マルチメディア通信と分散処理 86－3
グループウェア 26－3
（1998. 1. 29）

緩やかなオブジェクト連携モデル:Field-Reactor Model

岩尾 忠重　岡田 誠　竹林 知善
富士通研究所 情報サービス研究部
iwao@flab.fujitsu.co.jp

コードの再利用と記述の軽減を目的としてオブジェクト指向プログラミングが導入され一般的になってきた。しかしその一方で、近年のアプリケーションやシステムはその複雑さを増し、それに伴ってオブジェクト間の関係や内部状態の複雑さもますます増加し続けている。そのため、オブジェクト間の関係を記述するためのコードも増加し、また頻繁に発生する仕様変更に対しても、プログラマはオブジェクト間の関係の不整合を調整するための本質的でない部分のコーディングに多くの時間を取られることになる。これは現在一般にもちいられているモデルが、オブジェクト間の関係を予め詳細に規定しなければならない１：１の密な連携モデルであることに起因している。そこで我々は、オブジェクトの関係を予め厳密に規定せず、変更に対して柔軟に対応可能な、緩やかな分散オブジェクト連携モデル（疎な連携モデル）について考察を行い、そのプロトタイプを作成した。本論文ではこのモデルのコンセプトの概要についての報告を行う。

1. 背景

最近のオブジェクト指向プログラミングは、CORBA[1]やDCOM[2]などにより、ネットワーク透過性も高く、オブジェクトは場所に依存しないようになりつつある。オブジェクトのプログラムコードとしての独立性は高まってきた。

しかし実際のインプリメントを考えると、オブジェクトを利用する際には、利用されるオブジェクトのIDL(Interface Definition Language)のみならず、そのパラメータの意味などもあらかじめ熟知する必要があり、これらの枠組みがオブジェクト間のロジックとしての独立性までを保証するものではないことがいえる。

また、オブジェクト間の機能インタフェースがうまく設計されていないと、オブジェクトのバージョンアップや仕様変更に伴い、これを利用する別のオブジェクトの変更も余儀なくされる。しかし、一般にはオブジェクト間の関係記述を設計の初期段階からすべて行うことは不可能である。したがって再三にわたる仕様変更やバージョンアップを行っているのが現状であり、全体として開発効率が飛躍的によくなっているわけではない。

このような問題は、オブジェクト連携が密なオブジェクト連携であるためである。すなはち密なオブジェクト連携では、利用側オブジェクトは常に相手（Server）オブジェクトを意識してあらかじめ設計する必要がある。これは相手のオブジェクトの状態を意識することを強いている。そのため、プログラミング時に予期しない状態が多数存在する場合やオブジェクトやアプリケーションを取り巻く環境が既知でない場合に対応することは密なオブジェクト連携では困難にしている。例えば、Mobile環境のように回線状態が不安定であったり、サービスを利用する場所毎の環境が異なりやすい状況においては、さまざまな状況を想定して慎重に設計しなければならない。しかし、一般には慎重に設計してもなお十分でないことが多い。同様の状況は、グループウェアシステムにおいても、新規サービスの追加・変更・削除においても生じる。

このように、設計時に予期しない状態やオブジェクトやアプリケーションを取り巻く環境が既知でない場合、言い換えると状態がOpenな系では、密なオブジェクト連携では設計が一般に非常に難しく、また複雑になってしまう。

Field Reactor Model For Distributed Objects
Tadashige Iwao, Makoto Okada, Tomoyoshi Takebayashi
Information Services Architecture Laboratory, Fujitsu Laboratories Ltd.

これを状態遷移の視点から捉えてみる。

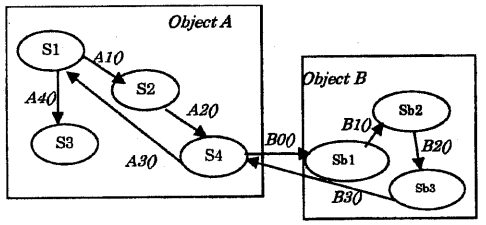

図1 密なオブジェクト連携での状態遷移概念図（同期）

密なオブジェクト連携でのオブジェクトの内部状態遷移では上記のように、ある状態Sから関数をコールしたり、変数を直接操作したりして、別の状態へ遷移する。すべて、プログラマによって、この遷移は、記述されており、状態空間は固定化しており閉じられている。

また、非同期についても、特別な“待ち”の状態があり、これから状態分岐するところが違うが通信するオブジェクトが特定されていることと、プログラマによって固定的にプログラミングされた状態遷移することには、変わりない。

状態空間を閉じてしまうと汎用性や拡張性が高いアプリケーションやシステムを作成したくとも限界が生じることは当然である。拡張性が高いシステムを構築するには1:1の密なオブジェクト連携は適しているとは言い難い。

そこで、我々は拡張性が高く、Openな系にも耐えうるようなモデルの構築を目指す。

2. Open な系のシステム

過去のモデルで拡張性が高いシステム（黒板モデル、タップル通信）を例にしてOpenな系に耐えうるような必要条件を考える。

黒板モデルは、カーネギーメロン大学で研究がなされた形態素解析システムHEARSAY-II [3]に代表される。

このシステムの特徴は、複数の知識源がそれぞれ割り当てられた黒板と呼ばれる共有メモリを参照し、それぞれの知識源がその知識源にとって必要な情報を受け取り処理し、その結果を黒板に書き込み、全体としてあるアプリケーションとして動作させようとするものである。

タップル通信は、マルチプロセッサ用のプロセッサ間通信システムで、代表的なシステムとしてLindaがあげられる。このシステムを利用するための言語としてClinda[4]などが開発されている。マルチプロセッサで変数を共有することが目的であり、その変数として文字列の並び(タプル)であることから由来している。

タップルにより変数を共有し、それぞれのプロセスは、そのプロセスに必要な変数をテンプレートにより、フィルタをかけ待ち状態に入る。テンプレートにマッチする変数が現れたとき、それぞれのプロセスは、個々の処理を行う。つまり、このシステムの特徴は、複数のプロセスがある変数を注目すること、また、その変数は他のプロセスに使われることを意識していない点である。

これらのシステムのメリットは、複数プロセスが独立動作し、PlugIn APIなどを介さず、プロセスを追加・削除が可能であることにある。タップル通信と黒板モデルの違うところは、タップル通信は、全体として一つのアプリケーションとして動作するわけでないこと、つまり、複数アプリケーションが一つのタップルを共有するところにある。

これらのシステムは、3つのOpen性を持っている。1つはパラメータ出現のOpen性であり、１つはモジュール種類のOpen性、１つはモジュール機能のOpen性である。

パラメータ出現のOpen性とは、はじめからパラメータの個数や種類が決められているわけでなく、論理的に無限種類のパラメータが出現できるという意味である。パラメータを増やす場合においても、システム全体の再構築は必要としない。

モジュール種類のOpen性とは、そのシステムに導入することができるモジュールの種類を特定していないことである。言うなればモジュールインタフェースでのOpen性である。これは特殊なAPIを介さず変数のみの着目により実現されている。特殊なAPIを介して行う場合、関数とそこに渡る引数に制限がなさ

れ、結果的に種類が特定され、Open性は低い。これらのモデルは、このOpen性があるためにさまざまなモジュールを追加が可能となる。

黒板モデルの場合そのアプリケーションに特化したモジュールだけなので、このOpen性は低いといえる。

モジュール機能のOpen性は、上のモジュール種類のOpen性と似ているが、ここでいうモジュール機能のOpen性とは、内部処理および、取りうる状態のOpen性である。それぞれのモジュールの処理は全く独立に動作をし、処理としてお互いに干渉せず、変数のみによって、お互いに影響し合う。

関数コールによるモジュールの呼び出しでは、その制御自体がそのモジュールに移ってしまう。これを防ぐには、別プロセスやスレッドにするなど予め設計をしておく必要がある。これはモジュールとしての内部処理をある程度制約していると考えることができる。

タップル通信と黒板モデルは上の3つのOpen性を持つ。しかし一方でこれら2つのモデルは、変数の信頼性とその一貫性を保証している。このため、実際にこのようなシステムを構築しようとするとき、信頼性を保証するために、ロックなどの排他制御やデータの一貫性の保証をするための仕組みが必要となる。またこのシステムを利用するモジュールはその機能を提供してもらうためサーバに接続する必要がある。このように信頼性と一貫性の保証はシステムのOpen性を一方で制限している。

モデルとしてのOpen性は高いが、保証のために要求されるハードウェアが限定されたり、通信手段が限定されたりしてしまい、実際のシステムが動作する環境の適応範囲は狭い。システムの持つ拡張性の可能性よりも強くその領域を狭めている。

さらに、データ保証をするための仕組みは、保証を行うための複雑な機構を必要とし、それなりに大掛かりになり、その分バグの混入やパフォーマンス低下の可能性が大きくなる。データ保証の仕組みが不具合を起こした場合、モデル自体はロバスト性に優れているにも関わらず、システム停止を余儀なくされる。

データの保証が前提となっているが、データ保証をしない立場からのモデルも考えられ、データ保証をしないことにより、よりモデルの適応範囲を広げることができるのではないかと我々は考えている。なお、データ保証はデータ保証しないモデルの上に構築することも可能である。

そこで我々は、データの保証をしないモデルについて検討を行った。

3. Field Reactor Model

前項で述べてきたようにOpen性を確保しながら、データの保証をしないモデルを考える上で以下の3点について注目する。

- 共通の情報（データ）の感知の場
- 情報（データ）は揮発的であり、保持されない
- 場の情報（データ）に応じて独立反応

共通の情報の感知の場では、あるアプリケーションに関連するすべてのオブジェクトが共通に情報を感知することで、特定のオブジェクトを意識する必要をなくすことができる。あるオブジェクトが必要とする情報は、その場を見る（感知）ことで得ることができる。

また、その情報は、揮発的にその瞬間だけ存在し、保持されず、後からの参照はできない。しかし、これは、その場に現れる情報を保持し、要求によって吐き出すオブジェクトを入れることでデータ保証をすることもできる。ここでは、前述したように保証しない立場をとる。さらに、この情報は全てその場にいるオブジェクトが受け取れるとも保証しない。この保証もオブジェクト間のプロトコルを組むことにより実現できる。

場の情報に応じた独立反応では、オブジェクトはそれぞれ必要とするパラメータ条件が整った時に発火することにより、独立動作する。場の情報に独立して反応するとき、特定の情報に特定の反応を固定的にプログラミングしたのでは、柔軟に場に適応できない。パラメータ列の反応をダイナミックに変更できる仕組みが必要である。

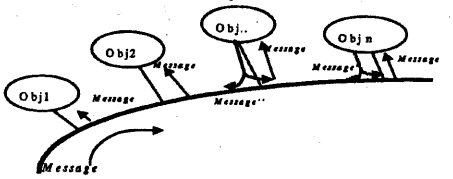

図2 緩やかな連携モデル概念図

以上述べてきた緩やかな連携モデルを、Field-Reactor Modelとして具体化した。

Field-Reactorモデルは Field、Reactorの２つの部分より構成される。

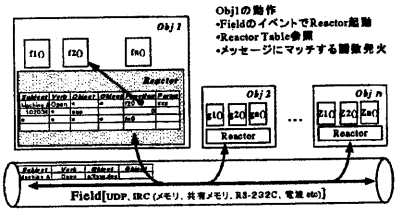

図3 Field Reactor Model

それぞれFieldとReactorについて説明する。なお、FieldおよびReactorの詳細な仕様については別途報告を行うものとする。以下に概要を示す。

3.1. Field

それぞれのオブジェクトが情報を感知するための媒体としてFieldを設けた。Fieldは共通の情報の感知の場所そのものである。Field自体は仮想的な概念であるため、テキスト列を運べるメディアはすべてFieldとして使用することが可能である。実際にはFieldの実体としてUDP、IRC、共有メモリについての実装を行った。これらをIrDA、USB、電波など他の媒体に拡張することは非常に容易に行える。このことはつまりFieldを抽象化したことによって物理的なメディアにとらわれることはなく、オブジェクト間で連携が可能となったことを意味している。具体的なプログラミングにおいて新たにメディアを追加するには、CFieldという抽象クラスから派生させ、Fieldからのデータを受信し、これをメッセージに変換するメソッドとFieldへのメッセージをそのメディアに合ったストリームデータとして変換し送出するメソッドを記述すればよい。

Fieldとしては、Common FieldとApplication Fieldの2種類設けた。Common Fieldは、Field Reactor Model オブジェクトの共通のFieldであり、メディアによってそのポートなり、チャンネルなりがあらかじめ定義されている。全オブジェクトはこのFieldからのメッセージを受ける。Application Fieldは、Application 固有のFieldである。

Fieldを流れるメッセージ形式を任意に定義できるとすると、そのメッセージを利用するためには、そのさまざまなオブジェクトが送出するメッセージに対して、専用に解析部が必要となり、結局個々オブジェクトを強く意識することになる。逆に情報の形を細かく定義すると汎用性がなくなる。オブジェクトによってはその定義から外れることも考えられる。したがって、Fieldのメッセージ形式は文法だけを定義しており、SVOOの形のメッセージが流れる。ここでは柔軟性を考慮して、メッセージ内容にワイルドカードの利用を可能とした。

3.2. Reactor

Reactorは、Fieldを流れるメッセージに反応し、オブジェクトのメソッドを呼び出す部分である。個々のオブジェクトはそれぞれ固有のReactorを持ち、同じクラスであってもインスタンスによって振る舞いが違う。Reactorは２つの部分からなる。

メッセージの内容と発火するメソッドの対応表であるReactor Tableとその制御部分である。

Reactor Tableの構成を以下に示す。

Subject	Verb	Object1	Object2	Func	Param

左から４列のSubject, Verb, Object1, Object2, はFieldを流れるメッセージの形と同じであり、マッチング部分である。

右から2列は発火部分であり、Funcはメソッドの関数へのポインタが入る。Paramはオブジェクト定義の任意のパラメータ列を入れる。

このテーブルにダイナミックに追加・削除することで、メッセージに対するオブジェクトの動作を動的に変更でき、さらに、同一クラスであっても別の動作をさせることも可能となる。

なお、Reactor Tableの発火条件として、ワイルドカードを記入できる。

制御部は、Fieldからのメッセージと上記のテーブルを参照する。メッセージとマッチしたとき、発火を起こすが、この発火には、複数個の異なるメッセージパターンが到達したときのみ発火を起こすとしている。１種類のメッセージにより発火も含まれる。

発火関数は、ユーザ定義関数を発火する場合と制御部関数を発火する場合がある。複数種類メッセージがマッチしたとき発火を行う場合は、制御部関数が発火する。制御部関数内では、あらかじめ与えられた発火条件メッセージリストを照合し、条件がそろったとき、与えられた関数または、出力メッセージを発火する。発火条件であるメッセージの到達の順番は依存しない。

ユーザ定義関数は、ユーザが任意の関数を定義することができる。

4. アプリケーション例

Field Reactorモデルのアプリケーションとしてネットワーク上の複数マシンでのマウスポインタ共有をとりあげる。もちろんCORBAやDCOMなど既存のオブジェクト連携だけでもこのようなアプリケーションは作成可能であるが、システムが複雑で必要以上に大きくなってレスポンス性能が下がる上に、Open性に関する柔軟を確保することも難しくなる。

このアプリケーションは、2つのField Reactor Model オブジェクトから構成される。1つはフィールドに対してマウスポインタに関するメッセージを投げるMouse Objオブジェクトである。このオブジェクトは、送出するメッセージが受信されることを期待しない。もう1つのオブジェクトは、Fieldに流れてくるマウスポインタに関するメッセージに反応し、そのオブジェクトが動作する端末上のポインタ操作を行うものである。このオブジェクトもメッセージが必ずくることを期待しない。

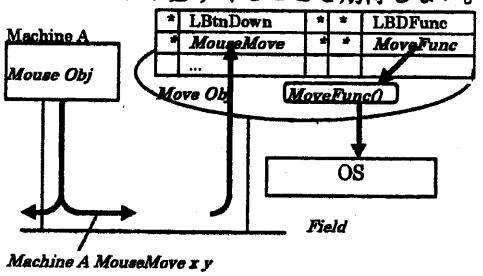

図4 アプリ例（マウスポインタ共有）

しかし、それぞれのオブジェクトを動作させるとリモートにポインタ操作を行うことができ、あたかも協調動作しているかのように見える。それぞれのオブジェクトがお互いを強く意識せず、Reactorテーブルを柔軟かつダイナミックに変更できるため、1台の端末から複数台の端末を、また、複数端末から1台の端末の操作が可能となる。また、別のMouse Objのメッセージを受信するオブジェクトとして、情報を発する端末別に最後に情報が発生した時間を表示することで、ユーザのステータス表示をするアプリケーションと捉えることも可能である。このとき、Mouse Obj等のこれまで動作していたオブジェクトの変更は全く必要ない。

5. 考察

Field Reactor Modelでは、それぞれのオブジェクトが発するパラメータによって、そのパラメータを注目する別のオブジェクトが発火していく構成である。

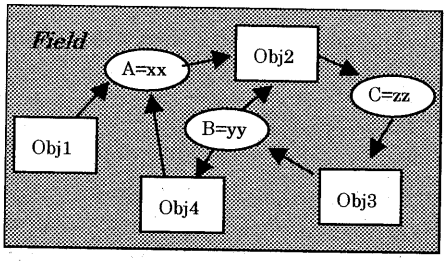

図5 Field Reactor Model でのオブジェクト関係図

Field Reactor Modelでのオブジェクトの関係は、Fieldに流れるパラメータによって確立される。

Fieldを流れる1つの変数を最小単位と考えることができ、その変数に注目するオブジェクトをダイナミックに導入することができる。また、その変数を操作するオブジェクトについても同様である。つまり、新規のオブジェクトやFieldを流れるパラメータを特定のオブジェクトやメソッドに制限を受けることなく、必要に応じて追加、削除が可能である。

前項のアプリケーション例では、Event ObjはMachine＋Mouseという変数を吐き出し続ける。これを解釈するMouse Objは、PCのマウスを実際に動かす。他方、ユーザステータス表示を行うオブジェクトはこのメッセージをユーザ操作としユーザの状態を表示する。そ

れぞれのオブジェクトは全く独立動作し、お互いに直接干渉しない。

これは、関数によって動作を定義しておらず、注目する変数により解釈を後から与えていることになっている。状態遷移として考えた場合、はじめから状態空間は閉じられていない。

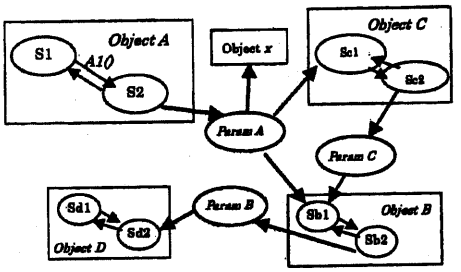

図6 FRMでの状態遷移概念図

Field Reactor Modelでは、上のように関数主体のインタフェースというよりは、パラメータ主体のインタフェースとなっており、さらに、データの到達などの保証はしていない。それぞれのオブジェクトは関与するデータを吐き出すのみであり、個々のオブジェクトは自分の関与するデータが出現すれば反応するだけである。もし、あるオブジェクトが異常終了したとしても、他のオブジェクトはパラメータが出現しないというだけで、デットロックや異常な状態に陥ることはない。このような意味でロバスト性にも優れている。再びそのオブジェクトを起動すれば動作が継続する。

6. まとめ

本稿では、予め詳細に規定しなければならない1：1の密なオブジェクト連携モデルではないモデル、緩やかな連携モデルについて述べた。このモデルでは、オブジェクト間が疎な連携をすることにより、Openな状態遷移系に対応でき、ロバスト性も高くなる。また、ダイナミックなロジック合成を行うことができ、状態遷移もある程度補完するため複雑なシステムやGUIなどに適応が可能である。

Fieldを流れる情報を状態の要素と考えると、その出現パターンによって、状態とその状態遷移が自動的に構成されていくと考えることができる。今後、このような状態およびその遷移が自然と構成されていくことを研究したいと考えている。

謝辞

研究を始めるにあたりさまざまな刺激を与えてくださった情報サービス研究部の方々と、有益な御討論を頂いたネットメディア研究部・ネットワークコンピュータ部の方々に感謝いたします。

Reference:

[1] OMG http://www.omg.org/corba/

[2] DCOM - A Technical Overview http://www.microsoft.com/msdn/sdk/techinfo/

[3] HEARSAY-II D. L. Erman, F. Hayes-Roth, V. R. Lesser, and D. Reddy. The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. ACM Computing Survey, 12(2):213-253, 1980.

[4] Clinda R.Cohen and B.P.Molinari, "Implementation of C#Linda for the AP1000" Proc. Second Fujitsu-ANU CAP Workshop, Australian National University, 1991.