性を持つ。 その属性値は対応するNNS識別名である。

##### nns属性(Var要素)

修飾子付きの変数参照を表現するVar要素はnns属性を持つ。 その属性値は対応するNNS識別名である。

##### parent\_class属性

メンバー関数を表現するfunctionDecl要素・functionDefinition要素はparent\_class属性を持つ。 その属性値は所属するクラスのデータ型識別名である。

constructor要素またはdescructor要素を持つ functionDecl要素・functionDefinition要素から識別子を復元するのに使われている。

##### type属性(functionDecl要素・functionDefinition要素・varDecl要素)

functionDecl要素・functionDefinition要素・varDecl要素はtype属性を持ってもよい。 その属性値は関数(または変数)の型に対応するデータ型識別名である。 省略された場合、name要素(またはoperator要素・constructor要素・destructor要素) および入力されたXML文書のもつsymbols要素・globalSymbols要素から適切なデータ型識別名を決定する。 適切なデータ型識別名が一意に決定できない場合はエラーである。

#### clangDecl要素とclangStmt要素

Clang ASTの定義するノードであって、XcodeML/C++に対応する要素を持たないものを表現する。

##### clangDecl[@class="Friend"]

friend宣言を表現する。

friend関数の宣言に対応するclangDecl要素は次の形式に従う。

<clangDecl class="Friend">  
  functionDecl要素 or functionDefinition要素  
</clangDecl>

friendクラスの宣言に対応するclangDecl要素は次の形式に従う。

<clangDecl class="Friend">  
  <typeLoc type= データ型識別名 >  
</clangDecl>

##### clangStmt[@class="CXXMemberCallExpr"]

<clangStmt class="CXXMemberCallExpr">  
  式の要素  
  [ 式の要素  
   ... ]  
</clangStmt>

非staticメンバー関数の呼び出しを表現する。

##### clangStmt[@class="CXXTemporaryObjectExpr"]

<clangStmt class="CXXTemporaryObjectExpr"  
  type= データ型識別名 >  
  <typeLoc/>  
  [ 式の要素  
    ... ]  
</clangStmt>

C++における関数形式の明示的型変換を表現する。

#### 独自要素

##### classDecl要素

<classDecl>  
  [ 宣言の要素 ([-@sec:decl])  
  ... ]  
</classDecl>

クラス宣言を表現する。

##### constructorInitializerList要素

<constructorInitializerList>  
  [ constructorInitializer要素  
  ... ]  
</constructorInitializerList>

C++のmem-initializer-list(メンバー初期化子リスト)を表現する。

##### constructorInitizlier要素

C++のmem-initializer(メンバー初期化子)を表現する。

非staticデータメンバーのmem-initializerに対応する constructorInitizlier要素は次の形式に従う。

<constructorInitializer  member= メンバー名 >  
  式の要素  
</constructorInitializer>

基本クラスのmem-initializerに対応する constructorInitizlier要素は次の形式に従う。

<constructorInitializer  type= データ型識別名 >  
  式の要素  
</constructorInitializer>

#### その他

* signed\_char データ型識別名

### CXXtoXMLを用いた正変換の処理手順

4.1.1節で述べたように、新しい設計方針で作られた正変換ツールは、clangを利用してC++で記述されたCXXtoXMLというプログラムと、その出力結果（XMLの形式をしているがXcodeMLではないもの）を基にして何段階かのXSLT変換をしてXcodeML仕様に近づけるための各種のXSLTファイルに分かれている。これは、2.3節で述べたようにscripts/CXXtoXcodeMLというスクリプトを用いることでこの一連の手続きの詳細を知ることなく正変換ツールとして利用することができる。下記ではその中でおこなわれている処理手順について軽く解説する。

1. 入力として指定されたファイルを、 CXXtoXMLを用いてXMLに変換し、結果を標準出力に出す。
2. 前項の標準出力を受け、xsltprocでdrop\_prop\_column.xslを適用する。  
   CXXtoXMLの生成するXMLにはソースファイルの位置情報としてfile,lineno以外にcolumn（桁位置）も含まれているのに対し、XcodeMLにはそのような属性はないので、これを削除する。結果は標準出力に出される。
3. 前項の標準出力を受け、xsltprocでdrop\_prop\_file.xslを適用する。  
   これは、ソースファイルの位置情報のうちfileを削除するものであり、本来は必要のない手順であるが、本テストにおいてファイル名の保持は必要ではなく、単にXcodeMLの目視での読みづらさを増すだけであったので、便宜的に削除することにしたものである。結果は標準出力に出される。
4. 前項の標準出力を受け、xsltprocでreorder\_decl.xslを適用する。  
   これは、for文での初期化（カッコの中の最初のセミコロンまでの部分）が式ではなく宣言の形式をしていた場合、その宣言をXcodeMLでは扱えないので、for文全体を包むcompoundStmtを作成して宣言をfor文の外（生成されたcompoundStmtの内側の先頭）に移動する。結果は標準出力に出される。
5. 前項の標準出力を受け、xsltprocでadd\_symbols\_elem.xslを適用する。  
   CXXtoXMLの生成したXMLには、XcodeMLでのsymbols要素やglobalSymbols要素に相当する部分はなく、それらの情報はclangASTの情報の一部として保持されている、という形をとっている。ここから適宜情報を抽出し、XcodeMLでのsymbols要素を適切な場所に追加する処理がここでおこなわれる。結果は標準出力に出される。
6. 前項の標準出力を受け、xsltprocでadd\_global\_symols\_elem.xslを適用する。  
   前項の処理と同様にして、globalSymbols要素を追加する処理がここでおこなわれる。結果は標準出力に出される。
7. 前項の標準出力を受け、xsltprocでProgram2XcodeProgram.xslを適用する。  
   この処理がXcodeMLへの変換の主要な部分である。clangASTでのclangStmtやclangDeclといった要素名を、それぞれ対応するXcodeMLの要素名に変換したり、構造上の差異を変更したりする。結果は標準出力に出される。
8. 最後に、前項の標準出力を受けて、xmllint –formatを適用し、結果を標準出力に出す。これは、XMLとしての目視での読みやすさを確保するためのものであり、XML構造の意味的な変化は発生しない。

## 納品ソフトウェアの内部実装に関する解説

### CtoXcodeMLの各ソースコードについて

* XcodeMlVisitorBase.cpp, XcodeMlVisitorBase.h  
  下記の各種の \*Visitor.cpp, \*Visitor.h の実装の基底クラスである  
  class XcodeMlVisitorBase を準備している部分。  
  意味的にはこのさらに上位に CRTPパターンで書かれたRecursiveASTvisitor クラスを基底に持つのだが、RecursiveASTvisitorクラスは大量のメソッドを持つため、本当に各種の\*Visitorの基底クラスとして実装するとコンパイル時間が何倍にもなるので、pimpl イディオム相当の class RAVBidirBridge をつかって、下記のXcodeMlRAV のほうに RecursiveASTvisitor の実装の部分を隠蔽している。
* XcodeMlRAV.cpp, XcodeMlRAV.h  
  clang の libtooling ライブラリ内の RecursiveASTvisitor.h を利用したクラスを実装している部分。 RAVBidirBridge をつかってclass XcodeMlVisitorBase との間で双方向に橋渡しをしている。
* TypeTableVisitor.cpp, TypeTableVisitor.h  
  clang の AST から XcodeML の <typeTable> 部を生成する部分。  
  また、 class TypeTableInfo というデータ構造を作成し、clang AST の QualType で示された値 (型の種別情報) とXcodeML の の type 名 (文字列) との対応関係を管理する。
* InheritanceInfo.cpp, InheritanceInfo.h  
  C++のクラス・構造体の継承関係の情報を扱う部分。TypeTableInfoでは表されていない情報を管理する。
* SymbolTableVisitor.cpp, SymbolTableVisitor.h  
  clang の AST から XcodeML の <globalSymbols> 部および <symbols> 部を生成する部分。
* DeclarationsVisitor.cpp, DeclarationsVisitor.h  
  clang の AST から XcodeML の <globalDeclarations> 部および <declarations> 部を生成する部分。この部分がCtoXcodeMLの中でもっとも大きな部分を占める。
* operator.cpp, operator.h  
  オーバーロードされた演算子の種類をXcodeMLでの演算子名に変換するためのOverloadedOperatorKindToString関数を持つ。
* CtoXcodeML.cpp  
  main 関数部分。与えられたコマンドラインから AST を構成し、上記各 Visitor にAST を渡す部分。

### CXXtoXMLの各ソースコードについて

(未執筆。これ、 docs/なんちゃら.md として書いてほしい)

### XcodeMLtoCXXの各ソースコードについて

* AttrProc.h  
  AttrProcクラステンプレートを定義する。AttrProcは、XML要素を読みながら何らかの処理を行うが、その際要素の属性に応じて処理を切りかえる。
* ClangClassHandler.{h,cpp}  
  入力されたXML文書に含まれるclangStmt、clangDecl要素を処理してC/C++プログラムを出力する。
* CodeBuilder.{h,cpp}  
  入力されたXML文書に含まれる要素のうち、clangStmt、clangDecl以外の要素を処理してC/C++プログラムを出力する。
* LibXMLUtil.{h,cpp}  
  libxmlに関わるユーティリティライブラリ。
* NnsAnalyzer.{h,cpp}  
  入力されたXML文書に含まれるnnsTable要素を読み、NNS識別名からNNSへのマップを構築する。
* Stream.{h,cpp}  
  CXXCodeGen::Streamを定義する。CXXCodeGen::Streamは、C/C++プログラムを出力するのに便利なストリームのクラスである。
* StringTree.{h,cpp}  
  CXXCodeGen::StringTreeを定義する。CXXCodeGen::StringTreeは、連接が高速にできる文字列のクラスである。
* SourceInfo.h  
  SourceInfoを定義する。SourceInfoは、入力されたXML文書を読んでC/C++プログラムを出力する際に必要となる情報を保持する。
* Symbol.h  
  SymbolMapを定義する。SymbolMapは、プログラムのある場所から可視である全ての識別子とその型(データ型識別名)に関する情報を保持する。
* SymbolAnalyzer.{h,cpp}  
  入力されたXML文書に含まれるglobalSymbols要素を読み、識別子からデータ型へのマップを構築する。
* SymbolBuilder.{h,cpp}  
  入力されたXML文書に含まれるglobalSymbols要素を読み、(特にtypedef宣言や構造体定義などの)C/C++プログラムを出力する。
* TypeAnalyzer.{h,cpp}  
  入力されたXML文書に含まれるtypeTable要素を読み、データ型識別名から型(XcodeMl::Type)へのマップを構築する。
* XMLString.{h,cpp}  
  XMLStringを定義する。XMLStringは、libxmlの文字列(xmlChar)をC++で扱うためのラッパークラスである。
* XMLWalker.h  
  XMLWalkerクラステンプレートを定義する。XMLWalkerは、XML要素を再帰的に読みながら何らかの処理を行うが、その際要素名に応じて処理を切りかえる。
* XcodeMLtoCXX.cpp  
  main関数を定義する。
* XcodeMlEnvironment.{h,cpp}  
  XcodeMl::Environmentを定義する。XcodeMl::Environmentは、データ型識別名から型(XcodeMl::Type)へのマップを表現する。
* XcodeMlNns.{h,cpp}  
  XcoedMl::Nnsを定義する。XcodeMl::Nnsは、(XcodeML/C++の定義する)NNSを表現する。
* XcodeMlType.{h,cpp}  
  XcodeMl::Typeを定義する。XcodeMl::Typeは、C++の型を表現する。

## テスト用ソースコードによる評価

今回の作業では、理化学研究所側で準備されたC++ ベンチマークテストから選択された所定のソースコードについて、正変換および逆変換が正しく動作するかの評価を主眼におき、下記のような方針で動作を確認した。

* 正変換の結果を逆変換し、その結果がC/C++コンパイラでコンパイルできるものであるかを確認する。
* いずれのソースコードも、実行すると何らかの文字列（多くは ok または ngの二文字のみ）を表示するように書かれているので、上記のコンパイル結果を実行した結果を確認する。

テスト用に用いたソースコードは、ライセンス的に公開が不可能であるため、 gitレポジトリには収められていない。かわりに、 ClangXcodeMLリポジトリを取得したディレクトリと同じ位置に CppTest.tar.gz というファイルを配置しておく（つまり、下記の手順のCCTestディレクトリからみると../../../CppTest.tar.gzという形で参照できるようにしておく）と、そのtarファイルを展開してテストを実行できるようにしてある。

テストの実行には次の手順を踏む。

* git レポジトリ tests/CCTest ディレクトリ内をカレントディレクトリとする。以下の手順はすべてこのディレクトリをカレントディレクトリとした状態でおこなう。
* make buildを実行する。  
  これは、 ../../CXXtoXML/src と ../../XcodeMLtoCXX/src での make clean; make の実行をおこなう。
* make prepare-cpptestを実行する。  
  これは、 ../../../CppTest.tar.gz をこのディレクトリ内に展開し（この時点でCppTest/ というサブディレクトリが作成されてその中にファイル一式が展開される）、その中から testobjects.txt に記載された名前をファイル名のベース部に持つ.ccファイルを選び、 .src.cpp というファイルにしてこのディレクトリの中に配置する。この際、もともとの.ccファイルのmain関数の戻り値が省略されている部分にintを明示するようにファイルを変換する。
* make check-cpptestを実行する。  
  これがテスト本体である。Testobjeects.txtに記載された名前をファイル名のベース部に持つ.src.cppファイルを対象にして、まずその名前およびそれに続くカンマを行頭に出力する。正変換をした結果を .xcodemlという拡張子で出力し、それを逆変換した結果を .dst.cpp という拡張子で出力する。ここまで成功した時点で一つ目の「OK,」が同じ行に出力される。この .dst.cppをclang++でコンパイルした結果を .out という拡張子で出力する。ここまで成功した時点で二つ目の「OK,」が同じ行に出力される。最後に、この .outを実行し、その結果が同じ行の続きとして出力される。  
  一つ目のOKが出る前に失敗した場合は、CXXtoXcodeML fail または XcodeMLtoCXX fail というメッセージがOKのかわりに生成されて、その行はそこで終わりになる。  
  一つ目のOKが出た後、二つ目のOKが出る前に失敗した場合は、 Compilation failというメッセージがOKのかわりに生成されて、その行はそこで終わりになる。
* 上記で標準出力に生成された内容と同じものが result.csv として保存されている。また、もし途中でエラーがでていればその内容はerr.logとして保存されている。

以下、テスト内容について解説する。

### access001 から access055 まで

（未執筆。どういうテストであるか簡単に解説する）

（ここに例のワークシートを適当に切り貼りする）

### class001からclass065まで

（未執筆。どういうテストであるか簡単に解説する）

（ここに例のワークシートを適当に切り貼りする）

### derive001からderive065まで

（未執筆。どういうテストであるか簡単に解説する）

（ここに例のワークシートを適当に切り貼りする）

### member001からmember058まで

（未執筆。どういうテストであるか簡単に解説する）

（ここに例のワークシートを適当に切り貼りする）