動的コンパイル可能な並列言語の設計と実現

児　玉　靖　司†

フォールトトレラントシステムを記述するためのプログラミング言語として，並列言語 SPL を設計し，実現した．SPL は，大規模分散環境から，オペレーティングシステムのマイクロカーネルまで，さまざまな並列計算システムを統一的に記述することができる．さらに，動的コンパイルを可能とし，柔軟性のあるプログラミングをすることができる．処理系はトランスレータ方式とし，20 個の C 言語関数を呼出す C 言語プログラムとして出力するため，ソースコードレベルでさまざまな並列計算システムへ可搬性の高いものとなっている．本論文では，SPL の設計概念，実現の仕組みを説明し，プログラミングの応用例を紹介した後，評価，議論を行う．

Design and Implementation of a Parallel Language with Dynamic Compilations

Yasushi KODAMA†

We have designed and implemented a parallel language called SPL for developing of the fault tolerant systems. Using SPL, various parallel calculation systems from a large-scale distributed environment to the micro-kernel of the operating system can be described. In addtion, a flexible programming is enabled using dynamic compilations. Bacause this processing system assumes the translator method and outputs the program codes that call only 20 functions of C language, it can be executed with high portability. In this paper, we explain the design concept of SPL, the examples of programming and the system evaluation.

1. はじめに

フォールトトレラントシステムを構築するソフトウエアの機能の一つとして，実行時に，障害を取り除き，新しい機能を追加することが必要である．この処理は，障害が発生していないソフトウエア部品で行われる．ソフトウエアが実行中に，自身の一部分を更新するために，まず，ソフトウエア全体が十分に部品化されている必要がある．各部品が並行に処理され，一つの部分が，他の部分を更新する．例えば，2 つの部品に対して，実行権を交互に与えることにより，互いに他方のプログラムを，データとして更新することを考えることができる．図 1 のように表すことができる．2 つの部品に対して，各々 `ProcA`，`ProcB` というプロセスが割り当てられると仮定すると `ProcA`，`ProcB` は適当なコンテキストスイッチにより実行権が与えられ，実行権がある間，他方のプロセスに割り当てられている部品を更新することができる．さらに，実際には，実行するプロセスは，更新途中に例外を発生する可能性があるため，実行中のプロセスを監視する処理も必要である．よって，ソフトウエア全体の部品化に加え，各部品を，自動的にコンテキストスイッチを行うなど並列に処理することができると都合がよい．

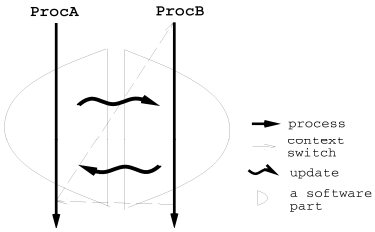

図 1　2 つのソフトウエア部品
Fig. 1　Two software parts

最近では，実行中のプログラムを更新するために，自己反映計算 (リフレクションパターン) を用いて，他方をメタオブジェクトとし，自身を更新する処理が考案されている[2)]．さらに，各オブジェクトを並列オブジェクトとすることにより，自由度が高い更新処理を考えることができる．別の方法として，永続オブジェクトとして 2 次記憶に永続化した後，更新することも考えることができる．

我々は，以上の考察から，実行時に，自律的に，ソフトウエアを更新するためには，以下の 4 点の特徴

† 南山大学数理情報学部
Faculty of Mathematical Information Science, Nanzan University

を持つことが必要であると考える．

(1) ソフトウエア全体が十分に部品化されている．
(2) 各部品が独立に，並列に実行可能である．
(3) 各部品は，さまざまなシステムで実行可能で，統一的記述がなされている．
(4) 実行時に，部品を更新することができる．

(1) ソフトウエアの部品化は，一部分が，他の部分を更新するために，各部分が独立していることを保証する．(2) 各部品が，並列に処理されるべきである理由は，上記で述べた．(3) あらゆるシステムで実行可能である必要がある理由は，更新ソフトウエアが，自律的に処理をするときに，部品を他のシステムから持って来たり，また，他のシステムへ持って行くために必要であるためである．(4) は，更新を自律的に処理するために，記述するプログラミング言語の一つの機能として導入することが必要であることを示す．自己反映計算を導入することも考えることができるが，本論文では，記述言語が，主にフォールトトレラントシステムを記述対象にしている点，(1)，(2)，(3) の特徴との関係を考慮し，動的コンパイルを導入する．さらに，基本ソフトウエアを実現することができるシステム記述言語とするために，実行効率も考慮する．

本論文では，上記 4 点の特徴を持つことを目標とし，フォールトトレラントシステムを記述するためのプログラミング言語として，並列オブジェクト指向モデルに従った並列言語 SPL(a Simple Parallel Language) を提案する．最近では，PVM/MPI を用いた PC クラスタなどの大規模分散環境から，スレッドを単位とした並列計算システム，オペレーティングシステムのマイクロカーネルまで，さまざまな並列計算システムがある．並列計算の概念は抽象的に表現でき，並列単位となるプロセスや，スレッド間の同期処理は，並列オブジェクト指向モデルを採用することにより各システム間で統一的に記述することが可能である．よって，ソースコードレベルで高い可搬性がある，並列オブジェクト指向モデルに従った並列言語とした．

以下，2 節で並列プログラミング言語 SPL の概要を示し，3 節で，言語仕様を説明し，4 節で応用例を示す．5 節で設計方針を説明し，6 節で評価を行う．7 節で議論をし，最後に 8 節で今後の課題を述べる．

2. 概　　要

SPL は，フォールトトレラントシステムを記述するためのプログラミング言語である．並列計算モデルを設定し，抽象的モデルに従った記述言語を設計することによって，各システム間で統一的な記述をすることができる．

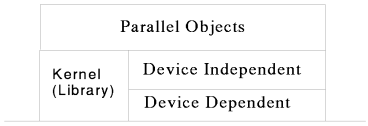

図 2　SPL の構成
Fig. 2　The construction of SPL

我々は並列オブジェクト指向モデル ABCM/1[6] を選択した．その主な理由は以下の 3 点である．

- 単純な構成要素からなる．
- 各オブジェクトで独立したメモリモデルを前提としている．
- ABCL/1 を始めとした実際の実現例がある．

単純な構成要素からなる点は，対応する記述言語で記述する際に細かな指定が可能となり，プログラムを効率良く実行することができる．各オブジェクトが独立したメモリモデルを前提としているため，対象となる並列計算システムが，共有メモリシステムでも，非共有メモリシステムでも，両システムに対して適用が容易である．逆に，共有メモリモデルを前提とする場合は，非共有メモリモデルシステムへの実現が困難で，効率がよくないと考える．SPL はシステム記述言語として，マイクロカーネルとその周りの記述も対象とし，効率よいシステムを実現できることを目的とするため，単純な構成要素からなることも必要である．本論文では，さらに言語仕様として，動的コンパイル機能を導入した．動的コンパイルにより，自己反映計算と同様の柔軟性の高い記述が可能となる．4 節では，動的コンパイルを利用した応用例を紹介する．

SPL は，ソースコードレベルでさまざまなシステムへの可搬性が高い並列言語とするため，C 言語へのトランスレータとした．20 個の関数を呼出すソースコードを出力する．これら関数のみ各対象システム上へ移植すれば，SPL で記述したプログラムを実行することができる．実際には，20 個の内，11 個の関数は，セマフォ，バッファを用いて処理する関数であるため，デバイス非依存である．セマフォ，バッファをもった Primitive Software 上へ移植する場合は，9 個のデバイス依存関数のみを新たに実現すればよい．

SPL の構成を図 2 に示す．最も低レベルとして Primitive Software があり，PVM/MPI，スレッドライブラリなど既存の並列計算システムを指す．マイクロカーネルを構成する場合は，スレッドまたは，プロセスを生成するスケジューラがここに位置する．Primitive Software 上で，Kernel(Library) としてデバイス非依

存の部分と，デバイス依存の部分がある．SPL によるプログラムは，最も上位レベルの Parallel Objects(並列オブジェクト群) である．

3. 言 語 仕 様

SPL は，ABCL/c+[1] の後継に位置するものである．本節では，ABCM/1 で提案された点など，重複する部分は除き，差分のみを説明する．

3.1 オブジェクト定義

SPL のオブジェクトは，以下のように定義する．

```
start {
    var 変数名 = 初期値; ...
    var 変数名 =
      [object オブジェクト名
        delegate-to 委譲先
        state var 識別子 = 初期値; ...
        script
          (=> メッセージパターン {
            var 識別子 = 初期値; ...
            文並び; })
            . . .
          ];
    文並び;
}
```

start { から } までの部分にプログラムを記述する．この部分は，最初に起動される main 並列オブジェクトの一つのメソッドを示す．メソッドは，var から始まる宣言部と，文並びからなる．宣言部は var 変数名 = 初期値 と記述し，変数名を名前とする変数を宣言する．この宣言は，参照される前であればどこに宣言してもよい．各変数の型は，初期値を用いて型推論される．初期値には，定数，オブジェクト式など式を記述する．新たに，オブジェクトを一つ生成する場合は，宣言部で行う．メソッドの本体である文並びは，メッセージ受渡し文，select 文を除き，C 言語と同様の記述をすることができる．変数への代入は，演算子 := を用いる．

オブジェクト定義は，3 つの部分 delegate-to 部，state 部，script 部からなり，各々委譲先オブジェクト指定，状態定義，メソッド定義をする．状態定義，メソッド中での変数宣言は，初期値から型を推論する☆．

同様の記法として，interrupt { ではじまり，} までを記述することができる．この部分では，割り込み手続きを記述するが，オブジェクトを新たに生成することはできない部分である．マイクロカーネルを構成する際には，この部分がハードウエアから直接割り込まれるよう設定する．

3.2 メッセージ受渡し

ABCM/1 で提案された，過去型，現在型，未来型のメッセージ受渡し型を定義することができ，select 文を用いて，休眠モードになることができる．通常モードのメッセージ受渡しのみで，速達モードのメッセージ受渡しは採用しない☆☆．

送り先オブジェクトを A，返答の宛先オブジェクトを B，未来変数を C とし以下のように記述する．

- 過去型
  `[A <= メッセージ];`
  `[A <= メッセージ @ B];`
- 現在型
  `[A <== メッセージ];`
  `[A <=== メッセージ];`
- 未来型
  `[A <= メッセージ $ C];`
- セレクト文

```
[select {
  (=> メッセージパターン {
    文並び; }) ... }];
```

過去型，未来型のメッセージ受渡しは，ACBM/1 で定義された記述と同様である．現在型は，効率を高めるために，NowQueue を生成しない呼出し形式を追加した．NowQueue は，ABCM/1 では，メッセージ受渡し先オブジェクトを特定するために用いるが，実際のプログラミングでは，メッセージ受渡し先からの返答メッセージしか受渡されない保証がある場合があり，[A <=== メッセージ] を用いることができる．実行効率を考慮する場合に有効である．より基本的な過去型，select 文を多用することにより，効率よいプログラミングをすることができる．

3.3 型 推 論

値，変数の型は，ポインタを除き，C 言語と同じ型を用いることができる．配列は，基本型の一次元配列を定義することができる．メモリのあらゆる位置を指すことができるポインタは，オブジェクト毎に独立したメモリモデルを仮定した並列計算モデルに反するので廃止した．オブジェクトを識別する型は，oid 型という．oid 型の変数は，各オブジェクト固有の情報を内部に持つ ObjectSlot を常に指している．Object-

☆ メッセージパターン部では，型を宣言しなければならない．

☆☆ 割り込み手続きを記述する場合は，interrupt {} ブロックを用い，通常モードのメッセージ受渡しを記述する．

Slot は，オブジェクト生成/複製時に，自動的に生成される．

変数宣言部，メソッドの文並び中の式，文は，Hindly/Milner 型推論アルゴリズム[5] を用いて型推論する．各変数は，副作用のある変数であるため，単相型となる．メッセージの受け口であるメッセージパターン部は，実行時にどの型の値を受理できるかを，宣言する意味で，型宣言をしなければならない☆．

型推論の詳しい説明は，本論文の趣旨ではないので省略する．

3.4 オブジェクト生成と複製

オブジェクト生成は，オブジェクト定義で述べた `[object` $Name$ `...]` により，生成，起動される．$Name$ は，オブジェクト名を示す．

オブジェクト複製には，演算子 `clone` を用いる．以下の 2 通りの方法がある．

(1) 上記のオブジェクト生成で起動されたオブジェクトを複製し，新たに起動する．

`clone` $ObjectID$

(2) プログラムを文字列として受取り，新たに生成，起動する．

`clone` $(ObjectName, Program)$

$ObjectID$ は，`oid` 型の変数，$ObjectName$，$Program$ は，各々オブジェクト名，プログラムソースであり，文字列として受渡す．(1) は，`[object` $Name$ `...]` により生成されたオブジェクトを複製し，起動する．(2) は，文字列としてのオブジェクトの状態，メソッド定義を含んだオブジェクトを動的生成 (コンパイル) し起動する．動的コンパイルにより，動的にさまざまなオブジェクトを生成，起動することができ，柔軟性の高いプログラミングが可能となる．

どちらの，複製も実行時 (動的) に，オブジェクトを複製，起動するが，本論文では，上記 2 つを区別するため，(1) を**動的生成**，(2) を**動的コンパイル** という．演算子 `clone` の返り値は，`oid` 型であり，複製されたオブジェクトの ObjectSlot を指す値を返す．

3.5 呼 出 し 型

SPL は，システム記述言語である．より低レベルの部分にアクセスするために呼出し型のメッセージ受渡しを記述することができる．低レベルのソフトウエアは，ルーチンオブジェクトとして記述することが前提である．C 言語の関数呼出しと同様の記述をする．例えば，`Primitive::printf("Hello, World!\n");` と記述し，`Primitive` がルーチンオブジェクト名，`printf` がメソッド名である．

4. 応 用 例

本節では，SPL を用いてシステム記述をした例を紹介する．特に，動的コンパイルを用いた例を中心に述べる☆☆．以下の 4 点の例を示す．

- 哲学者の食事問題
- デザインパターン
- プラグインと疑似マイグレーション
- その他

4.1 哲学者の食事問題

哲学者の食事問題を SPL で記述した例を示す．付録に，(A) オブジェクトの動的生成，(B) オブジェクトの動的コンパイルを用いたプログラムを示す．ただし，プログラムの実行の様子を出力するプログラム片 (`Primitive::printf` 呼出しなど) は，省略する．

`start` { から } までが，プログラムである．3 種類の並列オブジェクトを宣言する．部屋 `Room`，哲学者 `Philosopher`，フォーク `Fork` で，部屋は 1 個，哲学者，フォークを 各 3 個生成する．哲学者は，以下の処理を繰り返す．

- 部屋に入り，
  `[Room <=== :Enter];`
- 考え，
- 左右のフォークを掴み，
  `[RightFork <=== :Get];`
  `[LeftFork <=== :Get];`
- 食事をし，
- 左右のフォークを置き，
  `[RightFork <=== :Put];`
  `[LeftFork <=== :Put];`
- 部屋から出る．
  `[Room <=== :Exit];`

哲学者が，部屋に入り，フォークを掴む時に，同時に同じ側のフォークを持たないように，以下の 3 点の工夫した．

- 部屋オブジェクト (`Room`) に `:Enter` メッセージを送る．
- 左を最初にもつ哲学者と，右を最初に持つ哲学者を区別する．
- 部屋オブジェクト (`Room`) に `:Exit` メッセージを送る．

☆ メッセージパターン部の各変数の型を推論するためには，各オブジェクト間のメッセージ受渡しの流れを，データフォロー解析と同様の手法を用いて解析しなければならない．

☆☆ 以下のプログラムでは，動的コンパイル以外の実行を効率的に処理するために，NowQueue を起動しない，現在型メッセージ受渡し `<===` を用いる．

哲学者オブジェクト Philosopher は，一つのメソッドを終了することなく，常に，select 文で待機するので，メッセージキューを消費し続ける．実際の実行では，メッセージキューを消費しつくした時点で停止するが，メモリを割り当てることができる間，実行を続ける．

(A) は，各並列オブジェクトを最初に一つ生成し，演算子 clone により動的生成する．(B) は，全ての並列オブジェクトを動的コンパイルにより複製する．動的コンパイルを用いると，主に以下の 2 点の変更が可能となる．

(1) コンストラクタに相当する部分を文字列の変更で記述する．
(2) メソッドの追加/削除を，直接プログラム (文字列) の変更で記述する．

この例では，(1) を用いて，各オブジェクトの名前，識別番号を設定した．文字列操作は，Primitive::sprintf を呼出し型により呼出した．6 節 評価で，このプログラムの実行結果を示し評価するため，哲学者オブジェクトは 3 個とした．

4.2 デザインパターン

実行時にソフトウエアを更新するためには，ソフトウエア全体が十分に部品化されている必要がある．ソフトウエアの部品化には，デザインパターン[3] を用いて設計することが有効である．さらに，各デザインパターンは，構成要素の独立性が重視されるため，並列に実行する並列オブジェクトをその構成要素とすれば，より独立性の高いソフトウエア部品となることが期待できる．デザインパターンを活用して設計されたフォールトトレラントシステムを SPL を用いて記述することは，有効である．

例えば，デザインパターンの Command パターン，Singleton パターンを活用して，ウインドウシステムのメニューを記述する．並列オブジェクト menu は下図のように数個の項目を持つ簡単なメニューのプルダウン，ポップアップを司る役目を果たすとし，各項目には，項目番号が付いていると仮定する．

Menu
Create
Delete
Append
Switch
Exit

各メニュー項目は，動的に変更可能とするために，Command パターンを用い，独立した並列オブジェクトとする．これら並列オブジェクトにメッセージを振り分ける並列オブジェクトは，同時には一つしか存在しないので，Singleton パターンを用い，並列オブジェクト menu の内部に保持する．よって，並列オブジェクト menu を SPL で記述すると以下のようになる．

```
[object menu
  state var ItemNumber=0 var Singleton=Me
  script
    (=> :Move int X int Y {
      if( メニューが画面に表示されている ) {
//      マウスカーソルの Move イベントにより，
//      :Move メッセージが受渡され，
//      メニュー上で項目が選択される処理．
//      変数 ItemNumber に項目番号を設定する．
      }
      })
    (=> :PullDown int X int Y {
      ItemNumber:=0;
//    メニューを画面に表示する (プルダウン)．
      })
    (=> :PopUp {
      if( メニューが画面に表示されている ) {
//      メニューを画面から消去する
//      (ポップアップ)．
//      選択された ItemNumber を Singleton
//      が示す並列オブジェクトへ受渡す．
        [Singleton <= ItemNumber];
      }
      })
    (=> :SwitchMenu string ob {
      Singleton := clone("MenuItem",ob);
      })
  ];
```

並列オブジェクト menu は，状態変数として，マウスポインタが指している項目番号を持つ ItemNumber，選択された項目に，対応する Command を振り分ける並列オブジェクトを示す Singleton を保持し☆，:Move，:PullDown，:PopUp，:SwitchMenu のいずれかのメッセージを受理することができる．:PullDown メッセージにより，画面にメニューを表示し (プルダウン)，マウスの Move イベントにより受渡される :Move メッセージを受理することができ，各メニュー項目を移動する．最後に，:PopUp メッセージを受渡され，受理

☆ 変数 Singleton は，Me で初期化されているが，ここでは，最低一つの SwitchMenu メッセージを受理した後と仮定する．

すると，画面からメニューを消去する (ポップアップ). 同時にマウスポインタが選択している項目番号を示す `ItemNumber` を `oid` 型の変数 `Singleton` が示す並列オブジェクトへ受渡す．変数 `Singleton` が示すオブジェクトは，選択された項目番号と，各 Command を対応づける役目をおい，`:SwitchMenu` メッセージにより，動的に切替えることができる．

動的に，自由に切替えるために，`clone` 演算子を用いて，動的コンパイルを指示し，並列オブジェクトを生成する．Command パターンの各 Command を動的に切替える役目をおう並列オブジェクトを自由に生成することができる．

Command パターンと，Singleton パターンを活用して，ウインドウシステムのメニューを設計し，SPL により記述することができた．特に，デザインパターンの中でも Command パターンのように，動的に構成要素を切替えるデザインパターンは，State パターン，Strategy パターンなど多くのパターンがあり，SPL の動的コンパイルを用いて記述することが有効である．

4.3 プラグインと疑似マイグレーション

プラグイン (Plug-In) は，ソフトウエアの実行時に，一部分 (ソフトウエア部品) を，ネットワークからダウンロードし，追加/更新する機能である．SPL の動的コンパイルを用いることにより簡単に記述することができる．動的コンパイルを用いる場合は，ソフトウエア部品を並列オブジェクトとして記述し，メッセージの文字列引数としてオブジェクト定義そのものを受渡すことができる．

オブジェクトのマイグレーションも同様に，オブジェクト定義をメッセージとして受渡すことができる．しかし，このオブジェクト移動は，状態を「保持したまま」，移動するのではなく，初期状態を保持するオブジェクトを移動先で生成する．我々は，動的コンパイルを用いたオブジェクトの移動を**疑似マイグレーション** という．状態を「保持したまま」，オブジェクトを移動させるには，移動前の状態を取り出し，文字列に変換して，受渡さなければならない．

4.4 そ の 他

動的コンパイルを用いることにより，フォールトトレラントシステムを構築するソフトウエアの機能の一つとして，実行時にソフトウエアを更新することができる．最近では，自己反映計算 (リフレクションパターン) を用いて同様にソフトウエアを更新する方法が提案されている[2)]．自己反映計算では，例えば，フォールト (故障) であるソフトウエア部品を，新たな部品に交換するのではなく，その状態を「保持したまま」更新する．しかし，フォールトトレラントシステムでは，フォールトである部品を，取り除き，新しい部品と交換する方が，効率がよく，フォールトを完全に取り除くことができると考える．よって，自己反映計算と同様の処理を効率よく処理することができる動的コンパイル導入した方が良いと考えた．我々は，この機能を疑似自己反映計算という．

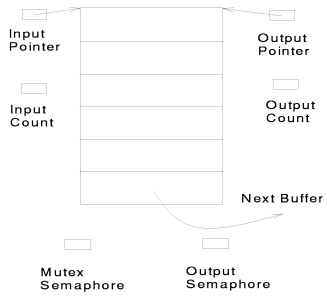

図 3 バッファ
Fig. 3 Buffer

その他，SPL を用いてさまざまなシステムを記述している．

5. 設 計 方 針

図 1 の Kernel(Library) 部と，言語トランスレータを設計，実現すればよい．並列オブジェクト指向モデルでは，オブジェクトに付随するメッセージキューを通して，メッセージ受渡しを行い，並列オブジェクト間の同期処理を行う．本節では，メッセージキュー周りを中心に説明する．

5.1 無限に長いメッセージキュー

並列オブジェクト指向モデルでは,「無限に長い」メッセージキューを持つことを前提としている．無限に長いメッセージキューを実現するために，一定個のメッセージを格納するバッファ(図 3) を集めたバッファプールを用意する．次に，バッファプールからバッファを一つ取り出す．このバッファには，メッセージの出力を管理するセマフォ(OutputSemaphore)，相互排除のためのセマフォ(MutexSemaphore) が付随している．バッファへの入力 (メッセージの受取り) は,「無限に」行うことができるので，セマフォによる管理はしない．このバッファにメッセージが一杯になったら，バッファプールより新たにバッファを一つ取り出し，次のバッファへのポインタを最後の要素として格納する．

5.2 オブジェクトの識別

オブジェクトを識別するために以下に示す要素を保持する ObjectSlot を動的に生成する．メッセージの同期処理を扱う値は，図 3 で示した値である．

Identifier	識別子 (32 ビット)
Name	名前 (文字列)
InputPointer	メッセージ入力 バッファへのポインタ
InputCount	メッセージ入力 バッファ中の位置
OutputPointer	メッセージ出力 バッファへのポインタ
OutputCount	メッセージ出力 バッファ中の位置
SelectPointer	select 文待機中 のメッセージ出力 バッファへのポインタ
SelectCount	select 文待機中 のメッセージ出力 バッファ中の位置
OutputSemaphore	メッセージ出力の ためのセマフォ
MutexSemaphore	メッセージキュー 相互排除のためのセマフォ
Entry	オブジェクト起動のため のエントリポイント
Misc	補助のポインタ

oid 型の値は，ObjectSlot を指す値である．ObjectSlot の項目は，大別すると，

(1) オブジェクトを識別するための値

(2) メッセージによる同期処理を扱う値

の 2 種類を保持している．

DequeueMessage	メッセージの取り出し
StartTo GetVariable	メッセージから変数 取り出しの開始
GetVariable	メッセージから変数 の取り出し
ClearList	メッセージの解放
AnalizeMessage	受理するメッセージの解析
SendMessage	一つのメッセージの送信
CreateNewObject	新しい並列オブジェクト の生成
ConnectObject	メッセージ送信のための オブジェクトコネクト
DisconnectObject	コネクト解放
InitializeObject	オブジェクト初期化
LoopingSelect Form	select 文での繰り返し を処理する関数
AnalizeSelect Message	select 文で受理する メッセージの解析
EndSelectMessage	select 文の終了
ClearSelectList	select 文での メッセージ解放
StartSelectForm	select 文の開始
CreateNowQueue	NowQueue の生成
KillNowQueue	NowQueue の消去
CompileAndCreate NewObject	並列オブジェクトの 動的コンパイル
Suicide	オブジェクトの自殺
SetTarget	委譲メッセージを生成する ためターゲット オブジェクトを変更 する関数

5.3 中 間 関 数

処理系として，ソースコードを C 言語へ変換するトランスレータ方式とした．メッセージキュー，ObjectSlot を扱う C 言語関数 20 個を Kernel(Library)(図 2) として定義し，トランスレータは，これら関数を呼出すプログラムを出力する．

中間関数を大別すると，

(1) 通常モードのためのメッセージ処理

(2) select 文のための処理

(3) NowQueue のための処理

(4) その他，補助関数

の 4 種類となる．

5.4 C 言語関数への変換

並列オブジェクト一つを形成する (トランスレータより出力された) C 言語の関数は，以下のようになる．

```
メッセージパターンの宣言;
ObjectSlot の宣言;
_オブジェクト名 ( char *argv[] )
{
 PASSING_TYPE TargetObjectPointer;
 oid Sender;
 oid SenderOfSelect;
 oid Me = (oid)(argv[0]);
 oid Mother = (oid)(argv[1]);
 _InitializeObject( argv, Me, TRUE );
 while( TRUE ) {
  message __Message; message __MessageP;
  int __From = 0; int __FromInSelect = 1;
  __Message = __MessageP =
   _DequeueMessage( NORMAL, Me, &Sender );
__ConstraintLooping:
  switch( _AnalizeMessage( __Message,
   MESSAGE_PATTERN_COUNT, _MessagePatternArray,
   __From ) ) {
   case 0: {
# ifdef DEBUG_OBJECT
    printf( "*** [%s] message 0 from %s.\n",
     Me->Name, Sender->Name );
# endif /* DEBUG_OBJECT */
    _Suicide( Me, FALSE );
    break;
   }
   case 1: {
    メソッド本体;
   }
   . . .
   default : {
# ifdef DEBUG_OBJECT
    printf( "No acceptable message.\n" );
# endif /* DEBUG_OBJECT */
    break;
   }
  }
__Exiting:
  _ClearList( __MessageP );
 }
}
```

各並列オブジェクトは，「_オブジェクト名」 の関数一つとなる．受理することができるメッセージを示すメッセージパターンは，配列として，「メッセージパターンの宣言;」に宣言されている．関数の最初で `Me`，`Mother` 疑似変数の宣言および設定，`Sender`，`SenderOfSelect` 疑似変数の宣言をする．`_InitializeObject` を呼出し，メッセージキューの生成を行う．次に，`while(TRUE) ...` の繰り返しを行う．繰り返しでは，以下の処理を繰り返す．

(1) 関数 `DequeueMessage` によりメッセージを一つ取り出す．
(2) 関数 `AnalizeMessage` によりメッセージが受理できるかどうか，解析する．
(3) メッセージを受理する場合は，対応するメソッドを実行する．
(4) メッセージが受理できない場合は，(6) へ進む．
(5) メソッド終了後に受理したメッセージを消去する．
(6) (1) へ戻る．

5.5 動的コンパイル

動的コンパイルは，中間関数 `CompileAndCreateNewObject` により実現する．この関数を，OS の機能として提供されれている動的リンクライブラリを用いて記述する．実際の記述としては，C 言語により 150 行程度であった．

6. 評　　価

SPL は，処理系としてトランスレータ方式とし，さまざまな並列計算システムへ可搬性の高いプログラミングが可能である．さらに，システム記述言語として，効率の高いシステムを記述することができる．現在，大規模分散環境として PVM を用いた PC クラスタ，p スレッドライブラリおよび，やゑ[4] マイクロカーネルのスケジューラを図 2 中の Primitive Software とし，設計，開発している．本節では，それらの実行効率を示すために，実行結果の評価を行う．

6.1 哲学者の食事問題

4 節 応用例で紹介した哲学者の食事問題を p スレッドライブラリを用いた実現での実行結果が図 4 である．哲学者オブジェクト 3 つについて，各 1000 回食事をした時の時間を計測した．横軸が食事の回数，縦軸が秒単位の時間である．PC には，CPU が Pentium MMX 233MHz，OS は，FreeBSD を用いた．上，中，下段の 3 段のグラフに対して，下段，上段は，各々付録 (A) 各オブジェクトを動的生成したプログラム，付録 (B) 各オブジェクトを動的コンパイルしたプログラムの実行結果である☆．中段は，各オブジェクトの最初の一つだけ，動的コンパイルし，他のオブジェクトは，動的生成により複製したプログラムの実行結果である．各段とも，3 つの並列オブジェクトの実行時間をプロットしたが，重なっている．

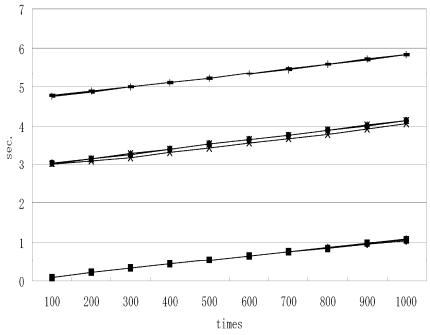

図 4 哲学者の食事問題の実行結果
Fig. 4 Execution time of dinning philosopher's problem

3 種類のプログラム共に，1000 回の食事まで各哲学者オブジェクトが飢餓状態を起こすことなく，均等に食事ができ，回数に比例して実行していることがわかる．動的コンパイルしたオブジェクトの個数分だけコンパイルのための時間がかかっている．一旦，コンパイルし，動的生成したオブジェクトは，静的に定義されたオブジェクトと同程度に実行している．

上段のプログラムは，部屋 (`Room`) オブジェクト 1 個，フォーク (`Fork`) オブジェクト 3 個，哲学者 (`Philosopher`) オブジェクト 3 個，計 7 個，中段のプログラムは，部屋オブジェクト，フォークオブジェクト，哲学者オブジェクト各 1 個，計 3 個のオブジェクトを動的コンパイルしている．

6.2 PC クラスタを用いた台数効果

1800 個と，900 個の整数値を並列オブジェクトを用いてソートするプログラムを作成し，1 台から 12 台 の PC を用いた PC クラスタ上で実行し，実行時間を計測した．計算の様子を図 5 に示す．各 PC は，CPU が Pentium III 750MHz，OS は，Linux を用いた．

`main` 並列オブジェクトに，N 個の要素を持つ配列を用意し以下の処理をする並列オブジェクトを 1 つ生成する．`main` 並列オブジェクトには，生成される並列オブジェクトより，ソートされた要素を持つ配列

☆ 付録 (A)，(B) のプログラムは，実行時間を計測するためのプログラム片は省略した

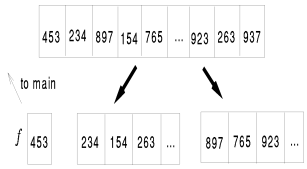

図 5 整数値配列のソート
Fig. 5 Sorting of an integer array

が受渡され，集められる．

最初に，M 個の要素を持つ配列をメッセージとして受渡され，最初の要素 f を取り出し，f 未満の値を集めた配列，f 以上の要素を集めた配列に分割する．次に，2 つの配列を別々に並列にソートするため，2 つの並列オブジェクトを `clone` 演算子により複製し，上記 2 つの配列を別々にメッセージとして受渡す．f は，`main` 並列オブジェクトへ受渡す．最初に受渡される要素の個数 M が，1 〜 3 の場合は，ソートし，`main` 並列オブジェクトへ受渡す．各要素は，f 未満の値を集めた配列，f 以上の要素を集めた配列に分割する段階で，i 番目〜j 番目であることが計算可能であるため (i，j は，各配列の要素の個数より計算)，i，j の値も配列と共にメッセージとして受渡す．結果として，動的に生成された並列オブジェクトより，`main` 並列オブジェクトにソートされた要素の断片が集められ (1 〜 3 個のいずれか)，上記 i，j をもとにして，配列に格納すればよい．

Primitive Software として，PVM を用いた場合は，各並列オブジェクトがプロセスとなる．1 台の PC では，1800 個程度が限界であるため，N の値を，1800 および，900 とし，ソートが完了するまでの実行時間を計測した (図 6)．横軸に PC の台数，縦軸に秒単位の時間とした．上段が 1800 個，下段が 900 個の場合の時間，各々上側のグラフが，SPL を用いて記述した場合の実行時間，下側のグラフが C 言語により PVM ライブラリを直接用いて記述し，実行した場合の実行時間である．

SPL を用いて記述した場合でも，C 言語により直接記述した場合に比べ，6 % 〜 10 % のオーバーヘッドに抑えられ，効率よく実行されていることがわかる．台数効果も，C 言語により直接記述した場合と同様に機能していることがわかる．

7. 議　　論

フォールトトレラントシステムを構築するソフトウエアは，機能の一つとして，実行時に，ソフトウエアの一部分を更新することが必要である．実行時に，自

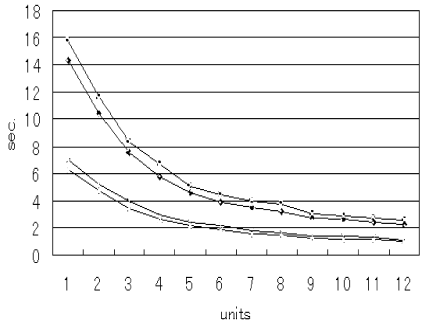

図 6 PC クラスタを用いた台数効果
Fig. 6 Execution time with increase of processors using a PC cluster

律的にソフトウエアを更新するための用件として以下の 4 点を挙げた．

(1) ソフトウエアが十分に部品化されている．
(2) 各部品が独立に，並列に実行可能である．
(3) 各部品は，さまざまなシステムで実行可能で，統一的記述がなされている．
(4) 実行時に，部品を更新することができる．

並列オブジェクト指向モデルを用いた設計，対応する記述言語 SPL を用いたプログラミングにより，(1), (2) を達成することができる．さらなる部品化として，デザインパターンを用いた設計を考えることができる．4 節 応用例で示したように，デザインパターンは，構成要素の独立性が重視され，さらに，並列に実行可能である並列オブジェクトを構成要素とすることにより，独立性の高いソフトウエア部品とすることができる．よって，並列オブジェクト指向モデルと親和性がよいと考えることができる．

SPL は，実現として C 言語へのトランスレータ方式としたため，低レベルのデバイス依存部分のみを実現することにより，PVM，p スレッドライブラリ，マイクロカーネルのスケジューラなど，さまざまな並列計算システム上で実行可能である．現在の実現では，トランスレータが出力する C 言語のプログラムは，20 個の中間関数を用いるのみであるため，これら関数を実現することで移植が完了する．その内 11 個は，セマフォ，バッファなど Primitive Software に通常備わっている機能を扱う関数であるため，デバイス非依存である．セマフォ，バッファなどが備わっている Primitive Software 上への移植は，基本的に，残り 9 個の関数を実現するだけでよい．

SPL は，その計算モデルとして並列オブジェクト指向モデルを採用したため，大規模分散環境から，オ

ペレーティングシステムのマイクロカーネルまで，幅広い並列計算システムに対して，統一的な記述をすることができ，ソースコードレベルで可搬性の高いプログラミングが可能である．以上より，(3) を達成することができる．

本論文では，SPL の言語仕様に動的コンパイルを導入した．各オブジェクトが，文字列であるプログラムから新しいオブジェクトを生成する能力を有することになる．プログラム (文字列) は，実行中の計算結果を反映して，実行中に自由に生成，更新することができるため，柔軟性の高いプログラミングが可能となる．4 節 応用例では，プラグイン，疑似マイグレーション，疑似自己反映計算について紹介した．ソースコードレベルでさまざまな並列計算システムへの可搬性が高い言語であるため，自律的に，オブジェクト間でソースコードを文字列引数として，メッセージ受渡しすることにより，さまざまな位置へ，オブジェクトを移動することができる．

一般に，オブジェクトのマイグレーションまたは，自己反映計算は，オブジェクトの実行中に，状態を「保持したまま」，各々移動するか，自身をメタオブジェクトに依頼して書き換える処理を示す．本論文で導入した動的コンパイルは状態を「保持せずに」，新しいオブジェクトを生成する点が異なる．我々が対象としているフォールトトレラントシステムの記述では，フォールト部品そのままを，新しい部品で交換する方法を前提と考え，オブジェクトの実行中に状態をそのまま保持しながら移動したり，自身を書き換えながら，高速に処理する場合は少ないと考え，動的コンパイルを導入した．動的コンパイルは，実行中に一度コンパイルすれば (コンパイル時間を除けば)，静的に定義された並列オブジェクトと同程度に実行できることが評価より確かめることができた．しかし，状態を「保持したまま」，移動したり，自己反映計算する必要がある場合は，状態をプログラム (文字列) として記録すればよい．つまり，プログラム (文字列) そのものを永続オブジェクトと考え，オブジェクトの状態を記録すればよい．再起動する場合は，再コンパイルすればよい．以上より，(4) を達成することができる．

実際のフォールトトレラントシステムを構成するソフトウエアを記述するためには，さらに，プログラム言語レベルで「例外処理」を扱う必要がある．本論文では，フォールトトレラントシステムにおいて，フォールトが発見されたとして，その対処方法に主眼をおき，記述言語の言語仕様として動的コンパイル機能を導入した．フォールトを発見し，言語レベルでの「例外処理」に関しての議論は十分でない．今後の課題とする．

8. おわりに

フォールトトレラントシステムを構築するソフトウエアの記述言語として SPL を設計し，実現した．最初に，SPL の設計目標として，実行時にソフトウエアを更新するために必要な特徴 4 点を挙げた．7 節議論より，SPL がフォールトトレラントシステムを構築するソフトウエアの記述に向いていることを確かめた．

SPL は，大規模分散環境から，オペレーティングシステムのマイクロカーネルまで統一的な記述ができる並列言語である．

各並列オブジェクトは，実行の単位として，意味的にも，物理的にも独立性が高いため，ソフトウエア部品と考えることができる．デザインパターンを活用し，組み合わせることにより，見通しのよいシステムを構築することができる．さらに，本論文で導入した，動的コンパイルを用いることにより，必要に応じて，動的に並列オブジェクトを追加し，不必要となった並列オブジェクトを削除することができるため，動的なソフトウエア部品の追加/更新が可能となる．SPL によるシステム記述の適用範囲は，フォールトトレラントシステムの構築するソフトウエアだけでなく，ネットワーク技術を利用したソフトウエア開発，ロボットなどハードウエアとソフトウエアが一体になり，自律的にフォールトを発見，取り除き，永続的に実行するシステムの記述など，非常に広いと考える．

謝　辞

本研究は，2001 年度南山大学パッヘ研究奨励金 I-A (特定研究助成) より研究助成を受けて行っています．

参考文献

1) Doi, N., Kodama, Y. and Hirose, K.: An Implementation of an Operating System Kernel using Concurrent Object-Oriented Language ABCL/c+, *ECOOP'88*, pp.250-266, ACM(1998).
2) Ferreira, L.L. and Rubira, C.M.F.: Reflective Design Patterns to Implement Fault Tolerance, *Reflective Programming in C++ and Java, OOPSLA'98 Workshop*,ACM(1998).
3) Gamma, E., Helm, R., Johonson, R. and Vlissides, J.: Design Pattern - Elements of Resusable Object-Oriented Software, Addison-Wesley(1995).
4) Kodama, Y. and Asumi, T.: Yewe:Yet An-

other Window Environment, *Computer System Simposium*, pp.1-8, IPSJ(1998).
5) Milner, R.: A Theory of Type Polymorphism in Programming, *Journal of Computer System Science, 17*, pp.348-375(1978).
6) Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y.: Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, *Object-Oriented Concurrent Programming* (ed. Yonezawa, A. and Tokoro, M.), The MIT Press(1987).

付　　録

哲学者の食事問題 (A)

```
1:start {
2:var room = [object Room
3: state var n = 0
4: script
5:   (=> :Enter { n++; !n; })
6:   (=> :Exit { n--; !n; })];
7: var f0 = [object Fork
8:  state var s = 1 var nn = 0
9:  script
10:   (=> :SetNumber int n { nn := n; })
11:   (=> :Get {
12:     if((--s)<0) {
13:      [select
14:       (=> :Put { s++; })];
15:     }
16:     !nn;})
17:    (=> :Put {s++;})];
18: var p0 = [object Philosopher
19:  state var nn = 0 var ccc = 0
20:  script
21:   (=> :SetNumber int n { nn := n; })
22:   (=> :Go oid Right oid Left oid Room {
23:    var rr = 0; var ll = 0; var b = 0;
24:     while(true) {
25:      rr := [Room <=== :Enter];
26:      b := nn%2;
27:      if(b==0) {
28:       rr := [Right <=== :Get];
29:       ll := [Left <=== :Get];
30:      } else {
31:       ll := [Left <=== :Get];
32:       rr := [Right <=== :Get];
33:      }
34:      [Right <= :Put];
35:      [Left <= :Put];
36:      rr := [Room <=== :Exit];
37:    }})];
38: var f1 = clone f0; var f2 = clone f0;
39: var p1 = clone p0; var p2 = clone p0;
40: [f0 <= :SetNumber 0];[f1 <= :SetNumber 1];
41: [f2 <= :SetNumber 2];
42: [p0 <= :SetNumber 10];[p1 <= :SetNumber 11];
43: [p2 <= :SetNumber 12];
44: [p0 <= :Go f2 f0 room];
45: [p1 <= :Go f0 f1 room];
46: [p2 <= :Go f1 f2 room];
47:}
```

哲学者の食事問題 (B)

```
1:start {
2: var room = clone("Room","\
3:  [object Room \
4:   state var n = 0 \
5:   script \
6:    (=> :Enter { n++; !n; }) \
7:    (=> :Exit { n--; !n; })]");
8: var fork = "\
9:  [object Fork%d \
10:   state var s = 1 var nn = %d \
11:   script \
12:    (=> :Get { \
13:     if((--s)<0) { \
14:      [select \
15:       (=> :Put { s++; })]; \
16:     } \
17:     !nn;}) \
18:    (=> :Put {s++;})]";
19: var str = char[2000];
20: Primitive::sprintf(str,fork,0,0);
21: var f0 = clone("Fork0",str);
22: Primitive::sprintf(str,fork,1,1);
23: var f1 = clone("Fork1",str);
24: Primitive::sprintf(str,fork,2,2);
25: var f2 = clone("Fork2",str);
26: var phil = "\
27:  [object Philosopher%d \
28:   state var nn = %d var ccc = 0 var tt = 0 \
29:   script \
30:    (=> :Go oid Right oid Left oid Room { \
31:      var rr = 0; var ll = 0; var b = 0; \
32:      while(true) { \
33:        rr := [Room <=== :Enter]; \
34:        b := nn%%2; \
35:        if(b==0) { \
36:          rr := [Right <=== :Get]; \
37:          ll := [Left <=== :Get]; \
38:        } else { \
39:          ll := [Left <=== :Get]; \
40:          rr := [Right <=== :Get]; \
41:        } \
42:        [Right <= :Put]; \
43:        [Left <= :Put]; \
44:        rr := [Room <=== :Exit]; \
45:      }}) \
46:     ]";
47: Primitive::sprintf(str,phil,0,10);
48: var p0=clone("Philosopher0",str);
49: Primitive::sprintf(str,phil,1,11);
50: var p1=clone("Philosopher1",str);
51: Primitive::sprintf(str,phil,2,12);
52: var p2=clone("Philosopher2",str);
53: [p0 <= :Go f2 f0 room];
54: [p1 <= :Go f0 f1 room];
```

```
55: [p2 <= :Go f1 f2 room];
56:}
```