ハイパフォーマンス
コンピューティング 58－2
（1995．10．18）

仕様記述言語VDMによる並列計算機ADENA4シミュレータの実現

杉山 和徳　今村 俊幸　野木 達夫
京都大学工学部応用システム科学教室
〒606-01 京都市左京区吉田本町

並列計算機ADENA4シミュレータをC++等のプログランミング言語より抽象度がより高い仕様記述言語VDMによって集合・写像・列などの数学的概念を用いて簡潔かつ正確に仕様記述し、更にVDMツールの一種であるIFAD VDM-SL Toolboxと C++ Code Generatorを用いて並列計算機ADENA4シミュレータを保守性も含めて効率的に実現した。本論文では、開発対象となった並列計算機ADENA4のアーキテクチャ、VDM言語による並列計算機ADENA4抽象機械の仕様記述、VDM言語ツールのシミュレータ開発への適用について述べる。

Implementation of Parallel Machine ADENA4 Simulator by Specification Language VDM

Kazunori Sugiyama, Toshiyuki Imamura and Tatsuo Nogi
Division of Applied Systems Science, Faculty of Engineering, Kyoto University
Yoshidahonmachi, Sakyo, Kyoto, 606-01, Japan

We have specified Parallel Machine ADENA4 Simulator precisely and concisely using the mathematical concept such as Set, Map, Sequence of Specification Language VDM which is more abstract than Programming Language such as C++, and efficiently implemented it including its maintenance by VDM tools, IFAD VDM-SL Toolbox and C++ Code Generator. In this paper we describe Parallel Machine ADENA4 architecture, VDM specification of Abstract Parallel Machine ADENA4 and application of VDM tools to this development.

1 はじめに

ソフトウェアが大規模化、複雑化し、高い信頼性を要求されるにつれ、VDM[1][7][8][9] や Z などの仕様記述言語を用いて形式的なアプローチにより簡潔性及び正確性などの数学的利点を生かしてのソフトウェア開発がなされる傾向にある。近年、VDM 言語の静的構文検査ツール、インタープリッタによる動的検査ツール、仕様の正当性推論ツール、C++[10] 等のプログランミング言語への自動変換ツールなどが欧州を中心に世界各地で開発されており、ソフトウェア開発において VDM 言語を利用することが効果的で現実的なものとなってきた。

現在、我々の研究室では、これまでに実現してきた ADENA アーキテクチャを改良し、最新のハードウェア技術を適用した ADENA4 の実現に向けて数々の考察がなされ、その基本アーキテクチャが固まりつつある。そして、ADENA4 シミュレータなどによる性能評価により基本アーキテクチャを確定すべき段階にあり、ADENA4 シミュレータを開発中で、その核となる部分が完成したところである。ADENA4 シミュレータ開発においては、微妙に異なる数々のアーキテクチャの ADENA4 システムの性能評価をシミュレータにて容易にできることを狙って C++ などのプログランミング言語より抽象度の高い仕様記述言語 VDM を用いて簡潔かつ正確に仕様記述し、IFAD VDM-SL Toolbox と C++ Code Generator というツールを用いて VDM 言語で仕様記述された ADENA4 シミュレータを C++ プログラムに自動変換してインプリメントする開発手法を取った [2][3][4][5]。これにより、仕様変更に容易にかつ迅速に対応できる ADENA4 シミュレータを効率的に実現することを目標とした。

2 並列計算機 ADENA4

2.1 並列計算機 ADENA4 アーキテクチャ

並列計算機 ADENA(Alternating Direction Execution Nexus Array) は ADEPS[2] と呼ぶ並列計算スキームに基づいて計算を進める並列計算機である。並列計算スキーム ADEPS とは多次元配列データを様々な次元方向から 1 次元配列データに切り分け、その 1 次元配列データごとに並列計算を繰り返し実行するスキームである [1]。

並列計算機 ADENA4 は、L をシステム・サイズを決定するある整数として、1 台のホスト・プロセッサ HP とそのホスト・メモリ hm、2 次元配列構成のローカル・プロセッサ群 $\{\mathrm{LP}[i,j] : i,j = 1,...,L\}$ と各々に付随するローカル・メモリ $\{lm[i,j] : i,j = 1,...,L\}$ 及び各プロセッサ間でデータ交換するためのグローバル・メモリとして機能する 3 次元配列構成のメモリ・バンク配列 $\{mba[i,j,k] : i,j,k = 1,...,L\}$ から構成される。ホスト・プロセッサ HP はホスト・メモリ hm にアクセスでき、各ローカル・プロセッサ $\mathrm{LP}[i,j]$ は自分のローカル・メモリ $lm[i,j]$ にアクセスでき、更に、3 つの計算状態 ($\mathrm{LP}[,/i,j/]$: x方向計算、$\mathrm{LP}[i/,,/j]$: y方向計算、$\mathrm{LP}[/i,j/,]$: z方向計算) に応じて、$\mathrm{LP}[,/i,j/]$ はメモリ・バンク配列要素 $\{mba[l,i,j] : l = 1,...,L\}$ に、$\mathrm{LP}[i/,,/j]$ はメモリ・バンク配列 要素 $\{mba[i,l,j] : l = 1,...,L\}$ に、$\mathrm{LP}[/i,j/,]$ はメモリ・バンク配列要素 $\{mba[i,j,l] : l = 1,...,L\}$ に各々アクセスできる。また、ホスト・メモリ hm, ローカル・メモリ $\{lm[i,j] : i,j = 1,...,L\}$ 及びメモリ・バンク配列 $\{mba[i,j,k] : i,j,k = 1,...,L\}$ 間でのデータ交換も可能となっている。

ADENA4 上での 3 次元偏微分方程式の数値計算においては、その 3 次元計算領域内のある格子点 (i,j,k) 上のデータはメモリ・バンク配列要素 $mba[i,j,k]$ にデータを置いて計算を進めていくことになるが、その 3 次元計算領域が ADENA4 のシステム・サイズ $L \times L \times L$ を越える場合は各メモリ・バンク配列要素 $mba[i,j,k]$ は複数の格子点上のデータを多重して持つことにより計算を進める。計算領域内 $\{(1,...,R_x) \times (1,...,R_y) \times (1,...,R_z)\}$ のある格子点 (x,y,z) 上のデータはメモリ・バンク配列要素 $mba[(x-1)\mathrm{mod}L+1, (y-1)\mathrm{mod}L+1, (z-1)\mathrm{mod}L+1]$ に置かれ、各ローカル・プロセッサ $\mathrm{LP}[i,j]$ は 1 次元配列をなすメモリ・バンク配列要素をアクセスしながら多重度分だけ繰り返して計算を進める。この処理を多重処理と呼ぶ。

2.2 並列言語 ADETRAN4

並列言語 ADETRAN4 は ADENA4 上の並列計算を記述するためのプログラミング言語であり、FORTRAN77 プログラミング言語に ADENA4 固有のプログラム単位、データ構造宣言文、並列動作実行文、並列データ転送文などが追加されている。

2.3 並列中間言語 ADEINTER

ADENA4 シミュレータは ADEINTER と呼ぶ並列中間言語で記述されたプログラムを 1 命令ずつ逐次解釈しながら並列計算機 ADENA4 でのプログラム実行をシミュレートする。並列中間言語 ADEINTER の各命令は 1 つの命令

[1] Vienna Development Method

[2] Alternating Direction Execution of Parallel over Segments

コードと0個以上のオペランドの組から成る。主な命令を以下に示す。

命令形式 {(命令コード, オペランド, オペランド,.....)}	内容
(region,R_x, R_y, R_z)	計算領域 $\{(1,...,R_x) \times (1,...,R_y) \times (1,...,R_z)\}$ の指定
(hm,*VariableID*, *Type*, *VectorOrScalar*, *length*)	HM変数についての情報
(lm,*VariableID*, *Type*, *VectorOrScalar*, *length*)	LM変数についての情報
(mba,*VariableID*, *Type*, *VectorOrScalar*, *length*)	MBA変数についての情報
(load,*RegisterID*, *VariableID*)	Load 命令
(save,*RegisterID*, *VariableID*)	Save 命令
(loadx,*RegisterID*, *VariableID*, *IndexRegisterID*)	インデックス付き Load 命令
(savex,*RegisterID*, *VariableID*, *IndexRegisterID*)	インデックス付き Save 命令
(loadi,*RegisterID*, *VariableID*)	間接 Load 命令
(savei,*RegisterID*, *VariableID*)	間接 Save 命令
(loadix,*RegisterID*, *VariableID*, *IndexRegisterID*)	インデックス付き間接 Load 命令
(saveix,*RegisterID*, *VariableID*, *IndexRegisterID*)	インデックス付き間接 Save 命令
(goto,*offset*)	相対ジャンプ命令
(if,*RegisterID*, *offset*)	条件付相対ジャンプ命令
(bop,$RegisterID_1$, $RegisterID_2$, $RegisterID_3$)	2項演算命令 ($Register_1$ bop $Register_2 \Rightarrow Register_3$)
(uop,$RegisterID_1$, $RegisterID_2$)	単項演算命令 (uop $Register_1 \Rightarrow Register_2$)
(prange,*Direction*, S_α, E_α, S_β, E_β)	並列計算の方向 *Direction* と領域 $\{(S_\alpha,...,E_\alpha) \times (S_\beta,...,E_\beta)\}$ の指定
(pdo)	並列計算をする部分の始まり
(pend)	並列計算をする部分の終り
(greturn)	GSUBROUTINE の終り
手続き呼出し、データ転送関連等の命令は省略	

3 並列計算機 ADENA4 シミュレータ

3.1 並列計算機 ADENA4 シミュレータの構成

並列計算機ADENA4シミュレータとは、ADETRAN4コンパイラによりADETRAN4プログラムからADEINTERプログラムに変換されたものをインタープリッタ形式で実行することにより、ADENA4の動作をシミュレートしながらADEINTER中間言語の各命令の実行回数などの統計情報を収集する並列計算機シミュレータである。

3.2 VDM言語による並列計算機ADENA4抽象機械の記述

並列計算機ADENA4抽象機械は1台のホスト・プロセッサHPとそのホスト・メモリ hm と $L \times L$ 台のローカル・プロセッサ群 $\{LP[i,j] : i,j = 1,...,L\}$ とそのローカル・メモリ $\{lm[i,j] : i,j = 1,...,L\}$ 及び $L \times L \times L$ 台のメモリ・バンク配列 $\{mba[i,j,k] : i,j,k = 1,...,L\}$ から構成される。各メモリ・バンク配列 $\{mba[i,j,k] : i,j,k = 1,...,L\}$ は、計算領域が $\{(1,...,R_x) \times (1,...,R_y) \times (1,...,R_z)\}$ である時、多重処理のため、M_x(x方向多重度)、M_y(y方向多重度)、M_z(z方向多重度)を、それぞれ、$M_x = (R_x - 1)\mathrm{div}L + 1, M_y = (R_y - 1)\mathrm{div}L + 1, M_z = (R_z - 1)\mathrm{div}L + 1$ と置くと、$M_x \times M_y \times M_z$ 個のエリアに分割される。ホスト・メモリ hm の場合はHM変数番号集合と添字集合[3]の直積からその値への写像として、ローカル・メモリ $\{lm[i,j] : i,j = 1,...,L\}$ はプロセッサ番号集合 $(1,...,L) \times (1,...,L)$,LM変数番号集合及び添字集合の直積からその値への写像として、メモリ・バンク配列 $\{mba[i,j,k] : i,j,k = 1,...,L\}$ はメモリ・バンク番号集合 $(1,...,L) \times (1,...,L) \times (1,...,L)$, 多重処理エリア番号集合 $(1,...,M_x) \times (1,...,M_y) \times (1,...,M_z)$, MBA変数番号集合及び添字集合の直積からその値への写像として定義する。ホスト・プロセッサもローカル・プロセッサも n 個のレジスタを持ち、レジスタ番号からその値への写像として定義する。変数、レジスタが取り得る値は、論理値、整数又は実数のいずれかである。ADENA4抽象機械には記号表があり、ここに各変数 / 配列に関する情報を格納しておくことにより実行時に動的に変数・配列がHM変数か、LM変数か又はMBA変数かを判断できるので、各種変数 / 配列へのアクセス命令 (Load/Save関連命令) を共通化できた。

VDM言語によるADENA4抽象機械の仕様記述では、まず最初にADENA4抽象機械内部の状態を表すレジスタ reg, ホスト・メモリ hm, ローカル・メモリ lm, メモリ・バンク配列 mba などのADENA4 stateの定義から行なった。stateの定義に必要なADENA4抽象機械固有のtypeは上の説明より主として 以下に示すものになった。これらのtypeを使ってADENA4 stateが次のように定義された。

[3] スカラ変数の場合は添字は1。

types

DIRECTION	= X | Y | Z | VOID;	— 並列計算方向集合：{X,Y,Z,void}
VariableID	= $\mathbb{N}_1$;	— 変数番号集合：自然数
Index	= $\mathbb{N}_1$;	— 添字集合：自然数
RegisterID	= $\mathbb{N}_1$;	— レジスタ番号集合：自然数
BANK	= $\mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$;	— バンク番号直積集合 $(1,...,L) \times (1,...,L) \times (1,...,L)$
AREA	= $\mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1$;	— エリア番号直積集合 $(1,...,M_x) \times (1,...,M_y) \times (1,...,M_z)$
PROCESSOR	= $\mathbb{N}_1 \times \mathbb{N}_1$;	— プロセッサ番号直積集合 $(1,...,L) \times (1,...,L)$
HostMemory	= $VariableID \times Index \xrightarrow{m} VALUE$;	— ホスト・メモリのタイプ
LocalMemory	= $PROCESSOR \times VariableID \times Index \xrightarrow{m} VALUE$;	— ローカル・メモリのタイプ
MemoryBankArray	= $BANK \times AREA \times VariableID \times Index \xrightarrow{m} VALUE$;	— メモリ・バンク配列のタイプ
REGISTER	= $RegisterID \xrightarrow{m} VALUE$;	— レジスタのタイプ
VALUE	= $\mathbb{B} \mid \mathbb{Z} \mid \mathbb{R}$;	— 値の種類:{論理型, 整数, 実数}
MEMORY	= HM | LM | MBA;	— メモリの種類: {ホスト・メモリ, ローカル・メモリ, メモリ・バンク配列}
TYPE	= INTEGER | REAL | LOGICAL;	— 変数タイプ
SymbolTable	= $VariableID \xrightarrow{m} SymTabDetails$;	— 記号表
SymTabDetails :: *memory*	: *MEMORY*	— メモリの種類
type	: *TYPE*	— 変数タイプ
vectorOrscalar	: VECTOR | SCALAR	— ベクトル変数か、或はスカラ変数か
size	: $\mathbb{N}_1$;	— ベクトル変数のときはその長さ、スカラ変数の時は 1

state *ADENA4-STATE* of

dir	: *DIRECTION*	— 並列計算方向
hm	: *HostMemory*	— ホスト・メモリの状態
lm	: *LocalMemory*	— ローカル・メモリの状態
mba	: *MemoryBankArray*	— メモリ・バンク配列の状態
reg	: *REGISTER*	— レジスタの状態
prog	: *PROGRAM*	— HP と LP のプログラムが混在して格納される
ip-stack	: $InstructionID^*$	— 命令カウンタ・スタックの状態
ip	: *InstructionID*	— 命令カウンタの状態
symTBL	: *SymbolTable*	— 記号表の状態
— 以下、シミュレータ用		
L	: $\mathbb{N}_1$	— 2 次元プロセッサ配列の一列のプロセッサ数
$ira, jra, kra, ina, jna, kna$	: $\mathbb{N}_1$	— 物理プロセッサ番号、バンク番号、エリア番号などの指定用
$l, l\text{-}S, l\text{-}E, m, m\text{-}S, m\text{-}E$	: $\mathbb{N}_1$	— 並列計算領域と現在シミュレートしている 論理プロセッサ番号 (l,m) の指定用

end

prog はプログラム・メモリのことで、シミュレータがシングル・プロセッサ計算機で の実行を前提として作成したので、ホスト・プロセッサ HP とローカル・ プロセッサ $\{LP[i,j] : i,j = 1,...,L\}$ のプログラムはここに混在して格納される。*ip* は命令カウンタのことでプログラム・メモリ *prog* 内で次に実行する命令をポイントする。
ローカル・プロセッサ LP は *dir* なるフラグを持ち、これに X(x 方向計算)、Y(y 方向計算) 又は Z(z 方向計算) をセットすることにより、その時の並列計算方向を指定できるようにしている。シミュレータ内にある l,m は PDO 文の並列制御変数をシミュレートし、 $l\text{-}S$,$l\text{-}E$,$m\text{-}S$,$m\text{-}E$ は並列計算の範囲を示すものであり、 (l,m) が取り得る値は $l\text{-}S \leq l \leq l\text{-}E, m\text{-}S \leq m \leq m\text{-}E$ である。

また、 *dir* と *ira*,*jra*,*kra*,*ina*,*jna*,*kna* そして l,m の値により、論理的[4]なプロセッサ番号 (l,m) はその時シミュレートしている物理的なプロセッサ番号及び多重処理の位置に変換できる。例えば、 *dir* の値が X であるとすると、この時は *jra*,*jna*,*kra*,*kna* が有効となり、各々、 $jra = (l-1) \bmod L + 1$, $jna = (l-1) \mathrm{div} L + 1$, $kra = (m-1) \bmod L + 1$, $kna = (m-1) \mathrm{div} L + 1$ と pdo 命令によりセットされ、これら値がローカル・メモリ或はメモリ・バンク配列をアクセス時に必要なプロセッサ番号、バンク番号、エリア番号を決定するとき参照される。この時は、 (jra,kra) が物理的なプロセッサ番号を、 (jna,kna) が多重処理の位置を示している。
次の VDM 仕様記述には、 prange 命令により並列計算領域が 初期設定され、 pdo 命令と pend 命令によりその並列計算が 単一プロセスにてシミュレートされる様子が記述されている。

[4]「論理的」とはシステム・サイズ L という物理的な制約が無いことを意味する。

```
Prange : DIRECTION × N1 × N1 × N1 × N1 →o ()
Prange (d, s-1, e-1, s-2, e-2) ≜
  (dir := d; — 並列計算方向の初期設定
   l := s-1; l-S := s-1; l-E := e-1;
   m := s-2; m-S := s-2; m-E := e-2); — 並列計算範囲とシミュレート論理プロセッサ番号の初期設定

Pdo : () →o ()
Pdo () ≜
  (ip-stack := [ip − 1] ↷ ip-stack; — 他プロセッサのシミュレートのための復帰位置退避
   cases dir:
     X → (jra := (l − 1) mod L + 1; jna := (l − 1) div L + 1;
          kra := (m − 1) mod L + 1; kna := (m − 1) div L + 1),
     Y → (kra := (l − 1) mod L + 1; kna := (l − 1) div L + 1;
          ira := (m − 1) mod L + 1; ina := (m − 1) div L + 1),
     Z → (ira := (l − 1) mod L + 1; ina := (l − 1) div L + 1;
          jra := (m − 1) mod L + 1; jna := (m − 1) div L + 1)
   end ); — 計算方向別、論理プロセッサ番号 (l, m) から物理的なプロセッサ番号及び多重処理の位置への変換

Pend : () →o ()
Pend () ≜
  (if l < l-E
   then (l := l + 1; m := m; ip := hd ip-stack)
   elseif m < m-E
   then (l := l-S; m := m + 1; ip := hd ip-stack)
   else dir := VOID;
   ip-stack := tl ip-stack); — 並列計算領域内の他プロセッサのシミュレートのための処理
```

ADENA4 抽象機械では、ADEINTER 中間言語の各命令に相当する 各 VDM operation の実行により ADENA4 state が変化する。 ADENA4 シミュレータは ADEINTER プログラムを実行することにより、 ADENA4 抽象機械のレジスタ *reg*, ホスト・メモリ *hm*, ローカル・メモリ *lm*, メモリ・バンク配列 *mba* などの ADENA4 state の内容が どのように変化するかをプログラムしたものである。

```
LoadX : RegisterID × VariableID × RegisterID →o ()
LoadX (r, v, r-i) ≜
  let i = reg (r-i) in
   cases symTBL(v).memory:
     HM → reg := reg † {r ↦ hm (mk-(v, i))},
     LM → (dcl p : PROCESSOR;
           cases dir:
             X → p := mk-(jra, kra),
             Y → p := mk-(kra, ira),
             Z → p := mk-(ira, jra)
           end;
           reg := reg † {r ↦ lm (mk-(p, v, i))}),
     MBA → let R = (i − 1) mod L + 1,
               N = (i − 1) div L + 1 in
           (dcl bank : BANK,
                area : AREA;
            cases dir:
              X → (bank := mk-(R, jra, kra); area := mk-(N, jna, kna)),
              Y → (bank := mk-(ira, R, kra); area := mk-(ina, N, kna)),
              Z → (bank := mk-(ira, jra, R); area := mk-(ina, jna, N))
            end ;
            reg := reg † {r ↦ mba (mk-(bank, area, v, 1))})
   end;
```

上記 ADENA4 シミュレータ内の loadx 命令用の VDM operation *LoadX* では、 loadx 命令によって ADENA4

state の一つであるレジスタ *reg* が レジスタ *r-i* の値をインデックス値として、変数 *v* がHM変数の場合はホスト・メモリ *hm*, 変数 *v* がLM変数の場合はローカル・メモリ *lm*, 変数 *v* がMBA変数の場合はメモリ・バンク配列 *mba* から変数 *v* の値を読みとり、レジスタ *r* の値とするようにレジスタ *reg* の内容を変化させることが記述されている。特に、ローカル・メモリにアクセスする時は *dir* と *ira,jra,kra* の値によって ローカル・プロセッサ 番号が決まり、メモリ・バンク配列にアクセスする時は それらと更に *ina,jna,kna*, レジスタ *r-i* の値によってメモリ・バンク配列のバンク番号と エリア番号が決まる。

3.3 IFAD VDM-SL Toolbox/C++ Code Generator によるインプリメント

ADENA4 シミュレータの開発は、最初に IFAD VDM-SL Toolbox にて ADENA4 シミュレータを VDM 言語にて記述し、その仕様記述の VDM 静的構文検査を行なった。この構文検査で いくつかの簡単な構文上の誤り (バグ) を発見できた。次に、VDM インタープリッタにて、シミュレータの動的検査を VDM 言語レベルで行なった。シミュレータの誤りはこの段階でほぼ捕捉された。そして、C++ Code Generator にて VDM 言語で記述されたシミュレータを C++ プログラムに自動変換した。さらに、VDM 言語がサポートしないシミュレータのファイル I/O 部分を中心に残りを直接 C++ 言語で記述して、全プログラムをコンパイル / リンクして ADENA4 シミュレータを完成させた。この時、VDM 言語による ADENA4 シミュレータの仕様記述は全部で 4,016 行、これから自動生成された C++ プログラムは全部で 27,047 行、そして、直接 C++ 言語で記述したプログラムは 231 行であった。

システム全体の 99 % 以上を VDM 言語により簡潔にかつ正確に記述でき、その VDM 言語による仕様記述はシステム仕様書としても有効で、更にそれは IFAD VDM-SL Toolbox にて VDM 言語レベルでテストでき、テストが終っている VDM モジュールは C++ Code Generator により C++ プログラムに自動変換された。自動変換された C++ プログラムは再度同じテストする必要が無かった。システム全体の約 99 %[5] を VDM 言語で仕様記述でき、VDM 言語から C++ 言語へのコード・ジェネレータを利用することにより、抽象レベルの高いシステム開発が可能となり、開発効率は非常に高かった。実際、仕様決定の作業量を除けば、実質約 2 人月の作業量で ADENA4 シミュレータを開発できた。また、ADENA4 アーキテクチャ及び ADEINTER 中間言語の仕様を VDM 言語にて簡潔かつ正確に記述できた。また、現在のところ仕様変更にも柔軟に対応できている。

4 今後の予定及び課題

ローカル・プロセッサがベクトル・プロセッサであるタイプの ADENA4 アーキテクチャ も考察されている。今後、このタイプの ADENA4 シミュレータも実現したい。その実現に関しては、今回開発したシミュレータと多くの共通点を有しているので短期間で実現できると見通している。

また、今回インプリメントした ADENA4 シミュレータはシングル・プロセッサ計算機での実行を前提として作成したが、VDM++ 言語と呼ばれるオブジェクト指向と並列処理の概念を VDM 言語に導入した仕様記述言語によって書き直すことにより、ソフトウェア工学的により洗練された、マルチ・プロセッサ計算機での実行を前提とする ADENA4 シミュレータを今後インプリメントしてみたい。また、OMT[6] などのオブジェクト指向分析 / 設計技法も導入して、ソフトウェア工学的により優れた並列計算機シミュレータのインプリメント技法を探ってみたい。

参考文献

[1] T. Nogi: Promising Data Parallel Environment — ADEPS, ADETRAN, ADENA —, *Proc. of the First Aizu Int. Symposium on Parallel Algorithms/Architecture Synthesis*, pp.45–53, IEEE Computer Society, 1995.

[2] The VDM-SL Tool Group: The IFAD VDM-SL Language, Technical Report, IFAD, 1994.

[3] The VDM-SL Tool Group: Users Manual for the IFAD VDM-SL Toolbox, Technical Report, IFAD, 1994.

[4] The VDM-SL Tool Group: User Manual for the IFAD VDM-SL to C++ Code Generater, Technical Report, IFAD, 1994.

[5] The VDM-SL Tool Group: The VDM C++ Library, Technical Report, IFAD, 1994.

[6] I. P. Dickinson: Typesetting VDM with VDMSL macros, Technical Report, NPL, 1991.

[7] C. B. Jones: Systematic Software Development using VDM -2nd ed., Prentice-Hall International, 1990.

[8] C. B. Jones and R. C. F. Shaw(eds): Case Studies in Systematic Software Development, Prentice-Hall International, 1990.

[9] D. Andrew and D. Ince: Practical Formal Methods with VDM, The McGraw-Hill International, 1990.

[10] B. Stroustrup: The C++ Programming Language -2nd ed., Addison-Wesley, 1993.

[5] 残り約 1 % は直接 C++ プログラム言語でプログラミングした

[6] Object Modeling Technique