並列オブジェクト指向システムＯＲＡＧＡ

5D-2

— 単一代入則に基づくオブジェクト指向プログラミング —

河野真治　立川江介　渡部真幸　田中英彦
東大　工学部

はじめに

ＯＲＡＧＡは，ソフトウェア開発をアーキテクチャの側から総合的にサポートするシステムである。特に並列アーキテクチャ上のソフトウェア開発を問題にする。ここでは，オブジェクト指向・データフロー・単一代入則を導入し，並列処理記述を簡略する。さらに名前の観点からソフトウェア開発を見直すことにより，従来とは，異なるプログラム生産方法を開発する。
このシステムは，以下の４つの部分からなる。

①並列オブジェクト指向言語DinnerBell
　並列処理を記述するために，オブジェクト指向とデータフローの概念を導入した言語
②並列デバッギング・ツールObjectPeeper
　オブジェクトと，単一代入型言語の特徴を活かした並列処理のためのデバッギング・ツール
③並列オブジェクト指向アーキテクチャOragaltSelf
　メッセージパッシングを単位とする並列実行を行なう計算機アーキテクチャ
④名前管理システムNameMaster
　ソフトウェアとオブジェクトを結ぶ名前の生成・管理・表示を行なうシステム。

これらのサブシステムが統一されて初めてＯＲＡＧＡができることになる。現在は，DinnerBell言語の試作（第２版）とObjectPeeperの表示部，NameMasterの試作を行なっている。

1.単一代入則とオブジェクト指向

オブジェクト指向のシステムは，通常Smalltalk- 80 を指すが，一般的には，データとその手続きを統一的に扱う意図を持つシステムを意味する。これは，以下の二つの動機の基づいている。

①データ抽象によるプログラムの共用
　Data Abstraction
②副作用を安全に取り扱う
　Atomic Transaction

Smalltalk- 80 では，実行は逐次なので①の抽象化に重点が置かれ，inheritance を中心としたプログラムの再利用が強調されている。一方で，オペレーティングシステムの研究者は，オブジェクトを副作用を持つ実行単位として捉えて，recoveryや，serializability の基本単位としている。

ＯＲＡＧＡシステムでは，データフローに基づく高並列実行を目指している。このようなシステムでは，副作用の存在は，プログラムの内蔵する並列度を小さくするので，一般的には望ましくない。そこで，副作用の存在するＩ／Ｏなどと，副作用の無い通常の計算とを分離することが必要である。

DinnerBellでは，単一代入則を採用することにより，通常のメッセージパッシングには，副作用が存在しない。この点で，再代入可能な変数をもつ，ORIET ／84K や，メッセージパッシングごとにSerialize をおこなうABCLとことなる。

副作用のある部分は，特別なオブジェクトに限定する。従って言語の構文とセマンティックスの定義には，副作用の概念は出てこない。ある特別なオブジェクト（例えば入出力を担当するオブジェクト）の定義に副作用が記述される。

この時，DinnerBellのプログラミングで問題となるのは，以下の点である。

①副作用のないオブジェクト指向言語でどの程度のプログラミングができるのか。
②pureな部分と，副作用のある部分とのインタフェースをどのようにするか。

以下の章では，例題を中心に，DinnerBellの構文とともに，これらの点について考察する。

2.並列オブジェクト言語DinnerBell

図１は，DinnerBellで記述した素数の生成である。$class がクラス定義を表す。$class の後に｛｝で囲まれたオブジェクトの本体がくる。本体がない定義は，前方参照のために使われる。DinnerBellのオブジェクトの定義は，以下の形をしている。

｛ pattern ｜｜message-send ｝

括弧には三種あり，三種の変数のスコープを形成する。それぞれitBlock，methodBlock，statementBlockと呼ぶ。順に，｛｝，｛｝，（）が対応する。変数は，オブジェクト生成時に作られるitBlock 変数（itBlock のスコープを持ち，～のprefixをつける。インスタンス変数に相当する）メッセージ受信時に生成されるmethodBlock 変数（一時変数に相当する）の２種がある。（statementBlockに対応するstatementBlock変数もあるが，あまり使われない。）

message-sendは，送り先とメッセージから成る。pattern は，下線を引いたメッセージの形をしている。メッセージには，引数のある形とない形の２種がある。：と！を使って区別する。

```
ret :  TRUE        引数あり
isMarked!          引数なし
```

クラスMarkedInteger では，make：N というメッセージを受け取るpattern があり，内容N をもつインスタンスを返す。答の返却は，continuation⇑に対する送信でおこなう。ここでは，ret ：が省略されている。返却されるのは，methodBlock そのものである。DinnerBellは，｜｜（ネック）があるとオブジェクト定義であり，ネックのない括弧は，単なるメッセージ送信の区切りとなる。ネストしたオブジェクト定義を許しているので，クラスメソッド，インスタンスメソッドの区別の必要がない。

continuationの指示は，関数として自動的におこなう形と，⇦で直接指示する形がある。クラスInteger では，D にメッセージを送った時のcontinuationは，pattern NDになっていて，答がNDに代入される。このpattern でも，標準的なメッセージであるret ：が省略されている。一方D に送信するメッセージは，引数としてメッセージ送信式を含でいる。このメッセージ送信式のcontinutaionは，通常の関数呼出しと同じになるように設定される。

ピリオドで区切られたメッセージ送信は，すべて並列におこなわれる。メッセージ送信の送信先を省略すると，一番内側のitBlock が送信先となる。これにより，繰返し（iteration ）を実現する。

このプログラムでは，クラスInteger が先ずNotMarke

Concurrent Object Oriented System 'ＯＲＡＧＡ'　—Programming in DinnerBell
Shinji KONO, Kousuke TATSUKAWA, Masaki WATANABE Hidehiko TANAKA

The Tokyo University

dIntegerを生成する。DinnerBellは，単一代入なので，生成したNotMarkedIntegerは同じ変数には送信されず，毎回，新しく生成された送信先に送られる。

送り先は，Sift（ふるい）であり，送られた値が，Markされていれば切り落し，そうでなければ，Outputに送信し，その新しい素数にたいするFilterプロセスを生成する。

このように，単一代入のオブジェクト指向言語でも，MarkedInteger のようなデータ抽象はもちろん，ストリームの生成もスムースに表現できる。実行は，データフロー方式を想定しているので，並列性の抽出は，自動的におこなわれる。したがって，特別な並列構文も存在しない。

このプログラムでは，Integer の生成は無制限におこなわれる。DinnerBellは基本的にはデータ駆動であるが，要求駆動もメッセージ送信の性質により容易に実現できる。

3.要求駆動の実現

図2 が要求駆動にする場合の変更点である。クラスInteger は，こんどは，送信先にたいして，waitメッセージを送る。waitメッセージの解決は，クラスSiftとFilterによりおこなわれる。waitメッセージは，ESP のバインドフックのような役目をしている。eval！は，デフォルトのメッセージパターンである。

メッセージパッシングは，一種の間接コールであり，これによって関数の遅延評価を実現している。

4.副作用をもつオブジェクト

実際に副作用をもつオブジェクトは，入出力関係であるか，ディスクなどである。DinnerBellのpureな部分での計算は，順序化が必要な場合は，明示的に変数でリンクをはる。これは，ストリームであり，必要なものは，ストリームと副作用をもつオブジェクトのリンクである。ストリームが一つの場合は，ストリームを単に追跡するだけでよいが，複数ある場合が問題であり，ストリームのnon-determinisic merge を記述する必要がある。

5.DinnerBellでの同期機構

これらの実現のためには種々なprimitive が使われてきた。

① Critical Section
② Monitor
③ Accept
④ Guarded Command

DinnerBellでは，メッセージのJoinを使って実現する。DinnerBellのpattern の部分には，message-sendの部分の並列メッセージ送信に対応して，Joinを書くことがでのきる。ピリオドで区切られたpattern は，そのすべてのメッセージがそろうまで，キューに格納される。すべてのpattern が満たされるとmessage-sendの部分が発火する。

⑤ Message Join

したがって，DinnerBellには，二つのことなる発火機構が存在する。一つは，データ依存性から決まるデータフローの発火機構，もうひとつは，データの道筋を決定するメッセージ送信の部分での発火である。

6.JoinによるCritical Sectionの実現

この簡単で，対称性の高いprimitive により，より低級なprimitive を簡単に構成することができる。たとえば，Critical Sectionは，JoinとRecursive Message Sed により，図3 のように記述できる。DinnerBellでは，Critical Sectionのための構文も用意されている。Merge についても同様に実現することができる。

```
$class MarkedInteger
[    _make: ~N_ || ^ {
 _isMarked!_ || ^ ret: TRUE;
 _content!_ || ^ ret: ~N;
 _mark!_
    }
]

$class NotMarkedInteger
[    _make: ~N_ || ^ {
 _isMarked!_ || ^ ret: FALSE;
 _content!_ || ^ ret: ~N;
 _mark!_ || ^ (MarkedInteger make: ~N)
    }
]

$class Integer [
    _N to: D_ ||
 _ND_ <- D ret: (NotMarkedInteger make: N).
 ret: (N +: 1) to: ND
]

$class Sift [
    _to: ~Out_ || ^ {
 _N_ ||
     (N isMarked!)
         yes: ( || ^ ret: (Sift to: ~Out) )
         no:  ( ||
              _NOut_ <- ~Out output: N.
              ^ (Filter cut: (N content!)
                   count: 1 to: (Sift to: NOut))
         )
    }
]

$class Filter [ (略)
```

Fig.1 N-Primes

```
$class Integer [
    _N to: D_ ||
 _ND_ <- D ret: (NotMarkedInteger make: N).
 ND wait: (|| ret: (N +: 1) to: ND)
]

_wait: B || B eval!
```

Fig.2 Demand Driven

```
$class CriticalSection
[
     _new: ~Body ;
     _enter: B . sync! ||
          _Wait <- ~Body do: B .
          Wait wait: (|| sync!)
]
```

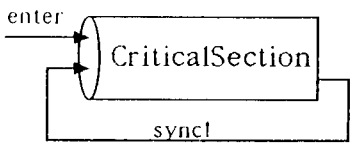

Fig.3 JoinによるCritical Section

参考文献

〔1〕 Yonezawa,Y.,"An object Oriented Approach for Concurrent Programming",T.I.T., Research Report C-63,Nov.,1984
〔2〕 横手靖彦，"Concurrent Smalltalk"，日本ソフ
〔3〕 Ishikawa, Y., "THE DESIGN OF AN OBJECT ORIENTED ARCHITECTURE ",Proc. of the 11th Int'l Symp. on Computer Architecture, Jun,1984
〔5〕 情報処理学会第30回全国大会 3P-6 (1985)