社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2007－ＳＬＤＭ－128（13）
2007／1／18

XMLをベースとしたCDFGマニピュレーションフレームワーク: CoDaMa

小原 俊逸† 史 又華† 戸川 望† 柳澤 政生† 大附 辰夫†

† 早稲田大学理工学部コンピュータ・ネットワーク工学科
〒169-8555 東京都新宿区大久保 3-4-1
E-mail: †kohara@yanagi.comm.waseda.ac.jp

あらまし 本稿では，ハードウェア/ソフトウェア (HW/SW) 協調合成システムや高位合成システム構築のための，XMLをベースとしたCDFG (Control Data Flow Graph) 操作フレームワークを提案する．CDFGは制御の流れを表すCFG (Control Flow Graph) とデータの流れを表すDFG (Data Flow Graph) で構成される．HW/SW協調合成や高位レベル合成では，アプリケーションプログラムの内部表現としてCDFGが用いられることが多い．それらの合成システムは，要求性能と設計制約を満たす最適なハードウェアやソフトウェアを合成するため，さまざまな最適化アルゴリズムによってCDFGを操作し，自動的に設計探索を行う．近年のSoC (System On a Chip) アプリケーションの大規模化に伴い，合成システムに求められる機能も高度化しており，合成システム開発にかかる工数も増加している．提案フレームワークでは，合成システム開発の生産性を向上させるため，アルゴリズムをモジュール単位で実装し，各モジュールの組み合わせによって合成システムを構築する．アプリケーションプログラムの中間表現をXMLで記述し，入出力インタフェースをライブラリとして提供することで，開発者は容易にアルゴリズムを実装し，合成システムを構築することができる．

キーワード CDFG, XML, フレームワーク，HW/SW協調合成，高位合成

CoDaMa: An XML-based Framework for Manipulating CDFGs

Shunitsu KOARA†, Youhua SHI†, Nozomu TOGAWA†, Masao YANAGISAWA†, and Tatsuo OHTSUKI†

† Dept. of Computer Science, Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
E-mail: †kohara@yanagi.comm.waseda.ac.jp

Abstract This paper proposes an XML-based framework to manipulate CDFGs (Control Data Flow Graphs) for HW/SW (Hardware / Software) co-synthesis systems or high-level synthesis systems. A CDFG is composed of CFG (Control Flow Graph) and DFGs (Data Flow Graphs). In HW/SW co-synthesis systems or high-level synthesis system, CDFGs are often adopted as an internal representation of input application programs. The systems explore design space automatically with various optimization algorithm in order to synthesize hardware and software which satisfy performance requirements and design constraints. However, with the increased scale of the recent SoC (System On a Chip) applications, synthesis systems require implemented more advanced functions, and it would result in increased development efforts. In the proposed framework, developers implement algorithm as modules and construct the synthesis systems by combination of the modules in order to improve development productivity. The developers can implement algorithm and construct the systems easily by using XML descriptions as intermediate representation of application programs and providing the input/output interface.

Key words CDFG, XML, framework, HW/SW co-synthesis, high-level synthesis

1. はじめに

半導体プロセスの微細化により，システムをワンチップに搭載可能になった．携帯電話やディジタルカメラなどの SoC (System On a Chip) アプリケーションは近年，大規模化・複雑化し，アプリケーションが要求する性能と面積・消費電力等の制約を同時に満たすシステムを設計するための設計工数は年々増大している．その一方で社会の情報化により，SoC アプリケーションの Time-to-Market の短縮化が要求され，設計者は短期間で要求性能と設計制約を同時に満たすシステムを設計するという困難な問題に直面している．こうした設計生産性危機を克服するため，C 言語等の抽象度の高い言語で記述されたアプリケーションを入力とし，要求性能と設計制約を満たすハードウェアやソフトウェアを自動的に合成するシステムが研究・開発されている．高位合成 (動作合成) 技術は，ハードウェアを合成するための入力言語として，従来の HDL (Hardware Description Language) ではなく，より抽象度の高い C 言語等からハードウェアを合成する技術である．高位合成システムは 20 年以上前から研究されており，[4], [10] など製品化されている例もある．プロセッサの HW/SW 協調合成技術は，C 言語等の動作記述からアプリケーションに特化した演算器や命令セットを持つプロセッサならびにソフトウェアを合成する技術であり，研究事例として [9] などがある．

近年の SoC アプリケーションの大規模化・複雑化により，こうした合成システムに求められる機能も高度化している．高度な合成システムは複数の系で構成され，開発者も複数人になる場合が多い．こうした開発体制において重要になるのは，入出力インタフェースと内部データ構造である．例として，合成システム内に系 A と系 B があり，系 A の出力データ X を系 B に入力する場合を考える．このとき，X の形式と解釈は系 A と系 B の開発者間で仕様を決める必要があるが，それぞれの系のアルゴリズムの実装に比べてあまり本質的とは言えない X の形式の策定と入出力処理の実装が，全体の工数に大きく影響する場合が多い．また，合成システムにとって重要な内部データ構造は，アプリケーションの動作記述であり，それには CDFG (Control Flow Data Graph) が用いられることが多い．CDFG は制御の流れを表す CFG (Control Flow Graph) とデータの流れを表す DFG (Data Flow Graph) で構成される．ただし，CDFG の表現方法やデータ構造は決まったものがあるわけではなく，合成システムの開発グループが独自で定義しているのが現状である．さらに，同じ合成システム内においても系によって CDFG の各要素に付加する情報が異なる場合が多い．

こうした背景から，我々は XML [11] をベースとした CDFG マニピュレーションフレームワーク「CoDaMa」(Control Data flow graph Manipulation framework) を提案する．CoDaMa フレームワークでは，アルゴリズムをモジュール単位で実装し，各モジュールの組み合わせによって合成システムを構築する．また，アプリケーションプログラムの中間表現を XML で記述し，入出力インタフェースをライブラリとして提供する．以上の方式により，開発者は容易にアルゴリズムを実装することが可能になり，合成システム開発の生産性を向上させることができる．

本稿の構成は以下のようになっている．第 2 節で関連研究に触れ，第 3 節で CoDaMa フレームワークの説明を行う．第 4 節で実際の開発状況から提案フレームワークを評価する．

2. 関連研究

アプリケーションプログラムの中間表現と，その中間表現を操作するツールに関連する研究として，[1], [3], [7], [8] が挙げられる．[1] は，ハードウェア記述言語 VHDL から CDFG を得るツールであり，他に CDFG から VHDL や C を出力するツールも用意されている．しかし，VHDL から得られる CDFG は関数呼び出しやメモリアクセスなど，ソフトウェア表現に必要な機能はなく，CDFG のフォーマットは独自形式のテキスト形式で，開発者が独自に情報を付加することはできない．[7], [8] は米スタンフォード大学の独自中間フォーマット SUIF (Stanford University Intermediate Format) を用いたコンパイラ開発システムである．C や Fortran と SUIF のインタフェースを用意することで，最適化アルゴリズムをモジュール単位で開発し，その組み合わせによってコンパイラを構成する．[7] ではモジュール 1 つにつき 1 つの実行ファイルになり，モジュールの入出力は SUIF ファイル (バイナリファイル) であったが，[8] では複数のモジュールをロードすることが可能である．[3] は並列化コンパイラ向け共通インフラストラクチャであり，内部に高水準中間表現 (HIR: High level Intermediate Representation) と低水準中間表現 (LIR: Low level Intermediate Representation) の 2 つの中間言語を持ち，ソース言語から HIR への変換部，HIR での各種最適化部，HIR から LIR への変換部，LIR での各種最適化部，LIR からアセンブリ言語への変換部を組み合せることによってコンパイラを実現する．[3], [7], [8] はソフトウェアを対象としたコンパイラであり，さまざまな並列化アルゴリズムを容易に実装・実験可能なフレームワークである．ハードウェアの合成を目的とした高位合成システムにそのまま導入することはできない．それらはターゲットマシンを定義し，ターゲットマシンに特化した最適化アルゴリズムを開発することでアセンブリ列を生成するが，HW/SW 協調合成システムはターゲットマシンのアーキテクチャやパラメータを探索するシステムなので，やはりそのまま導入することはできない．

また，[5] で報告されている高位合成システムでは，XML 記述で表現された DFG を用いている．この XML は [3] の HIR から生成されるとしている．

これらの既存研究と比較したときの CoDaMa フレームワークの特長は以下のとおりである．

(1) 中間表現を XML で記述するため，入出力 XML のアトリビュートなどの仕様が決まっていれば，開発者が容易に動作テスト用データを作成可能で，開発対象のモジュールのドライバやスタブを用意する必要がない．

(2) XML アトリビュートを開発者が独自に定義可能なため，アプリケーションプログラムの各要素に付加的な情報を容

易に与えることができる.

(3) 標準ではグラフやXMLアトリビュートを操作するクラスやメソッドしか存在しないため，開発者が学習する期間が短くすることができる.

(1) について，CoDaMaフレームワークでは中間表現をファイルまたはUNIXにおける標準入力・標準出力で行う．こうした方式としては[7]が類似しているが，SUIFファイルはバイナリであるため，人手で動作テスト用データを作成するのは困難である．これに比べ，XMLファイルはテキストデータであるため，人手で容易に作成可能である.

(2) について，[3]のHIR,LIRは仕様が決まっているため，合成システムの開発ごとに拡張するのはそれだけ工数がかかる．[1]で出力されるテキスト形式のCDFGも同様である．これに比べXML形式は，エレメントのアトリビュートは，パーサの変更なしで開発者が独自に定義可能で，さらに後述のインポート/エクスポートの機構を用いれば入出力処理も容易に可能である.

(3) について，[3],[7],[8]はコンパイラに必要な情報を網羅的に扱う設計のため，APIの規模が大きく，開発者が学習する期間が長くならざるを得ない．これに比べ，提案フレームワークで標準として用意しているAPIは最小限に留められており，学習期間を短くできる.

3. CoDaMaフレームワーク

本節ではCoDaMaフレームワークについて説明する．はじめにCoDaMaフレームワークにおける合成システム開発の概要について説明し，次に操作対象となるアプリケーションプログラムのモデルとXML記述方法について説明する．次にC++に依るソフトウェア・アーキテクチャについて説明し，最後に簡単なアルゴリズムの開発の方法を実例として説明する.

3.1 概　　要

CoDaMaフレームワークでは，アプリケーションプログラムの中間表現としてCDFGのXML記述を用いる．このXMLを**Program XML (Prog-XML)**と呼ぶ．CoDaMaフレームワークによる合成システム開発は，UNIXシステムの思想に則っている．すなわちUNIXシステムが小規模のツールを組み合せて構成しているように，提案フレームワークでは，**CoDaMaモジュール**と呼ばれるProg-XMLのフィルタを1つの実行ファイルとして実装し，これらの組み合わせで合成システムを構築する．CoDaMaモジュールの入出力を図1に示す．入出力のProg-XMLの他に，ユーザ（開発者）が独自に入出力を定義できる．Prog-XMLの入力と出力はそれぞれ，UNIXシステムの標準入力，標準出力になっており，図2のような使用が可能である．また，1つのProg-XMLを並列に複数のCoDaMaモジュールに入力し，出力した結果を比較する必要や，CoDaMaモジュールに与える入力パラメータを動的に変える必要が生じる場合がある．その場合は，CoDaMaモジュールを制御するためのスクリプトを用意する．このスクリプトを**CoDaMaコントローラ**と呼ぶ.

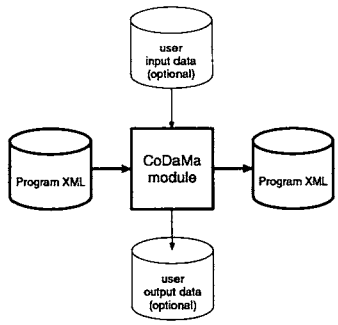
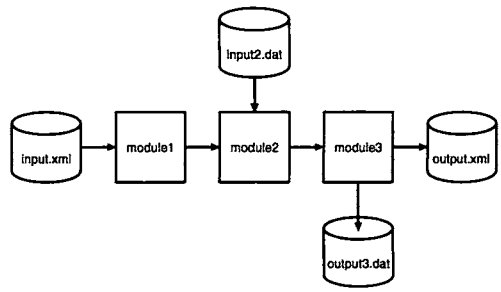


図1　CoDaMaモジュールの入出力.




```
$ cat input.xml | module1 | module2 -i input2.dat | module3 -o output3.dat > output.xml
```

図2　UNIXシステム上での実行例.

3.2 アプリケーションプログラムモデル

CoDaMaフレームワークで対象するアプリケーションプログラムのモデルを図3に示す．アプリケーションは1つの**Prog**（Program; プログラム）である．プログラムは，複数の**Subp**（SubProgram; サブプログラム）と複数の**Var**（Variable; 変数）によって構成される．Progに属するVarは静的変数である．Subpとは，C言語の関数やC++,Javaのメソッドにあたるもので名前は[3]のHIRから拝借した．Subpは1つの**CFG**と複数のVarで構成されている．Subpに属するVarはSubpの自動変数か，Subpの引数（**Arg**）である．CFGは制御の流れを表すグラフであり，CFGの各ノードには対応するDFGが存在する．**DFG**はデータの流れを表すグラフであり，DFGのノードは演算や命令を表す．このアプリケーションプログラムモデルは，C言語等を等価に表現することはできないが，画像処理や通信処理などのアプリケーションを記述するには十分であると考えている.

一般にCDFGの定義は合成システムごとに異なる．例えば高位合成システム（[5]を含む）では，分岐を含む演算を1つのグラフとして表したり，演算の開始を1つのsourceノード，演算の終了を1つのsinkノードで表すことが多いが，必ずしもすべての合成システムでそのような定義にする必要はない．そのため，CoDaMaフレームワークではCDFGの定義までは規定せず，Prog, Subp, Var, CFG, DFGの関係が図3に基づいていれば開発者がCDFGを独自に定義可能である.

3.2.1 オブジェクト指向分析

CoDaMaフレームワークでは，Prog-XMLの仕様さえ満た

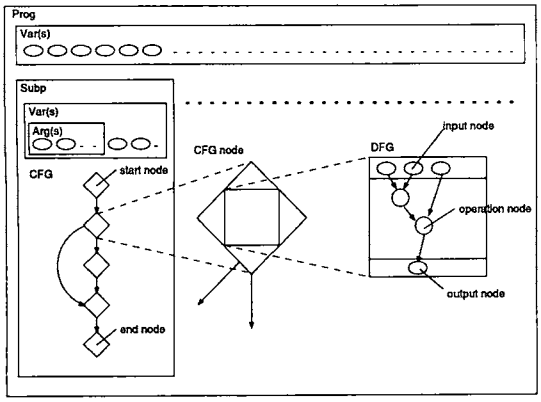


図 3　アプリケーションプログラムモデル.

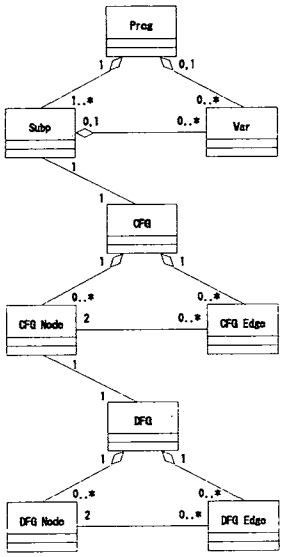


図 4　UML による各要素の概念クラス図.

していればCoDaMaモジュールやCoDaMaコントローラの開発言語は任意だが，現在はオブジェクト指向言語であるC++言語とRuby言語[6]のためのパーサを用意している．オブジェクト指向言語では，アプリケーションプログラムの各要素をオブジェクトとして捉えることができるので，可読性と保守性の高いアルゴリズム記述が可能と考えている．図4にアプリケーションプログラムモデルの各要素について，オブジェクト指向分析したときのUMLによる概念クラス図を示す．

3.2.2　XML 記述

図5にアプリケーションプログラムの各要素の関係に基づいたProg-XMLの記述例を示す．図4の各クラスと図5の各エレメントの対応は表1のようになっている．

また，各エレメントの関係を表すため，必須アトビュートが存在するエレメントがある．Prog, Subp, Varの必須アトリビュートは名前を表すnameである．CFG NodeとDFG Nodeの必須アトリビュートはノードの識別子となるnidである．

表 1　プログラム要素と XML エレメント

プログラム要素	XML エレメント
Prog	//prog
Subp	//prog/subps/subp
Var	//prog/vars/var //prog/subps/subp/vars/var
CFG	//prog/subps/subp/cfg
CFG Node	//prog/subps/subp/cfg/nodes/node
CFG Edge	//prog/subps/subp/cfg/edges/edge
DFG	//prog/subps/subp/cfg/dfgs/dfg
DFG Node	//prog/subps/subp/cfg/dfgs/dfg/nodes/node
DFG Edge	//prog/subps/subp/cfg/dfgs/dfg/edges/edge

```
<prog name="example">
 <vars>
  <var name="x" type="int" />
  <var name="y" type="int" />
 </vars>
 <subps>
  <subp name="func1" type="int">
   <vars>
    <var name="a" type="int" />
    <var name="b" type="int" />
   </vars>
   <cfg>
    <nodes>
     <node nid="0" />
     ...
     <node nid="10" />
    </nodes>
    <edges>
     <edge source_nid="0" target_nid="1" />
     ...
     <edge source_nid="9" target_nid="10" />
    </edges>
    <dfgs>
     <dfg did="0">
      <nodes>
       <node nid="0" class="input" name="a" />
       <node nid="1" class="input" value="1" />
       <node nid="2" class="operation" name="sub"
        control_step="1" />
       <node nid="3" class="output" name="b" />
      </nodes>
      <edges>
       <edge source_nid="0" target_nid="2"
        target_idx="0" />
       <edge source_nid="1" target_nid="2"
        target_idx="1" />
       <edge source_nid="2" target_nid="3"
        source_idx="0" />
      </edges>
     </dfg>
     ...
    </dfgs>
   </cfg>
  </subp>
 </subps>
</prog>
```

図 5　Prog-XML の記述例（一部）.

CFG EdgeとDFG Edgeの必須アトリビュートはソースノードとターゲットノードを指定するsource_nidとtarget_nidである．DFGの必須アトリビュートは対応するCFG Nodeのnidを指定するdidである．

3.3　ソフトウェア・アーキテクチャ

CoDaMaフレームワークのための開発用ライブラリを**CoDaMaライブラリ**と呼ぶ．CoDaMaライブラリでは，Prog-XMLの入出力インタフェースの他，合成システム開発のため

```
(a) CFGHandler*
    SubpHandler::get_cfg_handler() const;
(b) SubpHandler*
    ProgHandler::get_subp_handler
     (const SubpDescriptor& sd) const;
```

図 6　ハンドラゲットメソッド.

```
...
DFGHandler* dh = ...;
for (DFGNodeIterator i = dh->nodes(); !i.end(); ++i) {
  DFGNodeDescriptor n = *i;
  DFGNodeHandler* dnh = dh->get_node_handler(n);
  ...
```

図 7　イテレータの使用例.

の様々なツールが用意されている．現在 CoDaMa ライブラリで CoDaMa モジュール開発用言語となっている C++言語におけるソフトウェア・アーキテクチャについて説明する．

CoDaMa モジュールは，次の機構を利用することで開発する．

- ハンドラクラス，CoDaMa クラス
- ディスクリプタ，ハンドラゲッタメソッド
- イテレータ
- アトリビュート，プロパティ
- インポート，エクスポート
- CoDaMa エクステンション

3.3.1　ハンドラクラス，CoDaMa クラス

アプリケーションプログラムの各要素を取り扱うための**ハンドラクラス**が用意されており，これを定義することで CoDaMa モジュールを開発する．ハンドラクラスは，各要素毎に，ProgHandler, SubpHandler, VarHandler, CFGHandler, CFGNodeHandler, CFGEdgeHandler, DFGHandler, DFGNodeHandler, DFGEdgeHandler が用意されている．これらを**CoDaMa クラス**と呼ぶ．

3.3.2　ディスクリプタ，ハンドラゲッタメソッド

アプリケーションプログラムの各要素には，同じ集約要素を持つ要素間で固有な**ディスクリプタ**（記述子）が存在する．ディスクリプタは，各要素毎に，SubpDescriptor, VarDescriptor, CFGNodeDescriptor, CFGEdgeDescriptor, DFGNodeDescriptor, DFGEdgeDescriptor が用意されている．ProgDescriptor が存在しないのは，ルート要素だからであり，CFGDescriptor と DFGDescriptor が存在しないのは，それぞれ Subp，CFG Node と 1 対 1 対応なので不要だからである．

各要素のハンドラには**ハンドラゲットメソッド**が用意されている．1 対 1 対応の場合は引数なし（図 6 (a)）で，1 対 N 対応の場合はディスクリプタを引数にする（図 6 (b)）ことで各ハンドラオブジェクトのポインタを得ることができる．

3.3.3　イテレータ

1 対 N 対応の要素を巡回するためにイテレータクラスが用意されている．イテレータクラスには，SubpIterator, VarIterator, CFGNodeIterator, CFGEdgeIterator, DFGNodeIterator, DFGEdgeIterator がある．イテレータオブジェクトは*演算子でディスクリプタを返す。イテレータを返す CoDaMa クラスのメソッドは，Prog-XML の複数形のタグ名と同じである．イテレータの使用例を図 7 に示す．

3.3.4　アトリビュート，プロパティ

アトリビュートは XML における属性であり，C++においては std::string 型で表現されている．アトリビュートは XML パース時に自動的に取り込まれ，XML 出力時に自動的に出力される．

プロパティは CoDaMa クラスのインスタンス変数を指す．プロパティは開発者が独自に定義できる．

3.3.5　インポート，エクスポート

インポートとは，アトリビュートをプロパティに格納する操作を指す．アトリビュートは文字列型 (std::string) であるため，これを整数型や列挙型など適切な型のプロパティに変換する．

エクスポートとは，プロパティをアトリビュートに格納する操作を指す．プロパティは任意の型であるため，これを文字列型 (std::string) に変換する．

3.3.6　CoDaMa エクステンション

CoDaMa エクステンションは，CoDaMa クラスで継承し，これを拡張するためのクラス群である．その多くはアトリビュート毎に用意されており，プロパティやインポート/エクスポートが定義されている．

3.4　実　　例

実際の開発例として，DFG の実行サイクル数をカウントする CoDaMa モジュール「CyclesCounter」の開発手順を説明する．この CyclesCounter は，DFG において演算ノードに割り当てられているコントロールステップの最大値を実行サイクル数とする．CoDaMa フレームワークでは，DFG Node エレメントの `control_step` アトリビュートの値を，DFG エレメントの `cycles` アトリビュートの値にセットすることになる．

Step 1. CyclesCouter のスケルトン（テンプレート）を生成する．

```
$ ruby $codama/tools/make-skelton.rb -unsplit \
  CyclesCounter
$ cd cycles_counter/
```

ここで，`cycles_counter` ディレクトリには，CoDaMa モジュール開発のための幾つかのファイルが自動生成されている．

Step 2. 必要な CoDaMa エクステンションを継承する．ここでは，ControlStep エクステンションと，Cycles エクステンションを利用する．

```
#include <codama/extension/control_step.hpp> //追加
#include <codama/extension/cycles.hpp> //追加
...
class DFGHandler
  : virtual public DefaultDFGHandler<CoDaMa>,
    virtual public extension::Cycles::DFGHandler //追加

...
class DFGNodeHandler
  : virtual public DefaultDFGNodeHandler<CoDaMa>,
    virtual public extension::ControlStep::
    DFGNodeHandler //追加
...
```

Step 3. インポート/エクスポートメソッドを追加する．こ

こでは，DFG Node の ControlStep をインポートし，DFG の Cycles をエクスポートする必要がある．

```
...
void DFGHandler::export_properties()
{
  useCycles::export_properties(); //追加
}
...
void DFGNodeHandler::import_properties()
{
  useControlStep::import_properties(); //追加
}
...
```

Step 4. DFGHandler に count_cycles メソッドを追加・定義する．CoDaMa エクステンションにより，ControlStep や Cycles に関するメソッドが使えるようになっている．

```
...
void DFGHandler::count_cycles()
{
  int cy = 0;
  for (DFGNodeIterator i = nodes(); !i.end(); ++i) {
    DFGNodeHandler* dnh = get_node_handler(*i);
    if (dnh->is_control_step_assigned()) {
      int cs = dnh->get_control_step();
      if (cy < cs) cy = cs;
    }
  }
  set_cycles(cy);
}
...
```

Step 5. ProgHandler に count_cycles メソッドを追加・定義する．このメソッドは，全ての DFGHandler の count_cycles メソッドを実行する．

```
...
void ProgHandler::count_cycles() const
{
  for (SubpIterator si = subps(); !si.end(); ++si) {
    CFGHandler* ch = get_subp_handler(*si)
                     ->get_cfg_handler();
    for (CFGNodeIterator cni = ch->nodes();
         !cni.end(); ++cni) {
      DFGHandler* dh = ch->get_node_handler(*cni)
                       ->get_dfg_handler();
      dh->count_cycles();
    }
  }
}
...
```

Step 6. main 部にて，ProgHandler の count_cycles メソッドを実行する．

```
...
CyclesCounter::ProgHandler* ph;
...
ph = CyclesCounter::parse_prog_xml();
...
ph->import_properties_all();
ph->count_cycles(); // 追加
...
```

これで CoDaMa モジュール「CyclesCouter」のコーディングは終了したので，後はコンパイルし，テスト用の Prog-XML を入力するなどして動作確認を行う．

以上が CoDaMa モジュール開発の大まかな流れである．CoDaMa エクステンションの利用とインポート/エクスポート機構により XML との入出力インタフェースを容易にすることで，開発者はアルゴリズムの実装に専念することが可能となる．

4. 評　　価

現在，我々は CoDaMa フレームワークに基づいた，プロセッサの HW/SW 協調合成システムを開発している．開発に関わっている修士課程の学生 4 名を被験者として，従来の合成システム [9] での開発と比較したとき，従来 1〜2 年を費した HW/SW 分割系における最適化アルゴリズムの追加を，約半年に抑えることができた．その理由として，CoDaMa フレームワーク本来の目的の他に以下のようなものが挙げられた．

- 従来の合成システムでは，自分以外の開発者が書いた膨大なソースコードを理解する作業に数ヶ月の工数が費やされていたが，CoDaMa では各 CoDaMa モジュールの動作さえ理解すれば良いので，その作業が不要になったこと．
- CoDaMa モジュールの開発は，3.4 項に示したようなパターンが提示されているので，それに沿って記述すれば C++ 言語の詳細を知らなくともアルゴリズムを実装できること．

5. む す び

本稿では，ハードウェア/ソフトウェア (HW/SW) 協調合成システムや高位合成システム構築のための，XML をベースとした CDFG (Control Data Flow Graph) 操作フレームワークを提案した．現在開発中のプロセッサ HW/SW 協調合成システムについて，アルゴリズムの実装にかかる工数を半分以下に抑えることができた．なお，CoDaMa ライブラリは [2] にてソースコードを公開中である．

文　　献

[1] Control Data Flow Graph Toolset, http://poppy.snu.ac.kr/CDFG/cdfg.html, 2002.

[2] CoDaMa Wiki, http://www.yanagi.comm.waseda.ac.jp/~kohara/codama/

[3] COINS-project, A compiler infra structure, http://www.coins-project.org/

[4] Y Explorations, Inc., eXCite, http://www.yxi.com/

[5] 濱田 尚宏, 小西 隆夫, 齋藤 寛, 米田 友洋, 南谷 崇, "東データ方式による非同期式回路の動作合成手法の提案," 信学技報, VLD2006-63, pp.71-76, 2006.

[6] オブジェクト指向スクリプト言語 Ruby, http://www.ruby-lang.org/

[7] The SUIF 1.x Compiler System, http://suif.stanford.edu/suif/suif1/, 1994.

[8] The SUIF 2 Compiler System, http://suif.stanford.edu/suif/suif2/, 1999.

[9] N. Togawa, K. Tachikake, Y. Miyaoka, M. Yanagisawa, and T. Ohtsuki, "A SIMD Instruction Set and Functional Unit Synthesis Algorithm with SIMD Operation Decomposition," IEICE Trans. on Information and Systems, Vol.E88-D No.7, pp.1340-1349, 2005.

[10] K. Wakabayashi, "CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification, " IEEE VLSI-TSA International Symposium, pp. 173-176, 2005.

[11] Extensible Markup Language (XML), www.w3.org/XML/