並列オブジェクト指向言語 DinnerBell の概要

神田陽治 (Kohda) ・金子誠司・田中英彦・元岡 達
東京大学 工学部

I. 研究の位置付け

新しい言語とそれを支える言語設計上の考慮点を議論する。この言語は，オブジェクト指向言語に並列性を与える試みの一つであると同時に，ソフトウェアの生産性の向上をコンピュータアーキテクチャの改造からユーザインタフェースの改良に至るトータルな手段で達成することを狙うORAGA計画と称する研究の一部分である(2)。

ORAGAとは，object ORiented Architecture to Govern Abstractions の略称であり，人間の持つ抽象化能力をコンピュータ利用全般で活かすためのコンピュータシステムの構成技術を開発する計画である。ORAGAの各サブシステムは，共通の概念を念頭に設計される，以下の４つの部分から成る。

- □ 名前を管理するユーザインタフェースの NameMaster
- □ 対象を管理するユーザインタフェースの ObjectPeeper
- □ 名前と対象を記述するプログラム言語の DinnerBell
- □ 記述されたプログラムの実行システムの OragaItself

▒研究の意義▒

最初に，なぜ，トータルなアプローチが必要なのかを説明する。ソフトウェアの生産性について，使用する言語の種類にかかわらず数ステップ／１日１人が限度であり，１人の人間が把握し管理できるのは多くて数万ステップ程度であることが知られている。これらの事実は１ステップあたりの機能を高めない限り，ソフトウェアの生産性を真に向上することはできないことを示唆している。プログラミング言語の高機能化だけではなく，実行速度や動的適応性を確保するためにはハードウェアのステップの高機能化も不可欠である※。

これまでソフトウェアの生産性を高めるアーキテクチャの研究と言えば，高級言語マシンの研究であった。高級言語マシンはファームウェアの助けにより，ソフトウェアレベル及びハードウェアレベルの基本ステップの高機能化を達成する。しかしながら，これまでの研究は，実行速度の確保という２次的な目標に終ることがほとんどであった。それらに共通するのは，『支援するプログラミング言語は最新のプログラミング技術を支援するものであるので，プログラムを速く走行させるマシンを作れば，実用的なコンピュータシステムができる。』という主張である。しかし，当然のことながら，良いプログラミング言語の使用が，そのまま高いプログラムの生産性を保証するものではなかった。

ORAGA計画は，プログラミング言語の支援だけでは不十分であるとの反省の上に立ち，トータルなアプローチでプログラミング作業を支援する枠組みを与える。

※ 第５世代コンピュータの目指す推論マシンはソフトウェア・ハードウェアの１ステップを推論に置くマシンである。

II. 知識の整理

本論に入る前にいくつかの術語の意味を明確にする。説明は例を並べず，事柄の本質を述べるよう努める。

A. オブジェクト指向

▒オブジェクト▒

オブジェクトとは，実行環境を伴った抽象データ型である。抽象データ型とは，まとまりのある変数と手続き（メソッドと呼ぶ）の記述をプログラムの文上で一箇所に集めて隠蔽し，モジュール性を高める工夫であり，ユーザ定義のデータ型を組込みのデータ型と同等に扱うことを可能にする。コンパイルされた抽象データ型は，局所変数のために割付けられたメモリブロックに他ならないが，これに手続き実行のためのスタックを加えたのが，オブジェクトである。実行環境を持つので個別に保存して後で再び活動を続けこと（ migration）ができ，従来のデータ型との大きな違いになっている。

扱うデータの型に関する情報を汎用手続きに詰め込む方法を Verb Centered ，逆に手続き情報をデータの型に含める方法を Object Centered と呼ぶことがある。おそらく，オブジェクトという呼称は，動詞に対しての目的語から来たのだろう。

オブジェクトの一つの側面が抽象データ型であった。解こうとしている問題を解析し，分解した構成要素をオブジェクトで表現するから，問題とプログラムの間のいわゆるセマンティックギャップが縮まる。

記述の手間を省くために，同種のオブジェクトをまとめて記述するオブジェクトを作る。これをクラスと呼び，クラスから作られるオブジェクトをインスタンスという。さらに進めて記述が似かよったクラスを集めて，共通部分を上位のクラスに統合し，残る部分を各々のクラスに残す。上位のクラスをスーパークラス，下位のクラスをサブクラスといい，サブクラスがスーパークラスから全ての記述を引き継ぐことをインヘリタンスという※※。現在では，複数のスーパークラスを持てるマルチプルインヘリタンスが主流である。クラス概念とスーパー・サブの関係の概念で問題をより明確に表現できるばかりでなく，編集作業時の一貫した編集が可能になる。

オブジェクト指向のプログラムは，インヘリタンスによって階層化されたクラスの集りである。そこではオブジェクトの行動はクラスの記述のみで決定され，クラス記述を変更することでオブジェクトの活動を自由に決められる。

クラス自身の記述方式は次の３つに分類できる。

- □ 叙述形式 (Imperative Form)
- □ 関数形式 (Applicative Form)
- □ 宣言形式 (Declarative Form)

※※全てを無条件に引継ぐ以外にも有効な方式がありうる。

Smalltalk のように文を並べ置くのが叙述式であり，変数は記憶の概念を持ち，それへの代入と参照により計算が進む。関数方式では，変数は単に結果を伝え合う対応の指示に過ぎず，式が変数によって結びつけられる。lispを基にした Loopsが例である。宣言方式は，言明を並べるもので，prologを拡張したESP などがある。

オブジェクトのもう一つの側面は，実行環境を持つアクティブプロセスという面である。メッセージで互いの情報伝達を行う。メッセージは2つのオブジェクトの間で送られる。送り手をセンダ，受け手をレシーバという。メッセージの受理により，メソッドが選択されて実行に移される。

▒オブジェクト指向▒

あらゆる事柄をインヘリタンスネットワーク化されたクラスで表現し，オブジェクトの連携で結果を得ることを計算とするのがオブジェクト指向であり，それに向く記述を促すのがオブジェクト指向言語である。残念ながら，オブジェクト化は万能ではない。オブジェクト指向で成功するか否かは，問題のオブジェクトへの分解の仕方でほぼ決まってしまうが，この作業は依然としてプログラマに任せられている。オブジェクト化機構の本質は，すぐに役に立つ高い機能の提供が主眼なのではなく，むしろより低レベルの基本的な機能とそれらを組合せる枠組みの提供にある。これを利用して，まとまりある単位でオブジェクトにでき，理解・修正・部品化の容易なモジュールが得られる。オブジェクト指向のプログラミング環境で素速く良いプログラムを作りうるのは，あらかじめ，利用価値の高い数多くのクラスが登録されているからである。プログラムの語彙が豊富ならば，難しいことも短く表現できるというわけである。

B.　並列処理の記述の方法

▒同期機構，synchronization mechanism ▒

計算結果をより速く得る即効薬として，並列処理は思うほど簡単ではなかった。高い性能を得るのは，ソフトウェアだけハードウェアだけの問題でなく，両方の分野の協力を必要とすることがわかって来た。難しい問題の一つが同期である。同期の問題は，排他領域（critical region ）を作るための相互排除（mutual exclusion）プリミティブを決める問題である。

同期は，さらに，排他領域の入口でのＰ同期と出口でのＶ同期の2つに区別できる。Ｐ同期は，排他領域の入口で起動可能な複数のメッセージの一つを選び，それに排他アクセス権を与えて排他領域に入ることを許し，残りのメッセージを待たせる。Ｖ同期は，排他領域の出口で，排他アクセス権を放棄して他のメッセージの領域への侵入を許可する。

多くの提案のうち，セマフォはひろく有用と認められている低レベルの同期手段で，Ｐ命令でＰ同期をＶ命令でＶ同期を実装できる。セマフォの使用を構造化したのがモニタやパス式であり，セマフォにデータを載せたのがメッセージである。

オブジェクト指向の世界では，メッセージの送受を計算の基本の操作とし，メッセージの解釈をオブジェクトに記述することで，並列処理をうまく表現できる。メッセージはセマフォの拡張なので不可分に受理されなくてはならず，同じオブジェクトに同時に到着したメッセージ受付けは直列化（ Serialize）される。オブジェクトは内部状態を持ち副作用があるのが普通なので，計算途中の状態を外にみせないように同期機構が必要である。

▒順序化機構，sequence control mechanism▒

順序化とは複数の出来事の生起順を陽に指定することである順序化はさらに2つに区別できる。

□　協調（Cooperation ）※

□　統御（Coordination）※※

協調は，対等な複数の活動間の実行待ち合せの機構である。いろいろなレベルでの協調がある。同一手続き内のステートメント間の協調をとる例として，単一代入則がある。手続き同志の協調をとる例として，モニタのキュー変数を使う協調がある同期機構のところで触れたセマフォは，低レベルではあるが，協調用のプリミティブとしても有用である。

統御は独立した複数の活動に指示を与えて協調させる機構である。パス式が統御の数少ない例の一つである。

並列処理を有効に使うためには統御が重要である。

その理由を述べる。並列計算には本来の計算に加えて，計算の相互作用を管理する余分な計算（overhead）があり，これを隠すために本来の計算に多くの並列度があることが望ましい。今，2つの仕事ａとｂがあり，ａのつぎにｂが起れという要請を考える。ａとｂの2つを知っている者が，例えば，

Carry out task＂a＂. After that task＂b＂too.

と直接書くのなら問題はないが，今，仕事A のサブタスクとして仕事ａが含まれ，同くB がｂを含むとき，

Carry out task＂A＂. After that task＂B＂too.

と書くと，真に必要なのはａの次にｂが起ることであるのに，A の実行が終了してしまうまで，B に含まれる全活動が抑制される。並列度を高めるためには必要な箇所だけに順序を指定できればよく，そのために統御機構が必要なのである。

例えば，次の2つの問題の解決には，統御がいる

(a)　仕事の間カーソルの形を換えて，作業中を表示する

(b)　関連する活動が，共有のスクリーンへの出力をする

最初の例は，本来関連のないカーソル表示ルーチンの終了と仕事開始を順序付ける例であり，次の例は，正しい順の出力を得るために，スクリーンへ送るメッセージの順序の制御をしたい例である。これらの作業を逐次実行すれば問題はないが，高い並列性を保つためには統御がいるのである。

並列起動の構文として cobegin・coend があるが統御の問題を解くには十分ではない。並列に起動したプロセスの間の関連が記述できないからである。順序を付けるためには，例えばストリームを使い，ストリームの上でａの後にｂが来るように配置し，ストリームを仕事を遂行する者に流す方法がある。ストリームの前後する空きスロットをあらかじめ配り，そこに各自が起したい仕事を置けば，ストリームに載せられた順でａやｂ

※　不幸なことに，いろいろな呼び名がある。たとえば，(1)の文献では condition synchronizationである。

※※ Coordination とは，本来なら独立なものを互いに協調させて，目的を達成することである。

が生起する。統御の機構として役に立つためには，空きスロットの配り方を工夫する必要がある。ストリームを持つ言語は多いが，主に整数発生のプログラムなどのようなパイプライン結合のプログラムを書き易くするのが狙いで，統御の機構としては改良の余地があると思われる。Concurrent Prolog はストリーム並行を使う言語の一つである。

C.　システム記述の方法

▒ポリシーとメカニズム，policy and mechanism▒

ポリシーとは，計算機資源を有効に使うための方策である。状況により最適な策は異なる。混み具合に応じて CPUタイムスライスを決めることなどである。一方，メカニズムは，あらかじめ用意されるパラメータ化されたプリミティブである。ポリシーは，メカニズムにパラメータを与え組直すことで実装される。当然のことながら，用意するメカニズムは実現できるポリシーが多様にできるように強力な一方，どんな組合せに対しても安全なように慎重に決める必要がある。

慎重に選ばれた強力なメカニズムの下では，ポリシーをユーザに任せることができる。システム記述に必要なポリシー，例えば，固定したプロセッサでの実行の指定などが可能なようにメカニズムを与えることで，システム記述が行える。

▒オブジェクト指向のシステム記述▒

実行制御のポリシーのユーザへの開放のために，ハードウェア資源を操作するメカニズムを公開する。安全に見せるため，資源を操作する手続きを通すこと，すなわち，オブジェクトとして公開する。もう一つの方法は，インヘリタンス機構を利用する。上位に位置するスーパークラスなどに，システム記述をするメソッドを置けばよい。

前者の方法は特定の資源にしか通用しないような操作に適し，後者はほとんど全ての資源に適用できる汎用な操作の実装に適当である。

D.　オブジェクト指向と並列処理の融合

▒相性は本当に良いのか▒

オブジェクトを単位としてプロセッサに割り当てると，高い並列性能が得られると言われる。並列性は，オブジェクト内の手続きのメッセージを一斉に起動することで得られる。一つの手続きが複数のメッセージを放すと，連鎖反応のように活動が起きる。しかし，オブジェクトは内部状態を持ち，かつ，共有される。誰かがオブジェクトの内部状態をメッセージを送って変更すれば，その変化は共有者から見えなくてはならない。この副作用は高い並列性を妨げることこそしないが，処理を非決定的 (nondeterministic) にしてしまう。実装方法など偶然の作用により結果が左右されるのである。非決定性と先に述べた順序化は反対の関係にあり，順序化を進めていけば決定的になっていく。並列度を落さずに決定的にするためには先に論じたように，必要最小限だけ順序化する必要がある。

▒データフロー言語とオブジェクト指向言語▒

データ依存性から作られたデータフローグラフのみを順序化機構にしたのが，純粋のデータフロー言語である。共有物はなく，全てコピーが基本で，副作用がないことが前提になっているので非決定性はない。しかし，共有メモリを用いる効率良い構造体の実装のために，ストリームが導入された。ストリームをループにすることで，実行環境の伝達が模擬できるので，履歴性が必要な入出力も実現できる。さらに，データフローグラフの共有のためにマージ可能なストリームが導入され，これを用いて内部状態を持つ共有データフローグラフ，すなわち，マネージャも実現できた。

実のところ，マネージャは，ほとんどオブジェクトである。配管がプログラム時に決まるストリームに対し，メッセージを通してオブジェクトのポインタを自由に配れるので，固定したメッセージルートを持たないオブジェクトという違いを除けば，マネージャは，入口での直列化をマージ操作により，手続き本体の相互排除を実行環境を模擬するストリームループ上のトークンの流れで実現している。ストリームによる配管が静的であるのは単一代入則を使うからである。（単一代入則とは，名前への代入をただ一度しか許さない代入である。）

データフロー言語とオブジェクト指向言語の間に，決定的な違いというものはなく，言語の適用分野が違うだけである。前者は高速の数値・記号計算を狙い，後者はソフトウェア作成支援などを目指している。高速に実行するには，マネージャの使用は必要なところに限り，コピーを原則としてコンパイル時に解決する方が有利であり，プログラミング作業を支援するためには，クラス・インヘリタンスを持つオブジェクトを全面的に使って，なるべく実行時に解決するのが良い。マネージャがストリームなどで組み立てられる製品であるのに対し，オブジェクトが完成部品であるのは，この使用頻度を反映している。このような立場の違いが言語設計に強く現われ，異なるように見える２つの言語を作ったのである。

Ⅲ.　ＯＲＡＧＡにおける基本概念

ＯＲＡＧＡで新しく導入された概念を説明する。ＯＲＡＧＡの全てのサブシステムは，これらを中心に設計される(2)。

A.　ネームの概念

▒ネーム▒

ネームとは，計算機が扱う情報のうち，偽造・改造ができないように厳重に保護されたものと定義する。この保証は，オペレーティング・システムの核とハードウェアの組合せで行う。ネームは，たとえるならば，切符である。切符を持っていればサービスを受けることができるが，切符の指定したサービス以外は受けことはできない。一般のユーザは，ネームの所持によりネームが指定するサービスのみを受けることができる。

情報をコピーして配る代りに，情報をネームで指して，そのネームを配布して情報を共有すれば，ネームの保護で情報の保護が行なえる。また，ネームの同一性を調べる操作で，種々の形式を持ちうる情報の同一性が判定できる。これらは，セマフォだけを不可分な実行命令にすることで，排他領域全域を不可分にできるのと似ている。

発想を転換し，ネームの指す情報が無い場合もあるとする。このとき，ネームはある概念を指していて，同じネームの所持は同じ概念の共有を意味する。ネームに情報を埋込んだと思ってもよい。

歴史的にはネームはケーパビリティの発展形である。ケーパビリティはセグメントを指す保護されたアドレスに他ならず，安全なオペレーティング・システムを作る技術として追求された。モジュール性の高い抽象データ型との親和性もあり，新しい計算機像として抽象データ型言語とケーパビリティベース・アーキテクチャの組合せが議論された時期もあった。

抽象データ型言語が動的な機能を入れたオブジェクト指向言語へと進歩したのに歩調を合せ，アーキテクチャ像も修正を要する。我々はケーパビリティの本質を切符と捉え，ネームへと一般化した。ネームを扱うための計算機構を備えたネームベース・アーキテクチャ（OragaItself ）とオブジェクト指向言語（DinnerBell）の組合せがより新しい計算機像である。ネームの扱いをハードウェアに任すことで，従来にない，より動的でより柔軟なコンピュータシステムを作ることができる。

▒ＯＲＡＧＡでのネーム▒

ＯＲＡＧＡではネームは３種あり，総称してNameと呼ぶ。

□ オブジェクトを指す，Capability
□ キー概念を表す，　Key
□ イベント概念を表す，Event

Capabilityはオブジェクトを指すケーパビリティである。

Nameの使用を安全にするためにキーの概念を使う。たとえを使えば身分証明書であり，切符を使う際に見せなくてはならない。 OragaItselfでは，Capabilityを伝達するとき（パラメータを引き渡すとき）や，Capabilityが指すObjectを参照するとき（アクセスするとき）に，キーである Keyをいっしょに提示しなくてはならない。

イベントの概念は原因と結果を区別するためのもので，原因にあたる。例を引けば，割込み要因がイベントで，その解析の結果，一つの処理が選択されて実行に移される。原因と結果を区別するところが新しい。

B.　シンボリックネームの概念

▒シンボリックネーム▒

シンボリックネームは，ネームが指す事物・概念の説明を与えるものである。たとえるなら，切符の使用説明書である。この説明はネームの使用のときに参照される他，ユーザがネームの内容を理解するときに役に立つ。

歴史的にはシンボリックネームは識別子の拡張である。識別子は別のものに一斉に置き換えても支障がないことからわかるように，単なる”印”でしかなかった。

▒ＯＲＡＧＡでのシンボリックネーム▒

ＯＲＡＧＡのシンボリックネームはより複雑である。その実体はネームが指す物の性質を記述した知識記述子である。意味は比較によってのみ明確に表せるとの立場に立って，知識記述子全体を意味ネットワーク様なもので関連付ける。できるものは，全体でプログラミング用の語彙といえるものになり，これを名前ベースと呼ぶ。

さらに記述子は，ユーザに見える形，表現子へと変換される。その際，いくつかの記述子をまとめて，文脈で既知の情報を省き，重複や混同が起ることのないように変換する。表現子は文字列ばかりでなく音声や図形などから，最も適切な手段を選ぶ。名前ベースの管理をし，記述子から表現子の変換を受け持つのがNameMasterである。

一方，表現子から記述子への変換は，文脈をはっきりさせて表現子と記述子の対応を明確にして行う。この作業はパーザによる自然言語の文章の解析と同じである。DinnerBellのコンパイラはプログラムの構文解析とともに，プログラム中の表現子の解析も行い，従来よりも深く精密な中間木を作り上げる。この解析は，自然言語パーザが文法知識と語彙に基づいて作業するのと同じように，DinnerBellの文法と（プログラミング用の語彙である）名前ベースを基に行う。名前ベースの役割は，従来のコンパイラの記号表の拡張と考えることができる。記述子は複数のプログラムに渡って使われるので，従来のように記号表を使い捨てるわけにはいかなくなる。この結果，記号表を管理する作業はDinnerBellからNameMasterに移る(3)。

IV.　DinnerBellと他サブシステムとの連携

A.　DinnerBellとNameMasterの連携

▒タイプ付け▒

DinnerBellコンパイラは表現子を含んだプログラムを解析して，表現子の解析をも行った精密な中間木を作り，表現子を記述子へと変換する役目を果す。NameMasterは従来の記号表にあたる名前ベースを管理する。

予約語・特殊記号・リテラルにあたる記述子は，そのもの自身の意味を表す。表現子はいろいろわかり易く表示されるが，その本質は同一である。たとえば，複素数は，直交座標表現・極座標表現をとわず複素数であることには変りがない。

対して，変数にあたる記述子は変数に代入されるものが満たすべき性質を述べたものである。これは，従来のタイプ付言語でいう変数タイプにあたる。表現子がタイプを表明しているので，プログラムを読むときに変数のタイプを調べる必要がなくなる。必要なら代入時に動的に検査をしてもよい。DinnerBellはタイプレス言語であるが，以上のようにしてタイプに相当するものを持ち込むことができる。

▒名前ベース▒

名前ベースの適用範囲は大きい程良い。全てのプログラマが共通に一つの名前ベースを使えば，同じ母国語で話せばなんら困難なく理解できるように，読み易いプログラムにできるだろう。ただ，造語の扱いが問題となる。プログラムを作るとき，なるべく既にあるモジュールを再利用するのが得策だが，必要ならば新しい概念を創出し命名することもある。これら全てを語彙として登録するわけにはいかない。こなれていない概念や利用度の低い概念は外すべきである。自然言語では個人ごとに語彙を持ち，それを使って会話をしている。そのような過程のなかで有用なもののみが国語辞典に登録され，不要になった語は削られていく。各人は国語辞典を調べて自分の語彙を標準的な語彙に合せていく。名前ベースもこれと同じ機構で造語の問題を解決できよう。個人が管理し，自分の造語を含んでよい名前ベースと，国語辞典にあたる標準的な名前ベースである。

おもしろい応用は，プログラムのリストの交換に，個人用の名前ベースを使うことである。従来，プログラムの交換はプロ

プログラムテストを通すことで行っていた。名前付けの方法は個人ごとに違うから，渡されたプログラムの解読は大変な作業である。自然言語間の機械翻訳のように，２人の個人名前ベースを通して記述子のレベルでプログラムを交換することができたなら，もらう側は自分の語彙で記述されたプログラムを読むことができ，可読性が向上すると期待できる。

最後に強調したいのは，プログラムの名前を管理することが自然言語処理技術の発展を考慮すれば決して無理難題ではないことである。自動翻訳や質疑応答システムの技術をコンパイラへ持ち込むと，利得があることを主張しているのである。

B. DinnerBellとObjectPeeperの連携

▒コメント文は使わない▒

コメントは自由に付けることができるので，品質の保証が難しい。コメントは本当に説明が欲しい所に無く，自明な所に付けてあることがしばしばである※。また，コメントはプログラムの構文を乱し読み難くする。良いコメントは長くなる傾向があり，ウィンドウに表示する場合などにも不都合である。

DinnerBellではコメントを廃止する。代りに記述子・表現子の記述能力を使う。ウィンドウ上の表現子はNameMasterによってわかり易く表示されているはずである。より詳しい情報を見たいときには，表示子を選択して対応する記述子を調べれば良い。一般に記述子はより基礎的な記述子の組合せで表現されているだろう。自然言語文を意味記述から生成するのと同じ様に，記述子から説明文を生成することができるだろう。

▒ビットマップディスプレイ向きのシンタックス▒

ObjectPeeperはビットマップディスプレイを前提とした，オブジェクト管理サブシステムである。必ずしも十分な大きさが確保できないウィンドウにオブジェクトを表示するとき，スクロールを繰り返すのでは大変なので，オブジェクトを圧縮してウィンドウに納めて表示する機能を持つ。記述子の表現子へのわかり易い圧縮はNameMasterの仕事である。

Structured Programmingは，人間のためにプログラムを視覚化する技術であったとも理解できる。Goto文の使用を控えてコントロールの構造を if や whileで視覚化し，読み易くする。プログラムのシンタックスも同じ機能なら，視覚効果の高いものを選択すべきである。DinnerBellでは，四角で囲んだり，線を引くことで影響を受ける範囲を明確に表現する。begin,endや括弧のような起点と終点を示す方法では，開きと閉じの対応が明確でなくなってしまうからである。

V. DinnerBellの設計

ソフトウェアの生産性を真に向上させるために，言語・ハードウェアの基本ステップの高機能化を行う。DinnerBellの基本ステップはメッセージ通信であり，DinnerBellプログラムの実行環境である OragaItselfでのステップはネームの計算である。また，NameMasterの基本ステップはシンボリックネームの計算である。

もちろん，これらは互いに密接に関連している。DinnerBellの設計にあたり特に考慮すべき OragaItselfの特質は，メソッドの呼出し方法である。普通の手続き呼出しは，全てのパラメータを計算した後，新しくコンテキストを生成しパラメータを渡す，コール命令で実装される。パラメータの結合は仮引数と実引数の相対位置結合である。データフロー言語の手続きを実装するコール命令は，パラメータの代りにデータフローグラフのアークを接続し，コンテキストを initiation numberを与えることで生成する。オブジェクト指向言語には，静的なデータフローグラフは存在しないので，コール命令の仕事を分割する必要がある。メッセージ伝達は，コンテキストを作る命令とそのコンテキストへ引数を一つずつ引き渡す複数のメッセージ転送命令で実装される。 OragaItselfは，コンテキストの生成は既に存在するコンテキストの複製を作ることで，メッセージの転送は Keyと引数を組にして送ることで実装する。仮引数と実引数の対応は， Keyの一致による絶対位置結合である。さらに計算途中で生じる種々の例外の伝達は Eventを相手（自分自身でも良い）コンテキストへ送り付けることによる。

あらかじめ生成されているコンテキストへ，パラメータを一つずつ絶対位置結合で渡せることで，パラメータが計算できるごとにバラバラに引き渡すことができ，並列度を高めることができる。また，コンテキストに送られて来たパラメータを調べることにより，送り元の計算の進みぐあいを判断できるので，並列処理下でのデバッグ技術としても有望である。従来の場合，計算の進行状況は送り手のコードの中に埋め込まれていて，調べることは困難であった。

以下では，DinnerBellの設計をオブジェクト指向，同期機構，順序化機構，メカニズムに分けて説明していく。残念ながら全ての詳細が確定しているわけではない。並列性をオブジェクト指向に組み込む方針として読んで欲しい。

A. DinnerBellにおけるオブジェクト指向

コンテキストは，オブジェクトの実行記憶であり，プロセッサを割り当てられて活動する実行単位である。並列環境では数多くのコンテキストが同時に存在し，多くの処理の流れが並列に走っているので，逐次環境にはない問題が発生する。

たとえば，あるオブジェクトにメッセージを送った後で，一つ以上の返事が戻るかも知れない。メソッドに複数の返事文があり，そのうちのいくつかの返事文に処理の流れが来る場合である。しかも返事は一度にではなくバラバラに戻ってくる。

解決には必要な返事の選択を記述できなくてはならない。オブジェクトから，キーを付けて返事を引き出すこととし，返事もキーを付けて送り出す。（ Smalltalk流のシンタックス）

```
f:a  g:b ⇨〔 f:x  g:y ｜｜‥⇧ m:r ) ⇨ m:z
```

返事が非同期に返って来るのを許すなら，メソッドの送信も非同期にすべきであろう。メッセージを分割して一つずつ送り込む。この見方は上で述べた OragaItselfの計算機構とうまく整合する。メッセージの送信も返事の受信もキーを付けて，絶対位置結合で値を非同期に入れ，引き出す。

もう一つの問題は，並列性処理下では多くのオブジェクトが活動していることである。互いを区別するためにNameを必要な

※ たとえば，

（＊Comment statement considered harmful＊）。

細かさで付ける技術がいる。たとえば，オブジェクトを何度も呼び出すとき，どの呼出しに対する返事かが問題になる。

```
r:z ⇦ x m:a . r:w ⇦ x m:b
```

xにメッセージ m:-を2回送り，2回起動している。⇧ r:c がセンダにメッセージ r:c を送ることであるとすると，2つの結果の戻る順番はxの起動順とは限らないので，結果が戻って来たとき，2つの受け口 r:-のどちらに代入してよいかわからない。これは，戻す場所の指定の細かさが足りないからである。自分の名前の他，後で結果を正しく分配するための情報も伝える必要があるのである。

itBlockの構造

ＯＲＡＧＡでのコンテキストは itBlockである。itBlockは分割されたメッセージを受け取って対応するメソッドを起動し，最終的に計算した値をいくつか非同期に返す。

以下，シンタックスの説明とともに補足説明をする。

(a) program,itBlock,method.

programは itBlockの，itBlockはmethodの集まりである。それぞれ，直線で四角形に囲んで範囲を明確にする。

既に触れたように，ビットマップディスプレイを前提にするなら，視覚的に優れたシンタックスを工夫すべきである。

(b) criticalSection,criticalSpot.

criticalSpotは，bodyの一部に網掛けを施して表す。これらを集めて全体で一つにしたのが，criticalSection である。criticalSection は全体で一つの名無しの排他領域となる。

(c) statement,head,body

statement は，headと □（ネックと呼ぶ）とbodyをこの順で並べたものか，headかbodyである。headは receptionを，bodyはactivityをピリオドで区切って並べたものである。

(d) activity,reception,action,messages.

activityは，⇦をはさんで reception，actionの順に並べたものか，action自身である。receptionはアンダースコアした patternの並びである。ピリオドで区切られた間のreception はメッセージの受けを排他的に行う。つまり，その区間の全ての patternは同一のセンダを持たなくてはならない。また，同じくピリオドで区切られたactivityも排他的に行う。つまり，activityを構成するactionのメッセージ送信はパックして一度に行う。たとえば，

```
l:c . k:d e! □x m:a e!. y n:b
```

2つのreception と2つのactivityがある。k:d と e!は同じオブジェクトから発信されたメッセージを受けなくてはならず，メッセージ m:a と e!はレシーバxにパックして送信される。パックされたメッセージの間には他のメッセージが入り込めず，受取りが不可分に行われることが保証される。

actionはreceiverへメッセージを送る。メッセージは＜，＞で入れ子になった messages からなる※。messagesはmessageを並べたもので，内側のmessagesからパックされて一度に送信される。

```
x m:a n:b                m:a n:b x
＜m:a x＞n:b      n:b＜x m:a＞
```

どれもxに m:a と n:b を，この順でパックして送る。送るとき結果を正しく分配するために，自身の名前と分配情報を付けることは，前に述べた通りである。

receiverが空であるactionは itBlock自身への送信とする。コンテキストを新たに作らず単なる繰返しとなる。（再帰呼出しのためにはitSelf#1 などを使う。） Parnas は if-fi文とdo-od文を合体させた繰返し制御機構 it-ti文を提案している(CACM,vol.26,no.8) 。itBlockのitはこれから得た。

(e) pattern,message,receiver.

pattern はメッセージ受けである。variableに keyで指示される値を単一代入するか，eventを受ける。message はメッセージ送信である。valueの値を key付きで送信するか，eventを発生する。receiverは，レシーバを指すvalue である。

(f) value,variable,literal.

valuleは，variableか literalか，再びactivityである。入れ子をはっきりするために（，）で囲む。

単一代入則での代入は，variableに値を設定するとともにスコープを，それを直接かこむブロックであると定める。スコープのブロック内では，variableを参照することができる。

DinnerBellでは，itBlock,method,statementの3種類のブロックを区別する。variableも対応して3種類あり，形態で区別する。この方法はコンパイラにも有利な他，形態の区別をプログラマに意識させることにより誤りを減らせるなど，変数の強制宣言がもたらすものと同じ効果がある。itBlockVariable のスコープは itBlock全体，methodVariableではmethod全体，そしてstatementVariable では statement全体である。receptorによる代入も一回に数える。

(g) pseudoVariableなど

活動中の itBlockは他の itBlockと連係して動作している。それら関係する itBlockにはNameを付けて参照可能としておかなければならない。itBlockが生成され活動を始めたとき，自動的に設定される pseudoVariablesで取り巻く環境との関係をとることにする。

たとえば，itBlock中のメソッドは複数のセンダを持つことができ，また，自分自身を複数回再帰起動できるのでそれらを区別できなくてはならない。それぞれをsender#i,itSelf#iで示す。処理中のメソッドは複数のメッセージによって起動されたものであり，これらを message#i で示す。

他の単一代入則に従うVariableは形態で区別される。

itBlockの実行

itBlockはメッセージかイベントを受け，headを検査して充足したheadが見付かったなら，一つを非決定的に選んで，そのbodyを実行する。headが充足されるとは，その全てのメッセージ受けが messageを受け，eventを受けたときである。余分なメッセージは，メッセージキューで待たされる。headが選ばれ，そのbodyが実行に移されると，headを充足させたメッセージ以外はメッセージキューに戻される。

bodyの実行は，それが含む全てのactivityの実行を並列に開

※ 自然言語の動詞の意味表現に使う格フレームと似た発想である。格の順番には意味がない。

■Syntax of DinnerBell■

```
A.          program    is-set-of              object
                       and-is-surrounded-by   □
            object     is                     itBlock
            itBlock    is-set-of              method
                       and-is-surrounded-by   □
            method     is-set-of              statement
                       and-is-surrounded-by   □
B.criticalSection      is-sum-of              criticalSpot
    criticalSpot       is-part-of             body
                       and-is-shaded-by       ▒
C.          statement  is          head □ body
                       or          head
                       or                     body
                head   is-sequence-of         reception
                       separated-by            .
                body   is-sequence-of         activity
                       separated-by            .
D.          activity   is                     action
                       or     reception ⇦ action
                reception  is-sequence-of     pattern
                       and-is-underscored-by  _
                action is              receiver
                       or              receiver messages
                       or  messages    receiver
                       or  messages<action> messages
                       or            <action> messages
                messages   is-sequence-of     message
E.          pattern    is              key variable
                       or              event
                messages   is          key value
                       or              event
                receiver   is          value
F.          value      is              variable
                       or              literal
                       or            ( activity )
            variable   is        pseudoVariable
                       or      itBlockVariable
                       or       methodVariable
                       or    statementVariable
            literal    is              number
                       or              string
                       or              statement
                       or              method
                       or              itBlock
                       or-other
```

(読み方) 下線を引いたものはメタ記号。左辺に現われているのが非終端記号。その他，終端記号。改行で区切り。

```
G. pseudoVariable  is   itBlock#1, itBlock#2,...
                   or    sender#1,  sender#2,...
                   or   message#1, message#2,...
  itBlockVariable  is        ~~ name
   methodVariable  is         ~ name
statementVariable  is           name
           number  is           capability
           string  is           capability
              key  is           name :
            event  is           name !
       capability  is           name
             name  is-defined-elsewhere
```

始させる。actionの実行は全ての変数が確定したときに始まり，メッセージ，イベントの送出をして終了する。

▒インヘリタンス▒

インヘリタンスは陽に itBlockにメッセージの転送をプログラムすることで実装する。受理できないメッセージが送られたとき，メッセージを，擬似変数 message#i によって，いったん，オブジェクト化した後，それに Forward：をつけてスーパーのオブジェクトに転送する。プログラムを解釈実行しているメカニズムがこれを再びメッセージに戻して送り届ける。

Smalltalk-80言語のオブジェクトは，転送のチェインで結ばれた複数の itBlockで実現されることになる。

B. DinnerBellに於ける並列機構

順序化の機構は従来の言語にはない工夫と考えている。以下，同期機構と順序化機構について順に説明する。このときオブジェクト指向に，いかにうまく融け込ませるかが鍵である。

重要なポイントは，並列化に伴い発生する同期・順序化を解決するためには，逐次処理の環境に比べて，より多くの事柄を仕様としてオブジェクトに規定しなければならない事と，オブジェクトは抽象データ型であるので，決められた仕様以外のことを実現してはならない事に注意することである。

B. 1. 同期機構

同期は itBlockの criticalSectionで実装する。いったん，criticalSectionで実行が始まると，それが終了するまで他の活動は criticalSectionには入れなくなる。しかし，それに属さない活動は実行できる。これで，モニタが不必要に排他領域を占有する，ネストするモニタ呼出しの問題を軽減する(1)。

B. 2. 順序化機構

オブジェクトの世界での計算はメッセージの送受で進む。オブジェクトはメッセージの指示で自分の状態を変更していく。物事の起る順序を決める一つの方法は，メッセージのオブジェクトへの到着順を制御することである。

あるオブジェクトAが，別のオブジェクトBへ到着するメッセージの到着順序を制御したいとしよう。AがBに直接送るメッセージの場合には，Aでの仕様と実装で順序の制御ができるので問題ない。メッセージのセンダとレシーバの間での協調機

構を開発すればよい。この機構を itBlockに組み込む。

一方，そうでない場合，Aの仕様に順序の指示を置くわけには行かない。仕様を実現するコードをAの中に書くことになるが，Bへ送られるメッセージの事は知りようがないので書きようがない。ところが，順序の制御をしたいのはAなので，Aに仕様を置くべきである。この解決にはメッセージ伝達を統御できる機構を必要とする。この機構を Secretaryで実現する。

Secretaryの設計に既に作成済の Secretaryを使うなどして，入れ子構造のコントロール抽象を実現することができる。

▒DinnerBellでの順序化機構（1），itBlock ▒

itBlockでは，変数は単一代入される。単一代入される変数名と参照される変数名の一致で値の伝達経路が指定される。単一代入則に従うと，プログラム上での位置は実行する順とは無関係になり，値が確定した順に実行できる。しかし，

```
x m:a . y n:b . x o:c
```

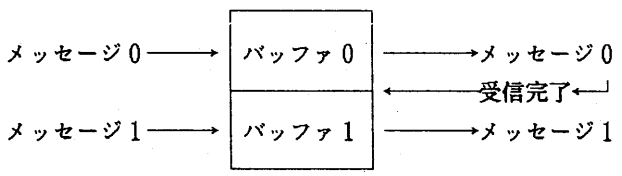
この3つのactionのそれぞれが持つ変数が確定すれば他に関係なくメッセージを送り終了できるが，この制約だけではxに到着するメッセージの順は保証できず，xが副作用を持つオブジェクトならば，計算は非決定的となる。決定的とするために，DinnerBellではプログラムの文面の順で到着順を指定する。上の例では，xとyが違うとき，xに m:a と o:c がこの順で着くことを itBlockが保証する。しかし，yに到着する n:b との到着の関係までは関知しない。この制約に従うとy が確定していないとき，後続の x o:c はたとえ条件が揃っていても実行できないことに注意が必要である。なぜなら，yの値はxと同じかもしれないからである。この問題は，番地渡しの仮引数での別名の問題に似ている。

▒DinnerBellでの順序化機構（2），Secretary ▒

忙しい人は余裕があれば，秘書を雇って自分に関係する仕事のスケジュールを頼む。もちろん，身の回りの世話は別で自分でこなす。これと同じで間接のメッセージの順次化も，それを専門として受け持つオブジェクトを用意して利用する。（そして，直接のメッセージの処理は itBlock自身でこなす。）このオブジェクトは，メッセージの統御の詳細を利用者から隠すコントロール抽象化モジュールであり，Secretaryと呼ぶ。

Secretaryはメッセージの流れの中に置かれて，メッセージを蓄えて，順序化の指定するタイミングで元の流れに戻す。

この目的のためにPairと呼ぶオブジェクトを用意する。Pairは2つのメッセージバッファを持ち，バッファ0のメッセージが相手のメッセージキューへ入ったことを確認した後に，バッファ1のメッセージを送信する。



このPairをカスケード接続することで，複雑な統御を実現する Secretaryを実装する。入れ子構造に Secretaryを組み上げて効果的にコントロール抽象を実現する方法は現在，開発中である。

C. DinnerBellにおけるメカニズム

システム記述言語として，メッセージの配送機構・メソッドサーチの機構を，言語のレベルで記述しコントロールする。言語の実行メカニズムも外にはオブジェクトとして見え，実行メカニズムへメッセージを送ることで記述する。自分自身を変えていくわけで安全なメカニズムの開発が要求される。

必要なメカニズムの機能は現在，考慮中であるが，一つの例を説明する。Secretaryの役割はメッセージをバッファし，決められた順で転送することである。考え方はデータ駆動の冗長な計算を抑えるために，要求フローをプログラムして要求駆動にする方法と似ている。しかし，オブジェクトの世界ではコンパイル時にではなく，必要に応じて自由に Secretaryをメッセージフローの中に配置できるメカニズムが要求される。一つの方法は，送信元の変数を書き換えて Secretaryを指させ，一方 Secretaryには受信先のオブジェクトを教えることである。

このような動的性は，オブジェクトの行動はそれを記述するクラスにより規定される約束からみて本質的な要求事項であり，高度な記述変更に対処できるメカニズムが不可欠である。

VI. まとめ

並列言語や並列マシンが数多く開発され，ある程度の高速化が達成されて来たが，並列プログラミングの開発が高速化で得られる利益を相殺するほど困難であったため，実用には至ってはいない。我々は最新のプログラミング技術であるオブジェクト指向の考えを基礎にしてＯＲＡＧＡ計画を開始した。サブシステムとして並列オブジェクト指向言語DinnerBellを設計し，プログラム作成用の語彙を管理するNameMasterと，ビットマップディスプレイにオブジェクトを圧縮表示するObjectPeeperの3者で，より良い並列プログラミング環境を提供する。これを支えるのが並列実行アーキテクチャの OragaItselfである。

この論文で提案した機能は多岐に渡るがゆえに，その全てが実現されてはいない。現在，ObjectPeeperとDinnerBellのインタプリタを作成している。これらの経験を活かしてDinnerBellをより良く設計し直すことになろう。

Secretaryでの逐一の記述から離れて，時相論理などでの高いレベルでの指定や，今のままではGoto文に近い itBlockの繰返しの記述をより構造化するなどの方向を考えている。いかなる機能を言語に持ち込むにせよ，常に基本（我々の場合はオブジェクト）に戻って考えることの重要性は論をまたない。

(1) Andrews G.R.,and F.B.Schneider.
Concepts and Notations for Concurrent Programming,
acm computing surveys,vol.15,no.1,1983.

(2) 神田陽治．他．
ソフトウェア作成支援のための
ネームベース・アーキテクチャとその実現法．
電子通信学会電子計算機研究会,1984,EC83-55.

(3) 神田陽治．他．
名前付けの統括管理システム，NameMasterの構成
日本ソフトウェア科学会第一回大会,1984,1C-3.