



HOKKAIDO UNIVERSITY

Title	分散ソフトウェアの構築モデルに関する研究
Author(s)	原田, 康德
Degree Grantor	北海道大学
Degree Name	博士(工学)
Dissertation Number	甲第3047号
Issue Date	1992-03-25
DOI	https://doi.org/10.11501/3060651
Doc URL	https://hdl.handle.net/2115/49765
Type	doctoral thesis
File Information	000000249597.pdf



分散ソフトウェアの構築モデルに関する研究

原田 康徳

①

分散ソフトウェアの構築モデルに関する 研究

原田康徳

平成3年12月26日

目次

1 はじめに	6
2 分散計算環境の特質	10
2.1 非均質分散プログラミング	10
2.2 プログラムの部品化	11
2.3 通信の構造化	13
2.4 開放的な名前付け問題	14
3 オブジェクトの型機構	18
3.1 オブジェクトの型	18
3.1.1 オブジェクトモデル	18
3.1.2 多重ビューに基づいたメッセージの構造化	19
3.1.3 オブジェクトの型	21
3.1.4 参照の型	22
3.2 送出メッセージによる型	24
3.2.1 インスタンス変数の型	25
3.2.2 出力ビュー	26
3.2.3 ビューの名前	28
4 通信の構造化	31

4.1	問題設定	32
4.1.1	通信相手の柔軟な指定	32
4.1.2	複雑な通信系列	32
4.1.3	複数のエージェント間の通信	33
4.1.4	多重ビューを持ったエージェント	34
4.2	タプルの定義	34
4.2.1	CCS	35
4.2.2	イベント	35
4.2.3	τ が先頭に付いた+	36
4.2.4	記法の変更	36
4.3	タプルのマッチング	38
4.3.1	2つのタプル	38
4.3.2	複数のタプルのマッチング	41
4.4	エージェントモデル	43
4.4.1	タプル文	44
4.4.2	未来変数	45
4.4.3	再帰	45
4.4.4	送出の select	47
4.4.5	多重ビュー	49
4.4.6	多重スレッド	52
4.5	複数エージェントの通信	54
4.5.1	通信の抽出	54
4.5.2	通信の型	55
4.5.3	型階層間のマッチング	56
4.5.4	複数のエージェント間の通信における型階層	56

5	開放的名前付け問題	60
5.1	ドメインポートモデル	60
5.1.1	ドメインとポート	60
5.1.2	ポートへの名前付け	61
5.1.3	逆方向のポート	62
5.1.4	ドメインへの名前付け	63
5.2	名前付けにおける協調問題	66
5.2.1	ドメイン名の変換	66
5.2.2	名前の最適化	69
5.2.3	名前の同一性	73
5.2.4	ルールの形成	76
5.2.5	論理的なポート	77
5.3	DPM による名前付けの例と比較	78
5.3.1	オブジェクト ID	78
5.3.2	通信路設定問題	82
5.3.3	多重環境間の関数名	84
5.3.4	IP アドレス	85
5.3.5	uucp 表記との比較	88
6	分散プログラミング環境	90
6.1	Kamui 環境	90
6.1.1	経緯	91
6.1.2	型システム	93
6.1.3	プログラム言語	93
6.1.4	オブジェクト ID の管理	93
6.2	記述言語	94

6.2.1	Kamui-C	94
6.2.2	Kamui-Lisp	97
6.2.3	KamuiScript	99
6.3	環境支援	100
6.3.1	オブジェクトの登録	100
7	応用システム	101
7.1	Kamui-Toolkit	101
7.1.1	Kamui 環境との関係	102
7.2	HyperDESK	102
7.2.1	ハイパーテキスト	103
7.2.2	実現イメージ	104
7.2.3	メディアの統合	105
7.2.4	分散データベース	105
7.2.5	新しい構築モデルの必要性	107
8	おわりに	109
A	Kamui-C 言語仕様	119
A.1	文法	119
A.1.1	プログラム	119
A.1.2	データ型宣言	120
A.1.3	プロトコル宣言	121
A.1.4	クラス宣言	123
A.1.5	メソッド宣言	124
A.1.6	メッセージの送出	125
A.1.7	プロトコルの本体	126

A.1.8	メインプログラム	126
A.2	実行時環境	127
A.2.1	起動の流れ	127
A.2.2	起動時のオプション	128
A.2.3	ルートオブジェクトの機能	129
A.2.4	I/O 処理の追加	130

Chapter 1

はじめに

近年、ネットワークの発達とそれに接続可能なコンピュータの普及により、単独なシステムから分散システムへ爆発的に移行してきている。そのため、分散システム上でのソフトウェアの構築が社会的に重要になってきている。すでに多くのソフトウェアが現実の要求に答えているし、CSCW 等の研究が進むにつれてその度合いはさらに増すことになる。これからのソフトウェア構築を考えた場合、ほとんどのアプリケーションで少なからずこのような分散システムを指向する必要があるであろう。

しかし、現状としてはこのような分散ソフトウェアの構築を支援する環境は貧弱である。ほとんどの分散ソフトウェアは従来から存在するプログラミング言語に通信に関するライブラリによる機能の追加したものによって作成されている。OS のレベルからの分散ソフトウェアの支援もあるが、これらの研究の大部分は速度などの性能の向上と、セキュリティなどの新しい基本機能の追加に関することであり、分散ソフトウェアの作成者を直接に支援してはいない。すなわち、プログラミング言語のレベルからの支援が求められている。

このようななかで、我々は分散ソフトウェア構築支援環境 Kamui 環境の作成を通じて、幾つかの問題に取り組んできた。Kamui 環境は、並行オブジェ

クト指向プログラミングによって分散ソフトウェアの構築を支援するものである。並行オブジェクトモデルは独立した実行主体が非同期なメッセージ通信によって情報を交換しながら計算が進められるもので、分散ソフトウェアの構築モデルとして優れたものである。

Kamui 環境で目標とした主な問題点は次の2点である。

1. 分散プログラムの誤りの検出
2. 動的に分散して増加する資源の管理

1. は、誤りが生じづらい言語を設計するとともに、様々なレベルでの誤り検出を可能とするものである。特に我々が対象としたシステムは、様々な OS 等の実行環境で、様々な言語によって記述された分散プログラムが互いに関係しあって動作するものである。このようなシステムは非均質なシステムと呼ばれている。そこで、必要とされるのは言語を超えた誤りの検出の方法である。つまり、各言語間で共通となる記述のレベルを与えて、それを環境からの支援によって扱うのである。

分散システムでは誤りが生じてでもできるだけその影響が少なく、正常な部分は動作し続けるような要請がある。そのため、早期の誤りの検出が必要である。従来のプログラミング言語における誤り検出はコンパイル時に行なうものと、実行時に行なうものが存在していた。さらに、分散システム固有の性質であるシステムの一部が動的に変化するような場合に適した誤りの検出のレベルが必要である。すなわち、システムの一部が変化した際にその変化に応じた誤りの検出の方法である。

2. は、開放系の資源管理の問題として捉えられる。開放系とはそのシステムの境界が刻々と変化するようなシステムである。例えば動的に資源が頻繁に出入りする場合や、システムが巨大で全体を一度に見渡すことのできない場合などに、そのような性質が現れる。我々はこの開放系において名前付

けの問題を解決した。大規模な分散システムにおいて、それを閉じたシステムとして捉えた名前付けの方法では、様々な問題が生じてきている。身近な例では、電話番号の桁があふれた問題などがそうであるし、現状の WAN ではソフトウェアではなく、多くのシステム管理者の手によって大規模システムの名前の管理が行なわれている。一方、それらのシステムを開放系として捉えることにより、局所的な整合性のみを保証するだけで全体が動作するため、以上のような問題は生じない。

さらに、以上のような提案が現実の問題にうまく適用されているかを確認するために、Kamui 環境上で二つの記述言語を作成した。一つは C++ をベースとした、高い効率と静的な誤り検出の機能を持つ Kamui-C と、もう一つは Lisp をベースとした、動的な性質と対話的な操作が可能である Kamui-Lisp である。これらは、Kamui 環境が提供する誤り検出の機能を持ち、互いに区別することなく動作する。また、それらの言語で作成されたオブジェクトの ID は開放系の名前付けモデルによって付けられ、開放的な管理が行なわれている。すなわち、Kamui 環境で作成したシステムは大規模で動的な開放的な性質を持つことができる。

また、Kamui 環境での実際の記述力を調べるために CSCW に適したユーザインタフェースの構築支援となる Kamui-Toolkit と、カード型の開放分散データベースである HyperDESK を作成している。

本論文は、8 章から構成されている。2 章は分散ソフトウェアに要求されている性質について明らかにする。3 章は非均質分散ソフトウェアに適したオブジェクトの型機構について述べている。4 章はメッセージの系列に着目した計算モデルの提案を行なっている。5 章は開放系名前付けモデルについて述べている。6 章は Kamui 環境について述べ、7 章でその上で構築したシステムについて述べる。最後に 8 章で以上のまとめを行なっている。さらに、

付録として Kamui-C のマニュアルが付加されている。

Chapter 2

分散計算環境の特質

2.1 分散計算環境の概要

分散計算環境とは、複数のコンピュータがネットワークを介して接続され、それぞれが独立して動作しながら、共同して一つの大きな計算タスクを遂行する環境である。この環境では、各ノードが持つ計算能力を統合し、大規模なデータ処理や複雑なシミュレーションなど、単一のコンピュータでは実現が困難な計算を実現する。また、分散計算環境は、柔軟なスケールアップが可能であり、必要に応じてノードを追加することで、処理能力を拡張できる。さらに、分散計算環境は、高可用性と信頼性を確保し、計算タスクの中断や失敗を最小限に抑えることができる。

分散計算環境の構成要素には、クライアント、サーバー、ネットワーク、データストレージなどが含まれる。クライアントは、計算タスクを发起する役割を担い、サーバーは、計算処理を実行する役割を担う。ネットワークは、クライアントとサーバーを接続し、データのやり取りを可能にする。データストレージは、計算に必要なデータを格納し、必要なときにクライアントやサーバーに提供するための役割を担う。また、分散計算環境では、分散ファイルシステムや分散データベースなどの技術が利用され、データの分散管理や高速アクセスを実現している。

分散計算環境の利点として、計算能力の向上、処理時間の短縮、コストの削減などが挙げられる。また、分散計算環境は、大規模なデータ処理や複雑なシミュレーションなど、単一のコンピュータでは実現が困難な計算を実現する。さらに、分散計算環境は、柔軟なスケールアップが可能であり、必要に応じてノードを追加することで、処理能力を拡張できる。また、分散計算環境は、高可用性と信頼性を確保し、計算タスクの中断や失敗を最小限に抑えることができる。

Chapter 2

分散計算環境の特質

この章では分散ソフトウェアに求められている性質について述べている。

2.1 非均質分散プログラミング

ネットワークで接続された分散システムでは、スーパーコンピュータや記号処理マシンなどのような専用のマシンと、ワークステーションやパーソナルコンピュータなどの個人用のマシンが互いに接続され、人間の様々な活動の支援を行なっている。このようなハードウェアは今後も様々なアーキテクチャのものが開発されるであろうから、分散システム全体は本来非均質でダイナミックな基盤であるといえる。

ソフトウェアに対する要求も、高い実行効率と高い柔軟性の両方を求めるなど、より高度になってきており、単一のソフトウェアシステムでそれを解決することが困難になりつつある。さらに、分散計算の特徴であるシステムの頑強性や機能の多様性を実現するためにも、非均質ソフトウェアが求められている。

一方、多くの分散システムは分散オブジェクト指向モデルに基づき、個々の計算主体がメッセージパッシングにより情報を交換して計算が進められている。オブジェクトは共通のインタフェースを持っており、非均質ソフトウェ

アの実現に適している。

非均質プログラミングを支援する方法が、これまでも幾つかの研究や実働システムにより提案されている。Apollo Domain や Macintosh の MPW では、C や Pascal などの言語間で互いの関数が呼び出せるようなコンパイラを用意している。Matchmaker[32]、Proxies[37]などは、通常関数呼び出しの機能を分散プログラミングのプリミティブとしたRPC(Remote Procedure Call)により、そのまま複数の言語間の呼び出しができるようになっている。また、Lisp や Smalltalk-80 などのような単一言語環境においても、他のシステム記述言語へのインタフェースを用意している。これらの方式の主要な部分はデータ構造の表現の違いを相互変換し、RPC を基本としている。Kamui 環境はさらに進んで、非均質な分散オブジェクトの型を導入し、より大規模なプログラミングに適した支援を行なうことを目指している。

2.2 プログラムの部品化

ソフトウェアの再利用性を高めることはきわめて重要であり、そのためにプログラムの部品化が行なわれている。個々の部品は独立に作成され、その仕様だけが外部に公開される。部品の実現は他の部品に影響を与えないので、それぞれの部品の特性に対して自然なプログラミング言語で記述できる。つまり、非均質なソフトウェアはプログラムの部品化の観点からも避けることができない問題である。オブジェクト指向プログラミングはこのようなプログラムの部品化を支援する強力なパラダイムである。

安全な部品化を行なうためには、部品の仕様を形式的に記述する必要がある。そこで、部品の仕様を表現する一つの側面として、オブジェクトの型を導入する。非均質なオブジェクトを考えていることから、オブジェクトの型は、すべてのオブジェクトから共通に見える表現、つまり、記述言語に依存

しないものにする必要がある。このような言語間で共通の表現は、RPC におけるスタブ記述言語 [32] でも行なわれている。

オブジェクトの型に関する多くの研究 [34] は、型をクラスとして扱うものであり、クラス間の継承関係が重要な問題となっている。しかし、異なった言語間の継承を考えることは不可能であるから、クラスを共通な表現として考えることはできず、型としてクラスとは別の見方が必要になる。一方、モジュールプログラミングの意味でもクラスを共通な表現として考えることはできない。単一な言語システムでも、多人数で大規模なプログラミングを行なう場合には、クラス名やメッセージ名の衝突が起こるかも知れない。

そこで、Kamui 環境ではクラスとは別の情報、すなわち次章で述べる多重ビューに基づくメッセージに着目してオブジェクトの型とすることを提案している。

部品化に伴う他の問題は部品の接続可能性である。部品化が有効に行なわれた後には、大部分のプログラミングではそれら既存の部品を接続して用いることになる。ユーザインタフェースにおけるツールキット等はそのよい例である。例えば、ボタンやメニューなどの部品を適当な値に初期化した後、それらは別のウインドウなどの部品へ張り付けられ、接続される。このようなことを安全に行なうためには、部品間の接続に関する情報が必要である。ここで、オブジェクトの接続可能性とは、一方が送出する可能性のあるメッセージを他方が全て受理できることである。そのため、Kamui 環境ではオブジェクトの仕様として送出メッセージの集合も与え [28]、型として受理メッセージのみでなく送出メッセージも含めたビューの集合に基づく型を提案している。

また、ここで作られたものは、全体として新しい部品（例えば、ボタン付きのウインドウ）として扱われ、さらに別の部品（スクロールするウインド

ウ)に接続される。すなわち、合成した部品群を新しい部品として扱うことができるような枠組も必要である。Kamui 環境が提供するオブジェクトの型は、オブジェクトの合成を安全に行なうための強力な枠組となっている。

2.3 通信の構造化

我々は、メッセージの交換により計算が進められる並行オブジェクト指向モデルを分散ソフトウェアのモデルとして採用した。しかし、このモデルはオブジェクトの独立性を高め、その記述能力の向上を主な目的としているため、逆に通信のやり取りの情報が大きく欠けている。つまり、交換されるメッセージの順序が決められているようなオブジェクトを記述することが困難である。そのようなメッセージの順序に関する情報はオブジェクト内のプログラム（メソッド）中に間接的に存在している。

CCS などに代表されるプロセス代数は、このようなメッセージを交換して動作する並行実行の単位をプロセスとして、それが外の環境と交換するメッセージのみに着目した代数である。そこでは、どのような順序でメッセージが飛び交うのであるかが形式的に論じることができる。分散ソフトウェア部品はこのようなプロセス代数の立場から見ることによって、それが関わるメッセージの順序に関する抽象化が可能となる。

プロセス代数によって、2つのプロセスの間で通信が正しく行なわれるかを事前に検査することができる。メッセージの順序の抽象化は、通信プロトコルの設計に相当するので、このプロセス代数の扱いにより、プロトコルの互換性の判定やデッドロックの回避などを行なうことができる。Nierstratzらは、このようなプロセス代数の記述が並行オブジェクトの型として扱えることを示した。そこでは、型の包含関係がプロセス代数の演算によって与えられている。

オブジェクトの通信相手を決定する際に、それをできるだけ柔軟に行なう方法が望まれている。Linda などはタプルスペース書き込まれているタプルとそれにパターンが一致するタプルとの間で相手を直接指定しない通信が可能となっている。

2.4 開放的な名前付け問題

大規模で動的に変化するシステムではシステム全体を大域的に扱うことが困難である。例えば、大規模分散システムでは通信の遅延 [16] 等により大域的な処理は困難であるし、多重言語による非均質システム [29] では、大域的な処理のためには全体の言語にまたがった宣言が必要になる等の問題が発生する。また、大規模なプログラムに部分的な変更を繰り返して行くと、プログラム全体の整合性を取ることが困難になる。さらに、稼働中のシステムを停止させることなくシステムの一部を取り替えたり、システムの設計段階で予測ができなかったような使用や拡張などを行なうことは、ソフトウェアのライフサイクル上避けられない。このような状況の下では、ソフトウェアは全体の整合性を常に保ちながら動作することは不可能であり、部分的な整合性のみで満足し、動作しなければならない。これは開放系 [6] から見た問題として捉えることができる。

開放系では、上で述べた問題を解決するために、動的に他の資源の存在を認識することが必要である [10]。その過程で、新しく知ることになった資源へいかにして名前を付けるか、という問題が生じる。開放系の特質として大域的な名前を用いることが不可能であるから、局所的な名前付けによってこのような問題を解決しなければならない。これを「開放系の名前付け問題」と呼ぶことにする。

開放系は [6] でも述べられているように、大域的なものが存在せず、サブ

システムのやり取りによって全体が動作する。そこでは、対象の全てを扱うことが不可能であるので、探索などの問題は開放系では完全な結果は得られない。また、常に計算コストなどと探索の満足度とのトレードオフにより、計算が進められる。開放系の計算モデルとしては[16]や[14]などの研究が進められている。

開放系がある程度うまく動作するためには、そのサブシステムが知的である必要がある[9]。他のサブシステムから新しい知識を得たり、ある計算を他のサブシステムに依頼するなど、従来の集中管理における計算とは異なった性質が要求される。このような開放系を考える際に、人間社会における人間同士の協調によって行なわれている行動を参考にすることができる。

大規模な計算機システムでは、ほとんどの資源を開放系として捉えることができる。例えば、大規模分散システムにおいて動的にその構成が変化する場合、全体を大域的に捉えることは不可能である。本論文では、大規模分散と開放分散を明確に区別するために、大域的な概念（特に名前付け等）が存在するか否かで分類することにする。しかし、両者のハードウェア的な性質は似ているので、それらで動作するアルゴリズムの中には類似のものもある。

また、モジュールプログラミングは局所的な関数名をモジュール内に閉じ込めることにより、他のモジュールからの干渉を防いでおり、モジュール同士の制限された接点を通じた関数の呼び出しにより、全体が動作している。モジュール名は大域的な情報であるが、より大規模なプログラミングではモジュール名自身をも局所的に扱いたくなるであろう。これも、開放系の問題と考えることができる。さらに、仕様が部分的に更新されている様々なバージョンのライブラリがあるとき、近いバージョン同士の関数呼び出しは可能かも知れないが、大きく離れたバージョン間ではそれが不可能に近くなるであ

ろう。つまり、システムの時間軸に対しても開放系としての性質が存在する。

このような開放系において、いかに開放された資源へ名前を付けるかが問題となる。すなわち、すでに自分が知っている資源と新しく知ることになった資源とを区別することができるような名前付けが必要なのである。開放されていないシステムでは大域的にユニークな名前を用いることができるので、このような問題は生じない。しかし、開放系では大域的にユニークな名前を付けることは不可能であり、局所的な解法が必要となる。

この意味で、現行の大規模分散システムは完全な開放系とは言えない。ここでは、システムの外部の機構、すなわち社会的な機構によりユニークな名前付けを行なっている。システムで動作する様々なアルゴリズムは名前のユニーク性に依存しているので、そのユニーク性が崩れるとアルゴリズムは正しく動作しない。つまり、システムの安全性は社会的な機構に委ねられているのである。

また、従来から木構造を用いて大規模分散システムを分散的に管理する方法が用いられてきた[12]。その際、名前の構造に不定長を取り入れれば、開放的な性質を含んでいるようにも思われがちである。しかし、このような単一の木構造では真の意味での開放系を掌握することはできない。この階層的な名前付けはシステム全体で共通の一つの根を設定し、ある基準によって全体の資源を木の構造に対応させて行く。例えば、根は世界であり、その下に国の名前があり、次に組織の種類で分類し、最後に組織の名前がくるような構造である。このような構造を最初に設定できるのは、そこで名前付けしようとしている資源の種類が最初から想定できるからである。この例では、すべての組織はどこかの国に所属していることを前提としているが、もし、どここの国にも所属していない組織が存在した場合、この木構造では不自然になる。まったく新しい木を導入して名前の再割り当てを行なう必要が生じるか

もしれない。開放系の立場は、このような予測が不可能である資源に対しても、対処できるようにしようとするものである。すなわち、大域的な木は仮定しないで、部分が動的に変化するようなものである。

DPM では、開放系の相対的な性質を利用して局所的な名前付けを行なっている。名前が相対的に付けられていることから、名前の移動に対して名前の変換が必要になる。また、大域的なユニーク性が存在しないため、2つの名前が同一の資源を指しているかどうかを判定する処理が必要となる。DPM ではこれらの処理を、局所的な計算の協調によって定義されている。

Chapter 3

オブジェクトの型機構

3.1 オブジェクトの型

3.1.1 オブジェクトモデル

Kamui 環境ではソフトウェアの資源をすべてオブジェクト (Kamui オブジェクトと呼ぶ) として捉える。そのために、Kamui オブジェクトのモデルは多くの資源が参加できるようにするために単純なものにする必要がある。しかし、これはより複雑なオブジェクトモデルの Kamui 環境への参加を妨げるものではない。

Kamui オブジェクトは以下のように定義される。オブジェクトは非同期にメッセージを受け、それに対応した処理を実行する。非同期なメッセージの受理のために無限長のメッセージキューが各オブジェクトに存在する。これは Actor モデル [21] と等価で、Kamui88[42] の流れを汲んでいる。

さらに、オブジェクト間の通信が非同期なメッセージ通信のみではプログラムがしづらいとの配慮から、同期通信すなわち RPC も用意している。しかし、RPC の実現は記述言語によっては実現が困難な場合もあるので、必ずしも必要というわけではない。

オブジェクトの実現方式に関しては特に規定はなく、あらゆる方式が Kamui オブジェクトの実現に利用できる。すなわち、クラス階層により作られたも

の、デリゲーションによるもの、継承の支援が無いオブジェクトベースによるもの [44] 等である。さらに、UNIX の `ls`、`grep` 等のコマンドや `awk`、`dc` 等の言語のようなものも、その上に適当な殻をかぶせることにより、Kamui オブジェクトとしての機能を持つことができるようにしている。

一つのオブジェクトの内部における並列性は Kamui 環境では特に規定しない。ある種のオブジェクトは複数のスレッドを内部に持つかも知れないし、別のオブジェクトではメソッドの実行中に到着したメッセージはキューに待たされるかも知れない。これらの違いはオブジェクトの持つ能力の違いであり、Kamui 環境ではそれらが区別されることなく扱われる。

3.1.2 多重ビューに基づいたメッセージの構造化

ここでは、様々な実現形式を持ったオブジェクトを統一して扱うための型機構を述べる。大規模なプログラミングになると、メッセージ名が大域的に一つの意味に保てなくなってくる。例えば `Size` というメッセージは、ウィンドウオブジェクトに対してはその画面上の大きさを意味するが、ライブラリオブジェクトに対してはそのバイト数を表現する意味となるであろう。このようにメッセージ名とそれが持っている意味とがあいまいである場合、メッセージ名を用いて仕様を定義することはできない。

さらに、ウィンドウとライブラリの両方の性質を持ったオブジェクトに `Size` というメッセージを送った場合、どの意味として扱うのかという問題も発生する。これは多重継承における名前の衝突の問題にも類似している。Eiffel [35] では多重継承した名前が衝突した場合のために、衝突したどちらかの名前をプログラマが付け直す機能を提供している。

しかし、より素直な解決方法は、モジュールプログラミングにおける関数名の扱いと同様に、メッセージの名前空間に構造を持たせることである。すなわち、メッセージを一意に定めることができる範囲まで名前空間を狭め、

どの空間における名前であるのかを示して用いるのである。

このようなメッセージ名のもつ意味が一つに保てる範囲は仕事の範囲に関係している。上の例では、ウインドウやライブラリの仕事の範囲で名前が閉じている。もし、Size という名前が衝突してもそのオブジェクトにどのような仕事をさせているのかがはっきりとしていれば、混乱は起こらない。ここでの「仕事」とは一般にビュー [39, 30] と呼ばれ、メッセージの集合として定義される。ビューの考えが重要となってくるのは、オブジェクトが複数の「仕事」を行なう場合、つまり複数のビューがある場合である。これを多重ビューと呼んでいる。

Kamui 環境ではビューのことをメッセージプロトコル（以下、混乱のない場合には単にプロトコル）と呼んでいる。プロトコルはメッセージ名の集合により定義され、プロトコル自身にも名前が付けられる。もし、2つのプロトコルの要素のメッセージ名がすべて同じものであったとしても、プロトコルに付けられた名前が異なっていれば、それらは別のプロトコルと見なされる。メッセージはプロトコル名とメッセージ名との組によって表現される。つまり、プロトコルが異なっていると、同じメッセージ名でも異なったメッセージとして扱われる。

Table 3.1: メッセージの構造の例

protocol	name	message
A	a	A:a
	b	A:b
B	a	B:a
	b	B:b
	c	B:c
C	a	C:a
	c	C:c

表 3.1 はメッセージの構造の例である。同表で、名前 a はすべてのプロトコルで使われているが、それぞれ異なったメッセージとして扱われる。図 3.1 は表 3.1 で与えられた 3 つのプロトコルを持つオブジェクトである。メッセージは異なったものとして扱われているが、それに関連したメソッドは共有されている場合もある。この共有は実現に依存することであり、外部の仕様とは分離されている。

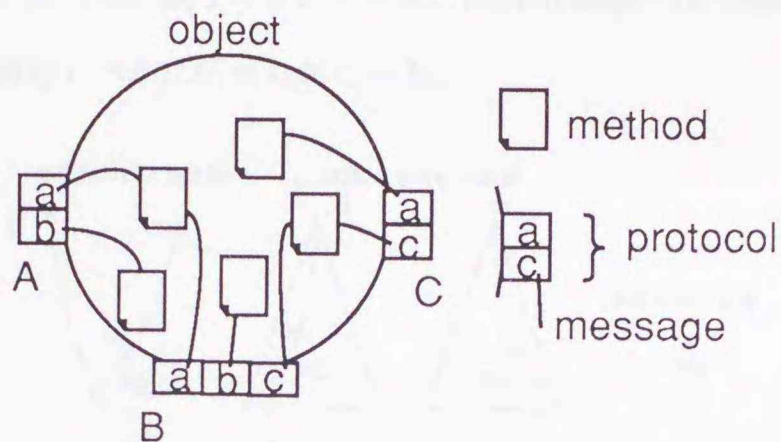


Figure 3.1: 多重ビューとメソッド

3.1.3 オブジェクトの型

オブジェクトはプロトコルの集合によって型が付けられる。図 3.1 のオブジェクトは型 $\{A, B, C\}$ である。

プロトコル間には包含関係は存在しない。その理由はモジュールプログラミングの性質を強調したいからと、オブジェクトモデルを単純化させるためである。しかし、オブジェクトの型がプロトコルの集合で定義されているため、オブジェクトの型のレベルでは包含関係は存在する。

プロトコルに包含関係を持たせたい場合には、その関係をオブジェクトレベルへ展開する方法が考えられる。例えば、表 3.1 でプロトコル B がプロトコル A を継承して作られた階層的な仕様の定義 [23] であるとする。つまり、

B は A を継承し、かつメッセージ c が新たに加えられたものである。あるオブジェクトが型 B である場合、それは仕様の継承関係によって型 A でもある。それを、Kamui 環境の方法では A と B の間に関係を持たせない代わりに、そのオブジェクトの型は $\{A, B\}$ であるとするのである。

このように、集合構造へ変換できるすべての表現による仕様、例えば図 3.2 のような異なった階層の仕様や階層を持たない仕様、あるいは多重継承による仕様などを同等に扱うことができる。Kamui 環境では仕様階層などの表現は記述言語のレベルに任せられている。

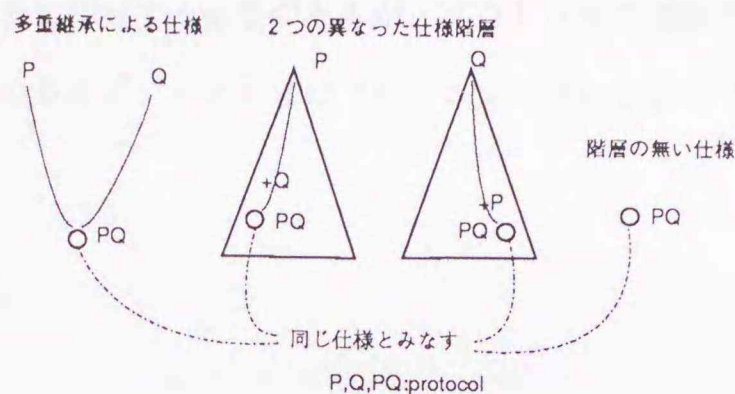


Figure 3.2: 2つの仕様階層

3.1.4 参照の型

オブジェクト自身の型を実体の型と呼ぶことにする。それに対して、オブジェクトを参照する変数に付いている型を参照の型と呼ぶ(図 3.3)。

参照の型はオブジェクトを使用する側からみた要求仕様であり、それはオブジェクトの内部の問題であるから、その扱いは記述言語に委ねられている。ここでは、後述する 2 つの言語 Kamui-C、Kamui-Lisp を通じて、参照の型の扱いについて説明する。

Kamui-C は静的に型が付けられた言語であり、参照の型はオブジェクトを参照する変数や、メッセージの引数や戻り値に対して付けられる。参照の

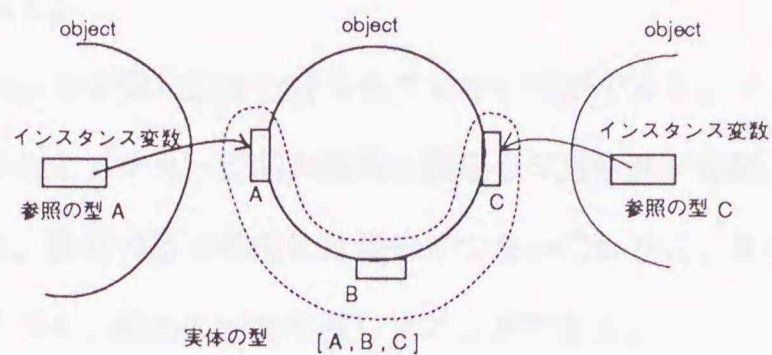


Figure 3.3: 実体の型と参照の型

型は1つのプロトコルが対応する。メッセージの送出には、プロトコルとメッセージ名の両方を指定する必要があるが、プロトコルの情報が付いた変数から参照されているオブジェクトにはプロトコルの指定を省略できる。

```
A obj;
:
obj.a();
obj.b(5);
```

A はプロトコルの型であり、obj はその型の変数である。ここでは、obj から指されているオブジェクトに対してメッセージ A:a と A:b とを送出している。その変数を通じて送るメッセージがそのプロトコルに含まれているということは静的に調べることができる。obj が指しているオブジェクトが正しくプロトコル A を知っているかどうかの判定は、A 以外の型から obj に代入される際に動的に行なわれる。

複数のプロトコルを持っているオブジェクトを使う場合、特定の局面では複数のプロトコルを同時に用いることは少なく、1つのプロトコルのみを用いることが多い。そのため、Kamui-C では参照の型を1つのプロトコルに対応させている。しかし、一般には参照の型は実体の型の部分集合として定義される。その場合にはメッセージ送出の際に、特定のプロトコルの指定を行

なう必要がある。

Kamui-Lisp は変数に型が付けられていない言語である。メッセージは通常、プロトコルとメッセージ名の両方を指定して送られ、毎回、動的な検査が行なわれる。参照の型は明確には現れてこない代わりに、次のような構文が用意されており、検査の回数を減らすことができる。

```
(with-view A obj
  (send-view a)
  (send-view b 5))
```

このフォームの中で obj をプロトコル A で参照しており、obj に対してメッセージ A:a と A:b を送出している。この構文に入った時点で動的な検査が行なわれるがそれ以降は行なわれない。

Kamui 環境が目指している動的なシステムでは、C++が行なっているような完全に静的な誤り検査だけでは扱えない問題が存在する。かといって、Smalltalk-80 や CLOS などの型の無いシステムのように、動的ではあるが、メッセージが送出されるたびに検査が行なわれるのでは、効率が悪く、誤りの発見も遅れる場合がある。発見が送れると、誤まった箇所とその検出までの間に他の処理が含まれ、その処理が中途半端に行なわれ、拡散する可能性がある。これは、特に大規模で分散したプログラミングでは危険なことであり、このような誤りの周囲への影響は最小限に抑えるべきである。ここで述べた Kamui-C や Kamui-Lisp の機能は動的で早期に誤りを発見する能力を持っており、大規模なプログラミングに適したものである。

3.2 送出メッセージによる型

これまではオブジェクト自身の型について述べてきた。しかし、オブジェクトを設計する際に、それが他のどのようなオブジェクトと共に動作できる

のかをも考慮する必要がある。つまり、どのオブジェクトとどのオブジェクトとが結合できるのかを陽に示す機構が必要である。

2つのオブジェクト A、B が接続できるためには、A が送出するメッセージの集合 S と、B が受理可能なメッセージの集合 R との間に $S \subseteq R$ の関係が存在すればよい。例えば、UNIX のフィルタは前段の出力データはすべて後段の入力として受け付けなければならない。受理可能なメッセージの集合はすでに実体の型として外部に公開されている。同様に、送出可能なメッセージの集合についても、オブジェクトの仕様として公開する。

3.2.1 インスタンス変数の型

オブジェクトが送出するメッセージに関する情報は、そのオブジェクトが持っているインスタンス変数に付けられた参照の型に示されている。これまで、そのようなインスタンス変数は情報隠蔽という立場から直接外部に公開されてはいなかった。しかし、その情報はその変数にアクセスするためのメッセージの引数の型を通して知ることができる場合がある。例えば、オブジェクト A が型 B のインスタンス変数 V1 を持っていた場合、その変数进行操作するために次のようなメッセージが用意されているとする。

```
setInstanceVarV1(B obj);  
B getInstanceVarV1();
```

これらのメッセージの引数や値の型に B が使われ、これによって変数 V1 の型が陰に示される。このメッセージの引数の型情報はオブジェクト A の実体の型に含まれているので、間接的な表現である。しかし、積極的に入出力の情報を分離して表現できる方が良い。

いま、図 3.4 に示すように、タートルグラフィックのオブジェクトが、その描画対象であるウィンドウを代入する変数と、タートルの位置やペンの状

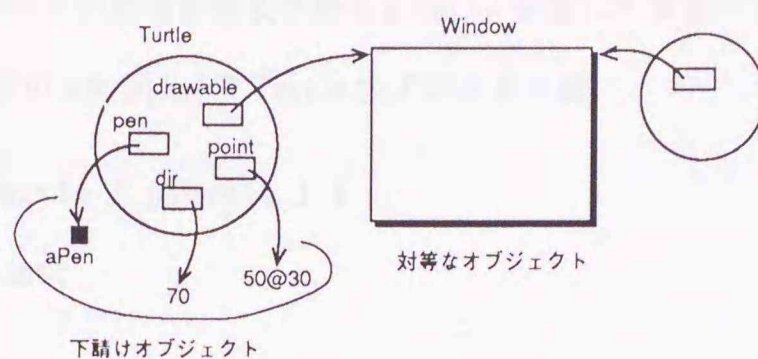


Figure 3.4: 情報隠蔽

態などの変数を持っているものとする。この場合、後者はタートルの実現に依存する変数であるから隠蔽されるべきであるが、前者はこのオブジェクトの実現に依存しない変数であり隠蔽する必要は無い。この2種類の変数の違いはそこに代入されるオブジェクトとの関係によって決められる。つまり、タートルとウインドウのような「対等」な関係にあるオブジェクトの場合と、ペンや位置のようにタートルの「下請け」となるオブジェクトの場合の違いである。対等なオブジェクトは別のオブジェクトからも参照される可能性があるもので、その存在を隠す理由はない。そこで、下請けのオブジェクトに関する変数のみを隠蔽し、対等なオブジェクトに関する変数は公開しても差し支えないと考えられる。

3.2.2 出力ビュー

対等なオブジェクトを代入するインスタンス変数を外部に公開し、この変数の型もオブジェクトの仕様に含めることを考える。すでに、Kamui 環境のオブジェクトには、メッセージプロトコルを複数集めた集合によって実体の型が付けられていることを述べた。その拡張として、公開する変数の型をオブジェクトの出力のビュー（すなわちプロトコル）と捉え、入出力のプロトコルの集合を以てオブジェクトの仕様と定義する。実体の型の包含関係は、この集合の包含関係によって定義される。

オブジェクトの型宣言の変化を Kamui-C を通じて具体的に見てみよう。
送出の型宣言がない図 3.4 の Turtle オブジェクトは

```
defclass Turtle [ pTurtle ] {  
    Point point;  
    int dir;  
    int pen;  
    pDraw drawable;  
}
```

と宣言する。ここで、[] 内には複数のプロトコルが記述でき、Turtle は 1 つの入力ビューを持っている。pTurtle はタートルグラフィックに関するプロトコルである。変数 drawable にアクセスする方法として、

```
send setDrawable(pDraw aDrawable);  
pDraw getDrawable();
```

のようなメソッドとして用意しなければならないが、ここでは pTurtle に含まれていると仮定しよう。

次に、変数 drawable を公開することになると、

```
defclass Turtle [ pTurtle , -pDraw drawable ] {  
    Point point;  
    int dir;  
    int pen;  
}
```

のようになる (図 3.5)。-pDraw は送出のビューを示しており、pDraw が送出プロトコルの型となる。drawable は内部的にはインスタンス変数名であり、外部的には出力ビューの方向を示す名前の両方の意味を持っている。公開し

た変数 `drawable` は C++ における `public` 宣言と似た扱いとなり、外部からアクセスが可能となる。そのため `pTurtle` は変数 `drawable` のアクセスメソッドを含む必要が無く、純粋なタートルの操作のみの宣言となる。これは `pTurtle` の再利用性にも寄与する。

公開された変数へのアクセスはオブジェクトの実体の型を知っている場合にのみ行なうことができる。しかし、通常は参照の型の範囲でオブジェクトが使用されているので、実体の型は制限されたレベル（例えばオブジェクトの生成等）でのみ知ることができる。それ故、このアクセスすなわちオブジェクト間の接続は通常の実行より制限された形で安全に行なわれる。

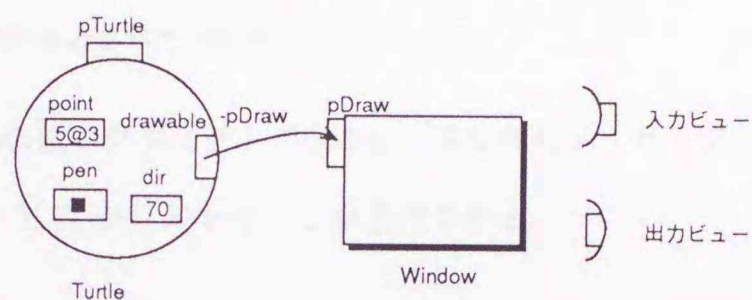


Figure 3.5: 入出力ビュー

3.2.3 ビューの名前

Kamui-C では、出力ビューに付けられた名前は直接にインスタンス変数の名前として用いられたが、他の言語でもそうである必要はなく、間接的にそれぞれの対応がつけばよい。オブジェクトを部品として見た場合には、ビューの名前は変数名としてではなく、メッセージを送出する方向を意味する。つまり、型が付けられた各ビューは他のオブジェクトとの接続のためのプラグとして見ることができる。

例として、図 3.6 のような UNIX のフィルタコマンドはそれぞれ、次のような実体の型を持っている。


```
[pStream stdin,-pStream stdout,-pStream stderr]
```

それぞれのビューに付けられた名前は他のオブジェクトとの接続の際に指定するために用いられる。入力ビューへの名前は合成されたオブジェクト群の型を考える際に必要である。例えば、2つのフィルタ f1、f2 の接続は、

```
f1.stdout = f2.stdin
```

```
f1.stderr = display
```

```
f2.stderr = display
```

のような記述になり、合成されたオブジェクトの型は

```
[pStream stdin,-pStream stdout]
```

となる。この合成オブジェクトの stdin へ送られたメッセージは f1 へ送られ、stdout を通じて f2 からメッセージが送出される。

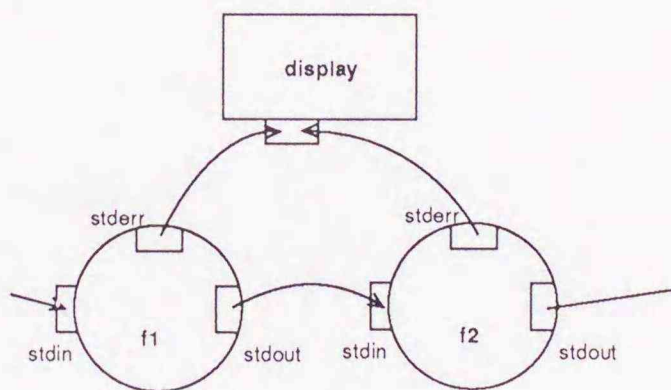


Figure 3.6: UNIX のフィルタ

ここで述べた情報はオブジェクトの寿命管理にも用いることができる。公開されない下請けオブジェクトの管理は親オブジェクトが一括して行なうことができる。それ以外の対等なオブジェクトの寿命管理はそれらの接続関係が複雑であるから、分散ガーベジコレクション (GC) などの支援が必要になる。GC ではオブジェクトが参照している他のオブジェクトをたどるための

情報が必要であるが、送出メッセージの変数を正しく外部に公開しているならば、GC はオブジェクトの実現を知ることなく行なうことができる。言い換えると、非均質なオブジェクトの GC のためにオブジェクトの接続関係の情報は不可欠なのである。

通信の構造化

Chapter 4

通信の構造化

ここでは、分散システムをマルチエージェントの一つのモデルとして捉える。マルチエージェントモデルは並行オブジェクトモデルの拡張として見ることができ、ごく新しい研究分野である。

この章で提案するモデルは複雑な通信を構造化するものである。ここで扱われる複雑な通信とは、

- 通信相手の柔軟な指定
- 複雑な通信系列
- 複数間の通信

などを扱うことができるものである。このような通信に関する情報はこれまでの並行オブジェクトモデルでは、各オブジェクトのメソッドの中に記述されていた。このように見通しが悪く、間接的な表現では、誤りのないプログラミングや誤りの検出などに不適當である。より安全な分散システムの構築の為にこれらの情報をより直接的に表現する必要がある。

4.1 問題設定

最初にそのモデルの説明を行なう前に、目標とする問題を明確にしておく。

4.1.1 通信相手の柔軟な指定

エージェントが通信を行なう場合、その通信相手を柔軟に指定できる方がよい。オブジェクト指向モデルでは、通信相手は自分の状態変数から直接指されている。並列計算モデル Kamui88 では、ある性質のイベントが必要なオブジェクトはそのイベントが放送されている場に自ら入ることによって、自分の意志による通信の制御を行なうことができる。

Linda モデル [51] とその拡張では、タプルスペースを通じたタプルのやり取りの機能により、同様の通信の制御に加えて、排他的な通信をも表現できる。松岡ら [50] はオブジェクト指向分散計算とタプルスペースによる通信の融合を試みており、型付きのタプルを導入している。Cellula[49] は Linda の拡張モデルであるが、各実行単位（これをセルと呼ぶ）が固有のタプルスペースを持つことにより、直接通信相手を指定することが可能である。さらに、各セルの階層がタプルスペースのスコープに対応するため、分割統治法の問題解決に適している。明石ら [8] もオーバーラップしたタプルスペースにより分散計算への拡張を行なっている。

4.1.2 複雑な通信系列

このタプルスペースを介した通信では、一回のタプルのマッチングにより、そのタプルに付けられた情報が交換される。しかし、多くの問題ではより複雑な通信系列が重要な意味を持っている。つまり、以上のようなモデルでは何度かのマッチングやメッセージ送出により、そのレベルの意味を表現している。

通信が正確に行なわれる為には各エージェントがある決められた通信プロトコルを守って記述される必要がある。CCS などのプロセス代数[48]はこのような通信の系列を形式的に扱うことができる。すなわち、CCS によって複雑な通信系列を表現でき、それを各エージェントの通信仕様とすることができ。Nierstrasz ら [36] はこのプロセス代数の記述がオブジェクトの型として扱うことが可能であることを示した。Lotos[56] は CCS に抽象データ型を導入した分散仕様記述言語である。その他にプロセス代数によらない幾つかの方法で通信系列を表現した例 [52] がある。

CCS では 1 対 1 同期通信を行なっているが、幾つかの拡張もある。ECCS[54] は CCS を非同期通信に対応できるように拡張したものである。川口ら [53] は放送を導入している。

それぞれのエージェントの通信仕様を CCS で記述した場合、エージェント間の通信可能性を調べる必要がある。通信可能とは、2 つの CCS 式を並行に動作させた際、外部との通信無しに、終了もしくは同じ状態へループするかのどちらかをいう。例えば、

$$1 : A \text{ where } A = x.A + e$$

$$2 : \bar{x}.x.\bar{e}$$

の 2 つの CCS 式は通信が可能である。

4.1.3 複数のエージェント間の通信

複雑なエージェントの問題では 1 対 1 通信のみならず、複数のエージェント間でのやり取りが重要な場合も多い。分割統治はその例であるが、もっと非均質な問題もある。例えば、Smalltalk-80 の GUI は Model , View , Controller の 3 つのオブジェクトによって構成される (MVC モデル)。3 つのオブジェクト間での通信は決められた通信プロトコルにより行なわれているのである。

が、それらは各メソッドのなかに組み込まれている。ただし、新しくユーザがプログラムを作る場合には各 MVC のサブクラスとして作成するので、それらのプロトコル自身も継承され、意識する必要は少ない。しかし、MVC のような構成を最初から作成する場合には、通信系列のレベルでの設計が必要であり、それなしにオブジェクトを直接作って行けば、発見しにくい誤りを生むことになる。

このような複数のエージェント間の通信も CCS により表現できる。この記述と各エージェント自身の仕様記述との関係を定義する必要がある。

4.1.4 多重ビューを持ったエージェント

各エージェントの働きが高度であるほど、それぞれは複数の仕事を同時にこなすことになる。前節の通信系列は複数のエージェントにまたがった1つの仕事を表現していた。それに対し、各エージェント単位での複数の仕事をまとめた記述が必要である。

その記述が必要な一つの理由は、各エージェント内での並列性にある。エージェントが複数の仕事を並列に実行できるのであれば、それぞれの仕事の通信系列の仕様は独立していてかまわない。しかし、それぞれの仕事で共有している資源があるならば、その資源に対する排他制御が必要になる。それは、従来からある並列プログラミングの記述が必要である。また、並列性を禁止するエージェントであれば、エージェント自身の全体の通信を記述し、それを複数のビューへの通信に分解する。

4.2 タプルの定義

タプルは、CCS 式に拡張をしたものである。最初に必要なことのみ簡単に CCS について説明し、次にそれを拡張してタプルの説明を行なう。

4.2.1 CCS

CCS はプロセスが入出力するイベントの系列のみに着目した代数である。

1. イベントの送受

イベントの受信は a, b などの文字で表わし、送信は $\bar{a}, \bar{b}$ のように上線をつけて表わす。

2. 直列

イベントの送受が直列に起こるときは、 $a.b.\bar{c}$ のように $.$ で繋げて表わす。

3. 分岐

イベントの送受に幾つかの可能性があるとき、それを、 $a+b$ のように $+$ で表わす。これは、 a か b のどちらかのイベントを受信する、という意味である。

4. 内部計算

プロセスの内部で何らかの計算を行なう場合には、それを τ で表わす。

分岐は拡張されて、 $\tau.\bar{a} + \tau.\bar{b}$ はプロセスの内部状態に応じてどちらかに分岐され、その結果 a か b を送出する意味となる。

5. 変数

繰り返される系列を記述するために変数を導入する。 $a.e, a.a.e, a.a.a.e$ などを表現するには、 $A; A = a.A + e$ と記述する。ここで、 A が導入した変数である。

このほかに並列演算などが定義されているが、省略する。

4.2.2 イベント

イベントには型 (int , char , etc.) が付いたものと、付いてないもの (これをシグナルイベントと呼ぶ) がある。 a はシグナルイベント、 $x:\text{int}$ は int 型

のイベントである。シグナルイベントはそのイベント名が重要であるが、型付きイベントはその名前よりも、その型自身が重要である。 $a.x:int.y:int$ は3つのイベントであるが、メッセージ $a(int, int)$ を表現しており、 x, y は仮引数に相当する。

イベントの送受する方向を明示するために、そのビューを指定することができる。 $\bar{a}/v1.\bar{b}/v2$ は $v1$ の方向にイベント a を送出し、次に $v2$ の方向にイベント b を送出している。ビューについては後で説明する。

4.2.3 τ が先頭に付いた $+$

CCS での $+$ の意味は、どちらかに遷移するということであるが、遷移の動機により2つに分類する。 $a.X + b.Y$ の意味は a か b のどちらかのイベントが送られてきた場合 X か Y に遷移する。これは、遷移の主導権が自分のプロセスにない場合である。それに対して、 $\tau.\bar{a}.X + \tau.\bar{b}.Y$ は最初にどちらかに τ で遷移してから、イベント a か b を送出する。この遷移の主導権は自分にある。この2種類の遷移の違いを明確にするため、後者を新しい演算 $\&$ として定義し、後者の例は、 $\bar{a}.X \& \bar{b}.Y$ のように書くことにする。すなわち、

$$X \& Y \equiv \tau.X + \tau.Y.$$

である。

4.2.4 記法の変更

CCS 式を実際のプログラム言語で用いるためにその構文を定義する。これは特殊記号や上線などを用いないものでもある。

- 送出イベント

イベントの送出は $\bar{a}$ と表現していたが、これを $-a$ と記述する。 $-$ は前置の単項演算子である。

- 変数

CCS において、変数を用いて記述された式は、

$A; A = a.X + b; X = x.X + e$

と記述する。ここでの演算子 $;$ は左側にタプルをとり、右側には左のタプルで未定義である変数の定義を行なう。通常この変数は大文字で始まることとする。

- 再帰

繰り返しを単純な再帰で表現することが多いのでその糖衣を定義する。

$A; A = a.A + b$ のような再帰は、再帰パターンにより

$[a.\textcircled{0} + b]$

と記述する。 $\textcircled{0}$ は再帰呼び出しで、最も内側のブロック $[]$ へ再帰する。

演算子の適用順序をまとめると、

$: > / > - > . > \& > + > = > ;$

となる。

例えば、次のようなものがタプルの記述である。

$\text{add}.x:\text{int}.y:\text{int}.-z:\text{int}$

これは、関数 $\text{add}(x:\text{int}, y:\text{int}):\text{int}$ の型に相当する。その関数を呼び出すには、

$-\text{add}.-x:\text{int}.-y:\text{int}.z:\text{int}$

である。

$[c:\text{char}.\textcircled{0} + \text{eof}]$

これは、入力ストリームを表わしている。出力ストリームは、

[-c:char.0 & -eof]

である。入力ストリームは相手の都合でeofを受信するので、+が用いられているが、出力ストリームはこちらが終了を決定するので、&が用いられることに注意してほしい。

4.3 タプルのマッチング

最初に、2つのタプルのマッチングを定義する。後で、複数のタプルによるマッチングにそれを拡張する。ビューの属性はここでは省略する。

2つの演算子+, &の扱いを用意にするため、これらを集合を引数とする単項演算子として扱うことにする。集合の要素はこれらの演算子で結合されたそれぞれの引数である。すなわち、

$$a + b + c + d \equiv +\{a, b, c, d\}.$$

である。さらに、+と&が混ざった式では&の方が結合が強いので、

$$a + b \& c + d \equiv +\{a, \&\{b, c\}, d\}.$$

である。

4.3.1 2つのタプル

2つのタプルX, Yのマッチング手続き $\text{match}(X, Y)$ を定義する。これはCCSの立場からは2つのプロセスを並列に実行させて、それらが停止することを判定することである。

1. X, Yがイベントのとき

- (a) XとYが同じ名前でも反対の符号がついたシグナル型イベントのときは真。例えば、 $\text{match}(a, -a)$ は真。

(b) X と Y が同じ型の型付きイベントで反対の符号が付いている場合は真。例えば、 $\text{match}(a:\text{int}, -b:\text{int})$ は真。イベントの名前、 a, b は異なってもよい。

(c) それ以外は、偽。

2. X, Y が直列のとき

列の各要素が match する場合、真。

$$\begin{aligned}\text{match}(x.X_r, y.Y_r) = \\ \text{match}(x, y) \wedge \text{match}(X_r, Y_r).\end{aligned}$$

3. X と Y が+のとき

$$\begin{aligned}\text{match}(+X_s, +Y_s) = \\ \exists x \in X_s \exists y \in Y_s \text{match}(x, y).\end{aligned}$$

4. 一方が+で、他方が&のとき

$$\begin{aligned}\text{match}(+X_s, \&Y_s) = \\ \forall y \in Y_s \exists x \in X_s \text{match}(x, y).\end{aligned}$$

5. 一方が+で、他方がそれ以外のとき

$$\begin{aligned}\text{match}(+X_s, Y) = \\ \exists x \in X_s \text{match}(x, Y).\end{aligned}$$

6. いずれかが再帰のとき再帰は変数の表記に変換して行なう。

$$\begin{aligned}\text{match}([X], Y) = \\ \text{match}((@; @ = X), Y).\end{aligned}$$

7. 一方のみが変数を含む式のとき他方を満足させるのに十分、変数を展開し行なう。

8. 両方が変数を含む式のと看それらの変数がマッチすると仮定し、式の
右辺に対し `match` を呼び出す。その `match` のなかでそれらの変数の
マッチングが生じた場合は、それを展開せずにそのまま真とする。それ
以外で現れた変数は、7. により展開する。

例を見てみよう。次のタプルの対はマッチする。

<code>a.-x:int</code>	<code>-a.y:int</code>
<code>a+b+c</code>	<code>-c + -d</code>
<code>a&b</code>	<code>-a + -b + -c</code>
<code>[c:char.0 + eof]</code>	<code>[-c:char.0 & -eof]</code>
<code>A; A = a.A + e</code>	<code>-a.-a.-e</code>
<code>A; A = a.A + e</code>	<code>B; B = -a.-a.A & -e</code>

最後の例をたどってみよう。

```
match((A; A = a.A + e),
      (B; B = -a.-a.B & -e))
8. により A = B を仮定
match(a.A + e, -a.-a.B & -e)
match(a.A, -a.-a.B)      match(e, -e)
match(A, -a.B)            true
match(a.A + e, -a.B)
match(a.A, -a.B)
match(A, B)
true
```


4.3.2 複数のタプルのマッチング

複数のタプルのマッチング mmatch を定義する。 mmatch はタプルの集合を引数にとる。要素が一つの場合はマッチするものがないので偽であるし、要素が二つの場合は match を呼び出す。そこで、要素が二つ以上の場合について考える。

1. 要素の一つが+のとき

$$\begin{aligned}\text{mmatch}(\{+X_s\} \cup Y) = \\ \exists x \in X_s \text{ mmatch}(\{x\} \cup Y).\end{aligned}$$

2. 要素の一つが&のとき

$$\begin{aligned}\text{mmatch}(\{\&X_s\} \cup Y) = \\ \forall x \in X_s \text{ mmatch}(\{x\} \cup Y).\end{aligned}$$

3. 要素の二つがイベントの直列か1つのイベントのとき

$$\begin{aligned}\text{mmatch}(\{x.X_s, y.Y_s\} \cup Z) = \\ \text{match}(x, y) \wedge \text{mmatch}(\{X_s, Y_s\} \cup Z).\end{aligned}$$

X_s, Y_s が空のときはとりのぞく。

4. 要素の一つだけの変数のとき

必要に応じて適当に展開する。

5. 要素の二つ以上の変数のとき

すべての変数からはじまる要素をとりだし

$$\begin{aligned}\text{mmatch}(\{(A; A_s), (B; B_s), \dots\} \cup Z) = \\ \text{mmatch}(\{A, B, \dots\}) \text{ を仮定して} \\ \text{match}(\{A_s, B_s, \dots\} \cup Z).\end{aligned}$$

6. 要素の一つが再帰のとき

変数を導入する。

ここで、マッチする例を見てみよう。

```
mmatch({ a.b, -a, -b })
```

```
mmatch({ a.b.a.b, -a.-a, -b.-b})
```

```
mmatch({ a+b, -a })
```

```
mmatch({ a&b, -a, -b})
```

```
mmatch({ [ -a.-b.Q & -c.-d] ,  
          [ a.Q + c], [ b.Q + d]})
```

最後の例をたどってみよう。

```
1: mmatch({[-a.-b.Q & -c.-d],  
           [a.Q+c], [b.Q+d]})
```

```
2: mmatch({A;A= -a.-b.A & -c.-d,  
           B;B= a.B+c,  
           C;C= b.C+d})
```

3: mmatch({A,B,C}) を仮定

```
mmatch({-a.-b.A & -c.-d,  
        a.B+c,  
        b.C+d})
```

```
4: mmatch({-a.-b.A, a.B+c, b.C+d}) and  
   mmatch({-c.-d, a.B+c, b.C+d})
```

```
5: mmatch({-a.-b.A, a.B, b.C}) and  
   mmatch({-c.-d, c, d})
```

```
6: mmatch({-b.A, B, b.C}) and true
```

```
7: mmatch({A, B, C})
```


8: true

ここで4:から5:へは+の展開で、ここに表れていない可能性はタプルに現れるイベントが対をなさないので、マッチしないことがわかり、あらかじめその枝を刈ることができる。例えば、`mmatch({-a.-b.A, c, b.C})`ではイベント-a, cに対応するa と -cがないため、これはマッチしないことがすぐわかる。

4.4 エージェントモデル

エージェントは自分が希望する通信プロトコルを表現したタプルを特定のタプルスペースに書き込む。タプルスペースは各エージェントより書き込まれたタプルにより、エージェント間の通信相手と通信プロトコルの選択を行なう。それらが決定された後は、タプルスペースから離れて、エージェント間のみで通信を行なう。

エージェントは自分が書き込んだタプルに関して正確にそれを満足するように動作する必要がある。タプルの記述が静的であるならば、自分の動作の保証はコンパイル時に行なうことができる。

各エージェントはタプルと対になる実行文を持っている。タプルがマッチし、通信相手が特定されるとその実行文が起動される。タプル自身の示している情報に加えて、マッチング時により詳しいプロトコルの情報が付加される場合がある。実行文中でこのような情報を用いて分岐する制御文が用意されている。

タプル上の型付きイベントのイベント名は実行文では未来変数に対応する。イベントの送信は未来変数への代入に、イベントの受信は未来変数の参照に相当する。変数のスコープを制御するための構文が用意されている。

行動の記述は通常のプログラム言語で行なう。ここでは、C言語を拡張し

た文法を用いる。

4.4.1 タプル文

まず、タプルを送出する構文を与える。

<式> '←' '(' <タプル式> ')' <文>

最初の式は送出先のタプルスペースを示す。次のタプル式は、前の節で定義した記法を用いたタプルである。最後の文は、このタプルのマッチングが行なわれた際に実行される部分である。

例として、並行オブジェクトを実現してみよう。

```
object(TS self) {  
  int value = 0;  
  for(;;) {  
    self←(init+ add.x:int+ get.-x:int) {  
      select {  
        init: value = 0;  
        add: value += x;  
        get: x = value;  
      }  
    }  
  }  
}
```

value はインスタンス変数に相当する。このオブジェクトは3つのメソッドを持っており、その型をタプルにしてタプルスペース self に送出している。self は各エージェントに1つずつ存在するタプルスペースで、Cellula[49]の影響を受けている。select は各イベントのどれかの受信による分岐で、いずれか

のイベントを受信した際に、それに対応する文を実行する。変数のスコープも対応するタプルにより決められる。ここでは `add` と `get` の両方に変数 `x` が用いられているが、これらは異なった変数である。

4.4.2 未来変数

受信のイベントは、タプル送出的実行文の中では未来変数として扱われる。前出で定義したオブジェクトを使用する側のプログラムを記述してみよう。

```
test(TS object) {  
  object <- (-get.x:int) {  
    -get;  
    /* any works */  
    printf("his value is %d.\n", x);  
  }  
}
```

ここで `object` には相手のタプルスペースが入れられているものとする。タプル `-get.x:int` は相手のタプルとマッチする。`-get` はイベント `get` を送出する。その返答の到着のまえに何か仕事をすることができ、その値が必要になる `printf` 文でその実行がブロックされる。この例は ABCL の未来型通信に対応している。

4.4.3 再帰

タプル内の再帰の記述は、そのまま再帰文により表現する。例として配列オブジェクトを定義する。これは、メッセージ `size` によりその大きさを返し、`contents` によって自分の内容の列と終わりを示す `end` をメッセージとして送り返すものである。

```
array(TS self,int mySize) {
```

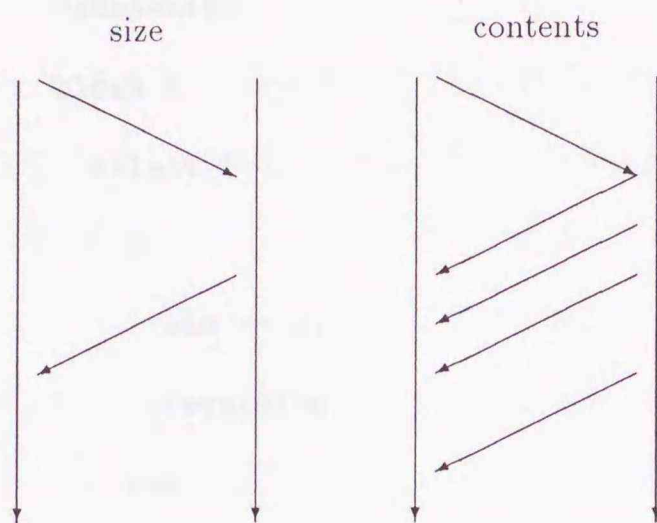


```

int body[mySize];
for(;;) {
    self<-(size.-x:int+
        contents.[-x:int.@ & -end]) {
        select {
            size: x = mySize;
            contents:
                int i = 0;
                block {
                    if (i < mySize) {
                        x = body[i++];
                        recursion;
                    } else {
                        -end;
                    }
                }
            }
        }
    }
}

```

block は再帰の始まりを示し、その中のrecursion により再帰する。変数 x が再帰中で用いられる変数であり、これがrecursion により更新される。



4.4.4 送出の select

相手の仕様によってこちらが送出する内容が変化する場合がある。例えば、前出のarrayのように内容を一度に送ることができる列と、`get.index:int.-value:int`のプロトコルで要素一つずつにアクセスすることしかできない列とがあったとする。そのいずれかによらずその内容の合計を求めることを考える。

```
int sum(TS array) {
  int size;

  int sum = 0;

  array <- (-size.x:int) {
    -size;

    size = x;
  }

  array <- (-contents.[x:int.@ + end] +
    -get.-index:int.value:int) {
    select {
      -contents:
        int i = 0;
```



```

-contents;

block {
  select {
    x:
      sum += x;
      recursion;
    end:
      return sum;
  }
}

-get:
-get;
index = 0;
sum = value;
for (int i=1;i<size;i++) {
  array<-(-get.-index:int.value:int) {
    -get;
    index=1;
    sum += value;
  }
}
return sum;
}
}
}

```


ここで、`select` はどちらのイベントが送出可能であるかによって分岐する。しかし、この分岐はタプルのマッチング時にすでに決定されている。タプルのマッチングの定義によると2つのタプルの先頭がどちらも+であるとき、その枝のなかで一つでもマッチングするものがあればよかった。そこで、マッチング時にどの枝を用いてマッチングが成功したのか、という情報がそれぞれのエージェントに伝えられ、それに従ってこの`select` は分岐を決める。もし、マッチする枝が複数存在するときには、それが非決定的に選択される。実用的には効率のよいプロトコルが選択されるべきであるので、選択の優先順位をタプルに付加することも考えられよう。

CCS では同期通信を行なっているために、動的に送出可能な分岐を決定することができる。我々のモデルがマッチング時に選択を決めてしまうのは、その実現レベルで非同期通信を考えているからである。そのため、送出の`select` が再帰などの繰り返しのなかに存在するのであれば、毎回同じ枝を使用することになる。

この例で、後者の`get.index:int.-value:int`によるインタフェースは、効率を犠牲にしているだけではなく、毎回タプルスペースへマッチングを行なっているので、いつも同じエージェントと通信できるか判らないという欠点がある。また、一連の処理の始まりと終わりとが判断できないため、そのエージェントの内容を更新する場合に必要な排他制御も難しい。

4.4.5 多重ビュー

エージェントは多重ビューを持っている。一つのビューはエージェントの一つの仕事の側面である。エージェントの外部仕様は各ビュー単位で公開されるが、エージェントの内部では各ビューが複雑に混じって記述される。

例として3種類のUNIXのフィルタコマンドを考えてみよう。`cat` コマンドは標準入力データをそのまま標準出力へ出す。`cat` のバッファは1文字

であるとすれば、標準入力からの1文字の入力と標準出力への1文字の出力とが交互に起こる。grep コマンドは標準入力を行単位で読み、指定されたパターンに一致する行のみを標準出力に書き込む。これは、複数の文字の入力と複数の文字の出力とが交互に起こっている。最後に、sort コマンドは標準入力からのデータをファイルの終わりまで完全に読み込み、それをソートして標準出力へ書き込む。これは、入力の処理が完全に終わってから、出力の処理に入る。

ここで考えた3つのコマンドは標準入力と標準出力という2つのビューを持っており、それぞれのビューの通信仕様は次のように同じものである。

入力 [c:char. @ + eof]

出力 [-c:char. @ & -eof]

sort は2つの処理が完全に分離されているので、これまでに述べた方法で記述できる。しかし、sort 以外のコマンドは入力と出力の処理を同時に行なっているので、新しい記法が必要である。そこで、最初にコマンドの2つのビューを合成した完全な仕様を書き、それから上の2つの仕様を生成する。3つのコマンドの仕様は次のようになる。

cat: [c1:char/stdin.-c2:char/stdout.@
+ eof/stdin.-eof/stdout]

grep: [c1:char/stdin.
(@ & [-c2:char/strout.@ & 1].@)
+ eof/stdin.-eof/stdout]

sort: [c:char/stdin.@ + eof/stdin].
[-c:char/stdout.@ & -eof/stdout]

ここで各イベントには新しくビューの属性が付加された。grep での1は空イベントで何も実行しない意味である。この記述がエージェントの完全な仕様

であり、これに従ってエージェントのプログラムが記述される。cat のプログラムを記述しよう。

```
cat(TS aStdin, TS aStdout) {  
    (aStdin,aStdout) <- (stdin, stdout):  
    [ c1:char/stdin.-c2:char/stdout.@  
      +eof/stdin.-eof/stdout ] {  
    block {  
        select {  
            c1/stdin:  
                c2/stdout = c1/stdin;  
                recursion;  
            eof/stdin:  
                -eof/stdin;  
        }  
    }  
}
```

ここで、最初のタプル送出の構文はビューを指定できるように拡張され、2つのタプルスペースに関連するタプルに分解して送出する。ビューに応じた分解は後で説明する。このプログラムの本体が起動されるのは、2つのタプルスペース aStdin, aStdout へ送出したタプルが両方マッチしたとき初めて起動される。しかし、このマッチングはそれぞれ独立に行なわれてよく、すでにマッチングした相手のエージェントはこちらの他方のマッチングに関係なく、実行を開始できる。つまり、ビューに応じた分解まではエージェントの内部仕様であり、外部仕様としてはあくまで分離されたビュー単位で扱われる。この意味で、3つのコマンドは区別できない。

タプルのビューへの分解は、次のように行なう。

1. 必要なビューに無関係なイベントを τ で置き換え、ビューの情報を省略する。
2. $\&$ を τ を用いた $+$ に変更する。
3. $+$ の先頭以外にある τ を省略する。
4. 可能な $\&$ を導入する。

grep の仕様でstdin に関するタプルを分解してみる。

```
[c1:char/stdin.  
  (@ & [-c2:char/stdout.@ & 1].@)  
+ eof/stdin.-eof/stdout]  
[c1:char.(@ & [ $\tau$ .@ & 1].@) + eof.  $\tau$ ]  
[c1:char.( $\tau$ .@ +  $\tau$ .[ $\tau$ .@ +  $\tau$ ].@) + eof.  $\tau$ ]  
[c1:char.( $\tau$ .@ +  $\tau$ .@) + eof]  
[c1:char.@ + eof]
```

ここで述べた多重ビューを混ぜた記法に対して、エージェント内で複数のスレッドが実行できるようにすれば、分離されたタプルだけで多重ビューのエージェントは表現できる。しかし、スレッド間の排他制御が必要になる。ここでのビューをまとめたタプルの表現は、各ビューごとの仕様と、ビュー間の実行の制御の両方を表現していると考えられる。

4.4.6 多重スレッド

並行オブジェクトの議論でも、多重スレッドの導入が問題となった。そこでは、オブジェクト内にスレッドを導入することにより得られる並列性の向上と、それによって失う記述の見通しの悪さのトレードオフが議論されたの

である。幾つかの並行オブジェクト指向言語では多重スレッドは許されていない。しかし、粒度が大きくなると並列性が重要となってくるので、OS によってはLWP(Light Weight Process)として多重スレッドを用意しているものもある。

マルチエージェントの問題ではエージェントの大きさがオブジェクトよりも大きくなることが予想されるので、多重スレッドが必要になる可能性は高い。

ここでは、安全な多重スレッドの導入の例を示す。先程の列のエージェントに要素単位の更新と読みだしの機能を追加する。

```
array(TS self, mySize) {  
  int body[mySize];  
  self <-& ( get.index:int.-value:int  
             +size.-x:int) {  
    select {  
      get:  
        if (index>=0 && index < mySize)  
          value = body[index];  
        else value = 0;  
      size:  
        x = mySize;  
    }  
  }  
  for(;;) {  
    self <- (contents.[-x:int.@ & -end]  
            + put.index:int.value:int) {
```



```

        /* 省略 */
    }
}
}

```

ここで導入した新しい構文`<-&`はタプルを永続的に送出するものである。通常のタプルは一度マッチするとそれは消滅するが、このタプルは何度でもマッチし、それに結びつけられた実行文は別なスレッドとしてエージェント内を並行に動作する。`get` と `size` は読み出しのメソッドのため、いつでも実行が可能である。それに対し、`put` と `contents` とは一つのスレッドとして動作するため、`contents` の最中に `put` は生じない。

ただし、この方式でも複数の `contents` の並行実行などはできない。より、細かい制御はタプルへの付加情報として必要であろう。

4.5 複数エージェントの通信

一般には、2つのエージェントだけの1対1通信だけではなく、複数のエージェントが協調動作している。それにより、タプルは対となってマッチングが行なわれるだけではなく、複数のタプルのマッチングについて考える必要がある。

4.5.1 通信の抽出

まず、タプルスペースを1連の仕事の申し合わせをするところとする。つまり、各エージェントは自分の実行しようと思っている仕様をタプルとしてタプルスペースに出力し、複数のタプルがマッチングした時点でその所有者達が実行を開始する。

3つのエージェント `a`, `b`, `c` が次のような順に通信を行なっている例を考えよう。

$t1 : a \rightarrow b$

$t2 : b \rightarrow a$

$t3 : a \rightarrow c$

$t4 : c \rightarrow b$

ここで、各エージェントの仕様は、

a $-t1.t2.-t3$

b $t1.-t2.t4$

c $t3.-t4$

のようになる。この3つのタプルが一つのタプルスペースに存在したとき、エージェントが動作を開始する。例えば、aとbのタプルがタプルスペースに存在しているとき、それらだけではマッチングが成立しないが、cのタプルが書き込まれたとき3つのタプルがマッチする。

このようなタプルの組みは複数のエージェント間の通信の仕様であるが、それを次のように宣言する。

formation $f(a, b, c) =$

$(a \rightarrow b)t1.(b \rightarrow a)t2.(a \rightarrow c)t3.(c \rightarrow b)t4$

これは、3つのエージェントによる通信をfという名前で抽象化したもので、多重ビューを導入したエージェントの記述と直交するものである。この記述から各エージェントへの仕様を生成することができる。タプルには並列動作の記述が含まれていなかったが、この記述には必要となる。

4.5.2 通信の型

エージェント間の通信に着目して記述することにより、それを型として扱うことが可能となる。マッチングアルゴリズムは複雑で処理に手間がかかるものであるが、この型を利用するとマッチングは極めて低コストとなる。

前述の f と宣言された通信の型に対して、3つのタプルに $f::a, f::b, f::c$,
 のような型を付けることができる。マッチングの判定はこれら3つがそろっ
 ているか否かで簡単に決定できる。これは通信の型が静的である場合に可能
 である。

4.5.3 型階層間のマッチング

型付きイベントにおける型には、一般の型概念と同様に型階層を考慮するこ
 とができる。その結果、型の階層間のマッチングを考える必要がある。それ
 によって、前出のマッチングの定義は修正される。

一般の型概念と同様に、イベントを送信する側の型は受信する側の型より
 狭い、すなわち階層の下に属している必要がある。型の関係を $\subset$ で表わそう。
 例えば、整数 int と奇の整数 $oddint$ には $oddint \subset int$ の関係がある。した
 がって、マッチングの定義は次のように変更される。

$$\text{match}(x:t_x, -y:t_y) = t_y \subseteq t_x.$$

4.5.4 複数のエージェント間の通信における型階層

複数のエージェントを組み合わせにより、一つのエージェントとして振る
 舞わせることができる。その際、それらのエージェントを使用する側には一
 つのエージェントと区別できないようにさせる必要があるため、中継のエー
 ジェントを導入する方法が考えられる。使用する側は中継エージェントを実
 際のエージェントであるかのように扱い、中継エージェントは自分に送られ
 てきた要求を下請けのエージェント達へ分配する。しかし、この方法では機
 能の向上は得られるけれども、並列度などの向上はあまり望めない。

ここでは、中継を用いること無しにこのようなエージェントの組み合わせ
 を行なう方法を述べる。それは、送信と受信の型の関係を拡張し、マッチン
 グ手続きを変更することで得られる。

例として、配列のアクセス速度を向上させるために、2つの配列をその添え字が偶数であるか奇数であるかによって、アクセスをインターリーブさせることを考えてみよう。配列 o は添え字が奇数である要素が、配列 e は添え字が偶数である要素が入れている。アクセスのための型は、それぞれ、 $\text{get}.x:\text{oddint}$, $\text{get}.x:\text{evenint}$ である。これに対し、それを使用する側 s では $-\text{get}.x:\text{int}$ という型でアクセスをする。つまり、これらの型がマッチングできるようにしたいのである。今、型をその要素の集合として表現することにすれば、以下のような関係が成立する。

$$\text{int} = \{1, 2, 3, \dots\}$$

$$\text{oddint} = \{x | x \in \text{int}, \text{oddp}(x)\}$$

$$\text{evenint} = \{x | x \in \text{int}, \text{not oddp}(x)\}$$

$$\text{oddint} \cup \text{evenint} = \text{int}$$

evenint の定義に evenp を用いていないので、2つの間で交わりがないことと、2つの型の和が int になることが記号的に判定できる。

イベントに付けられた型が集合の形で定義されているとき、Milner[48] が行なったように、次のようなタプルの形に展開できると考える。

$$\text{type1} = \{a, b, c\}$$

$$x:\text{type1} \equiv a + b + c$$

$$-x:\text{type1} \equiv a \ \& \ b \ \& \ c$$

これにしたがって、複数のタプルのマッチング mmatch を以下のように修正する。

$$\begin{aligned} \text{mmatch}(\{-x_1:s_1.X_1, -x_2:s_2.X_2, \dots, \\ y_1:r_1.Y_1, y_2:r_2.Y_2, \dots\} \cup R) = \end{aligned}$$

$$s_1 \cup s_2 \cup \dots \subseteq r_1 \cup r_2 \cup \dots$$

$$\wedge s_1 \cup s_2 \cup \dots \cap r_i \neq \phi (i = 1, 2, \dots)$$

$$\wedge \text{mmatch}(\{X_1, X_2, \dots, Y_1, Y_2, \dots\} \cup R).$$

つまり、イベント送信側の型の和が受信側の型の和の部分集合であれば通信が可能である。2つめの条件は送信側と無関係な型を受信するエージェントの発火を防ぐ為に必要である。

このマッチングによって、実行可能な3つのエージェントが決定される。送信側 s のプログラムでは型 int のままでアクセスを行なっている。そこで、マッチングの情報により型 int から2つの型 oddint , evenint へ分離する述語 oddp を抽出する。 s の実行時に int のデータが oddp を満足するか否かによってエージェント o か e のどちらに送るのかを決定する。この処理は、明示的にインターリーブのプログラムを記述したものと等価である。しかし、元のプログラム中この処理は存在せず、マッチングの後でエージェント s の中に組み込まれるものである。つまり、インターリーブしない1つの配列や、添え字の範囲によって分けられた複数の配列¹などとも、全くの変更無しにより効率で動作可能である。ここでの例は、うまく制御された効果の大きいリフレクションの例とも考えられる。

別の例は、グラフィカルユーザインタフェースの記述 [55] である。マウスと複数のグラフィカルアイテムとの通信がタプルスペースを通じて行なわれる。各アイテムの仕様は、

`down.pt:rect(30,0,50,20).`

`[move.pt:point. -next. @ + up.pt:point]`

である。ここで $\text{rect}(30,0,50,20) \subset \text{point}$ であり、 $\text{rect}()$ は point の部分範囲型である。このアイテムは長方形の領域 $(30,0)-(50,20)$ を占めており、その範

¹これにより、配列の大きさを動的に変化させる場合、配列そのものに手を加えずに、新しい配列との組み合わせによって行なうことができる

囲で起きたマウスのボタンダウンに反応する。様々な領域からなるアイテム
が同じタプルスペースに存在し、それに対応する次のようなマウスのタプル

```
-down.-pt:point.
```

```
[-move.-pt:point.next.0 & -up.-pt:point]
```

とマッチングする。その際マウスダウンが起きた点を含んでいる領域を持つ
一つのアイテムのみが起動される。これらが GUI の意味的に正しく動作する
ためには、マッチングを行なう順序の制御（例えば一番新しいタプルから順
にマッチする）を行なう必要がある。

この例は、グループウェアでよく行なわれる複数のマウスカーソルが存在
するようなアプリケーションのよいモデルとなっている。特にここでイベン
ト next によって処理に同期がとられている点が、イベント駆動によるモデル
と異なっている。イベント駆動ではマウスの移動が非同期に行なわれ、実際
のマウスの位置との遅れが生じるか、RPC による別のパラダイムと混合して
同期をとるなどしている。

Chapter 5

開放的名前付け問題

5.1 ドメインポートモデル

この章では、DPM の定義を行なう。最初に、ここで扱っている開放系のモデルを定義し、次にそれに基づいて開放系における名前付けを定義する。

5.1.1 ドメインとポート

図 5.1 に示すように、開放系をドメインとポートによってモデル化する。ドメインは近傍を意味し、ポートによって 2 つのドメインを接続する。各ドメインにはその周辺のドメインに対する知識しか存在せず、それらの協調によってより広い範囲の様々な問題を解決するものとする。

全ドメインの集合を $\mathcal{D}$ 、全ポートの集合を $\mathcal{P}$ とすると、以下の関係が成り立つ。

$$\mathcal{P} \subseteq \{p \mid p = (d_1, d_2), d_1, d_2 \in \mathcal{D}\}.$$

ポートには方向が付けられており、ポート p が (d_1, d_2) のとき、 d_1 をポートの入口となるドメイン、 d_2 をポートの先となるドメインとする。そこで、以下の関数を定義する。

$$from : \mathcal{P} \rightarrow \mathcal{D}$$

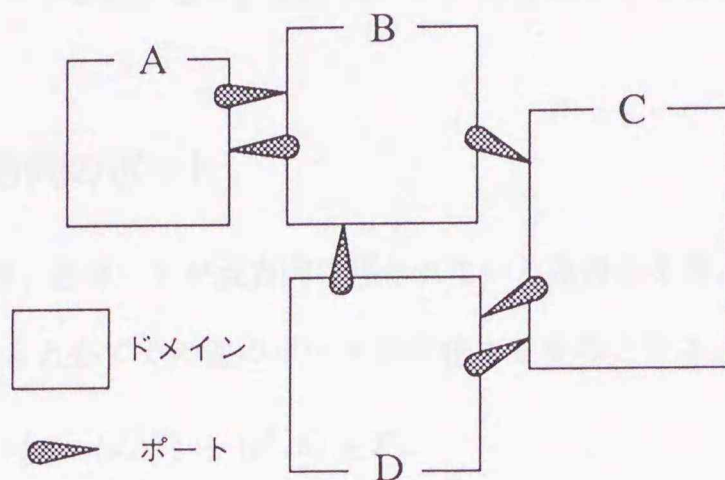


Figure 5.1: ドメインとポートの例

$$from((d_1, d_2)) = d_1.$$

$$to : \mathcal{P} \rightarrow \mathcal{D}$$

$$to((d_1, d_2)) = d_2.$$

また、ドメイン d から出ているすべてのポートの集合 P_d を定義する。

$$P_d = \{p \mid p \in \mathcal{P}, p = (d, d'), d' \in \mathcal{D}\}.$$

図 5.1 に示された各ドメインの P_d は以下のようにになる。

$$P_A = \{(A, B)\}.$$

$$P_B = \{(B, A), (B, C)\}.$$

$$P_C = \{(C, D)\}.$$

$$P_D = \{(D, B), (D, C)\}.$$

5.1.2 ポート への名前付け

各ドメインから出ているポートに対して、そのドメイン内でユニークな名前を付ける。名前の集合を $\mathcal{N}$ 、ドメイン d でのポート名の集合を $X_d \subset \mathcal{N}$ として、 P_d と X_d を 1 対 1 に対応させる写像 φ_d を導入する。

$$\varphi_d : X_d \rightarrow P_d.$$

異なったドメイン d に対して、 X_d が同一の元を含んでいても、別のポートを表わす。

5.1.3 逆方向のポート

これ以降は、各ポートが双方向に張られている系のみを考える。あるポートが与えられると必ずその逆のポートが存在するものとする。つまり、

$$\forall p \in \mathcal{P} : [p = (d, d') \rightarrow (d', d) \in \mathcal{P}].$$

である。そのような対になっているポートを

$$(d', d) = \overline{(d, d')}.$$

のように表わすことにする。すなわち、 $p = \bar{p}$ である。

次に、あるドメインにおけるポート名から、その逆方向のポート名を求める関数を定義する。

$$iname : \mathcal{D} \times \mathcal{X} \rightarrow \mathcal{X}$$

$$iname(d, x) = \varphi_{d'}^{-1}(\overline{\varphi_d(x)})$$

$$\text{where } d' = to(\varphi_d(x)).$$

図 5.2 はポートに名前を付けた例である。ここでは、理解しやすいように、あるポート名の逆のポート名には上線を付け、また全体で同じ名前が無いようにしてあるが、実際には各ドメイン内でユニークであればよい。ここでのポートとその名前関係の一部を示すと次のようになる。

$$\varphi_A(\mathbf{x}) = (A, B) = \overline{\varphi_B(\bar{\mathbf{x}})}.$$

$$\varphi_B(\mathbf{y}) = (B, C) = \overline{\varphi_C(\bar{\mathbf{y}})}.$$

$$\varphi_D(\bar{\mathbf{z}}) = (D, C) = \overline{\varphi_C(\mathbf{z})}.$$

$$iname(C, \bar{\mathbf{y}}) = \mathbf{y}.$$

$$iname(B, \bar{\mathbf{x}}) = \mathbf{x}.$$

⋮

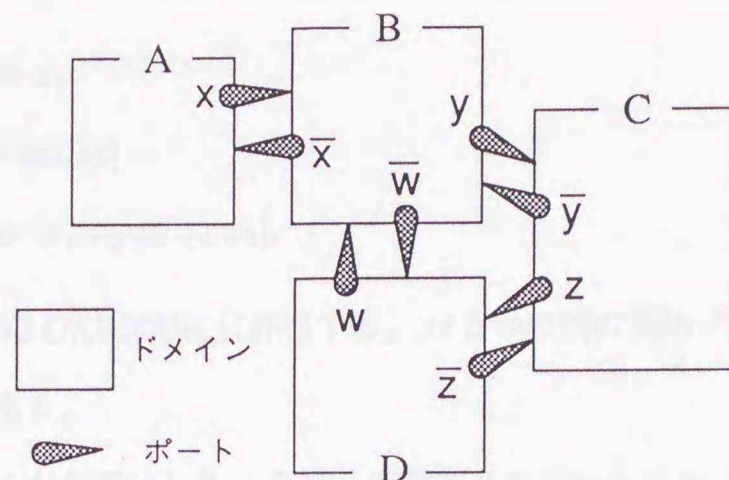


Figure 5.2: ポート名の例

5.1.4 ドメインへの名前付け

これまでの定義で、ドメインに対して A, B というような大域的な呼び方をしていた。しかし、開放的な名前付けを行なうために別の方法で名前を付ける必要がある。

ポート名はそれぞれのドメインによって局所的に管理されている。そのポート名の列 $\mathcal{N}^*$ を考える。列は以下のような形をしている。

$$n_1, n_2, n_3, n_r \in \mathcal{N}^*$$

$$x_1, x_2, x_3 \in \mathcal{N}$$

$$n_1 = [x_1].$$

$$n_2 = [x_1.x_2.x_3|n_r].$$

$$n_3 = [].$$

n_r は n_2 の残りの部分列を、 n_3 は空列を表わす。列に関して、以下の関数が定義されている。

$$n_1 = [x_1.x_2]$$

$$n_2 = [x_3.x_4.x_5] \text{ のとき}$$

$$hd(n_1) = x_1.$$

$$tl(n_2) = [x_4, x_5].$$

$$n_1 : n_2 = [x_1, x_2, x_3, x_4, x_5].$$

hd, tl は LISP の CAR, CDR に相当する。: $\cdot$ は 2 つの列の連結であり、LISP の APPEND である。

ドメイン d からポート名 x を通じて接続されているドメインを $\langle d, x \rangle$ と記述することになると、その定義は、

$$\langle \rangle : \mathcal{D} \times \mathcal{X} \rightarrow \mathcal{D}$$

$$\langle d, x \rangle = to(\varphi_d(x)).$$

となる。この定義をポート名の列に対しても適用できるように拡張すると、

$$\langle \rangle : \mathcal{D} \times \mathcal{X}^* \rightarrow \mathcal{D}$$

$$\langle d, [x|r] \rangle = \langle \langle d, x \rangle, r \rangle.$$

$$\langle d, [] \rangle = d.$$

となる。図 5.2 でドメイン A におけるポート名の列 $[x.y.z]$ は、

$$\langle A, [x.y.z] \rangle$$

$$= \langle \langle \langle A, x \rangle, y \rangle, z \rangle$$

$$= \langle \langle B, y \rangle, z \rangle$$

$$= \langle C, z \rangle$$

$$= D$$

となり、ドメイン D を指していることがわかる。このようなポート名の列は、あるドメインから他のドメインへの相対的な名前と見なすことができる。そこで、このようなポート名の列を改めてドメイン名と呼ぶことにする。ドメイン名の集合を $\mathcal{N}$ とすると、 $\mathcal{N} = \mathcal{X}^*$ である。ドメイン名を用いる局面では、

それがどこから指している名前であることを明示する必要がある。そのようなドメインを「ドメイン名の起点」と呼ぶ。ドメイン名は単純な名前の列ではなく、それぞれのドメインでの正しいポート名を繋いだものである。そこで、あるドメイン d を起点とするドメイン名の集合 N_d を定義する。

$$N_d = \{[]\} \cup \{[x|n] \mid x \in X_d, n \in N_{(d,x)}\}.$$

ここで、 $N_{(d,x)}$ は d に x を通じて隣接したドメインを起点とするドメイン名の集合である。ある起点から一つのドメインへの経路は、ポートを何度も往復するものも含めると、無限個存在する。そのために、ドメイン d から指し示すことができるドメインの数は有限であるが、 N_d は無限個の要素を持つ。例えば図 5.2 において、 N_B は次のようになる。

$$\begin{aligned} N_B &= \{[\bar{x}|n_a], [y|n_c], [\bar{w}|n_d] \\ &\quad \mid n_a \in N_A, n_c \in N_C, n_d \in N_D\} \\ &= \{[\bar{x}.x|n_b], \\ &\quad [y.\bar{y}|n_b], [y.z|n_d], \\ &\quad [\bar{w}.w|n_b], [\bar{w}.\bar{z}|n_c] \\ &\quad \mid n_b \in N_B, n_c \in N_C, n_d \in N_D\} \\ &= \{[\bar{x}.x.\bar{x}|n_a], [\bar{x}.x.y|n_c], [\bar{x}.x.\bar{w}|n_d], \\ &\quad [y.\bar{y}.\bar{x}|n_a], [y.\bar{y}.y|n_c], [y.\bar{y}.\bar{w}|n_d], \\ &\quad [y.z.w|n_b], [y.z.\bar{z}|n_c], \\ &\quad [\bar{w}.w.\bar{x}|n_a], [\bar{w}.w.y|n_c], [\bar{w}.w.\bar{w}|n_d], \\ &\quad [\bar{w}.\bar{z}.\bar{y}|n_b], [\bar{w}.\bar{z}.z|n_d]] \\ &\quad \mid n_a \in N_A, n_b \in N_B, \\ &\quad n_c \in N_C, n_d \in N_D\} \\ &\quad \vdots \end{aligned}$$

このようなポート列により、大域的な名前を用いずに開放系の名前付けが可

能となる。このようなドメイン名の名前付けを DPM と呼ぶことにする。例えば、図 5.2 のそれぞれのドメイン名の一つは表 5.1 のようになる。

Table 5.1: ドメイン名

	<i>from A</i>	<i>from B</i>	<i>from C</i>	<i>from D</i>
<i>to A</i>	[]	[$\bar{x}$]	[$\bar{y}.\bar{x}$]	[$w.\bar{x}$]
<i>to B</i>	[x]	[]	[$\bar{y}$]	[w]
<i>to C</i>	[$x.y$]	[y]	[]	[$\bar{z}$]
<i>to D</i>	[$x.\bar{w}$]	[$\bar{w}$]	[z]	[]

5.2 名前付けにおける協調問題

DPM による名前付けでは、複数の異なった名前が一つのドメインを指すことがある。そのために 2 つの名前の指しているドメインの同一性を判定する方法が必要である。また、あるドメインを指す名前に対し、それよりも短くて効率のよい名前が存在する可能性があり、その変換の方法が必要である。開放系では、このようなアルゴリズムをドメイン全体に対して大域的に動作させることはできず、そのため個々のドメイン間の協調計算として定義する。

5.2.1 ドメイン名の変換

ある起点から付けられたドメイン名を別の起点から見るためには、ドメイン名の変換を行なう必要がある。図 5.3 に示すように、 d を起点としたあるドメインへの名前 n_d は、 d' を起点とした場合 $n_{d'}$ となる。そこで、 d を起点とした名前 n_t により d' を指定し、 d 、 n_d 、 n_t により $n_{d'}$ を得るような関数 tr^* を定義する。

まず補助関数として、あるドメイン名を、隣接しているドメインを起点とする名前に変換する関数 tr を定義する。

$$tr : \mathcal{D} \times \mathcal{X} \times \mathcal{N} \rightarrow \mathcal{N}$$

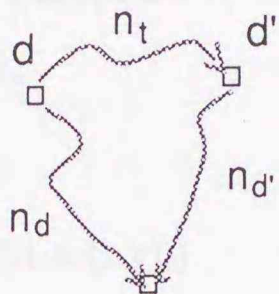


Figure 5.3: ドメイン名の変換

$$tr(d, x, [x|n]) = n. \quad (a)$$

$$tr(d, x, n) = [iname(d, x)|n]. \quad (b)$$

(a) は、あるドメイン名 $[x|n]$ が変換しようとする方向 x と同方向であることから、その先頭をとった名前を返し、(b) は、それ以外の場合に x の逆のポート名を先頭につないだ名前を返す。その結果 tr はドメイン $\langle d, x \rangle$ を起点とする名前に変換している。図 5.2 で、 D を起点とした C の名前 $[\bar{z}]$ を、ポート名 w を通じて B を起点とした名前に変換させた場合、

$$\begin{aligned} tr(D, w, [\bar{z}]) \\ &= [iname(D, w).\bar{z}] \\ &= [\bar{w}.\bar{z}]. \end{aligned}$$

となる。

次に、 tr を再帰的に用い、より離れたドメインを起点とする名前に変換する関数 tr^* を定義する。

$$\begin{aligned} tr^* : \mathcal{D} \times \mathcal{N} \times \mathcal{N} &\rightarrow \mathcal{N} \\ tr^*(d, [], n_2) &= n_2. \\ tr^*(d, n_1, n_2) &= \\ &tr^*(\langle d, hd(n_1) \rangle, tl(n_1), tr(d, hd(n_1), n_2)). \end{aligned}$$

例として、図5.2において A を起点とした C への名前 $[x.y]$ を D つまり $[x.\overline{w}]$ を起点とした名前に変換する。

$$\begin{aligned}
 & tr^*(A, [x, \overline{w}], [x.y]) \\
 &= tr^*(\langle A, x \rangle, [\overline{w}], tr(A, x, [x.y])) \\
 &= tr^*(B, [\overline{w}], [y]) \\
 &= tr^*(\langle B, \overline{w} \rangle, [], tr(B, \overline{w}, [y])) \\
 &= tr^*(D, [], [w.y]) \\
 &= [w.y]
 \end{aligned}$$

ここで、 tr^* を協調計算として解釈する。そのために、ドメインを実際分散されたプロセスに、ポートを通信路に対応させて考える。すると、 $tr(d, x, n)$ の処理はドメイン d における固有の情報を用いているので、 d 上でしか実行できない。つまり、 $tr^*(d, n_1, n_2)$ はドメイン d で処理し、その中で再帰的に呼ばれる次の tr^* は隣接したドメイン $\langle d, hd(n_1) \rangle$ で処理される。これを繰り返すと、 n_1 の示す経路上のドメインで次々に処理が行なわれることになる。上の例では、 tr^* の処理はドメイン A, B, D の順に行なわれる。

このように、複数のドメインにまたがって定義されているアルゴリズムは、各ドメインで実行可能になるまで分解してから実行される。ドメイン固有の情報を用いている計算はそのドメイン内でのみ実行され、それ以外の計算は任意のドメインで行なうことができる。例えば、 $f(d_1, x) + h(d_1, y)$ のような計算は、全てドメイン d_1 で行なわれる。しかし、 $f(d_1, x) + h(d_2, y)$ の場合には、 f を d_1 で、 h を d_2 でそれぞれ行ない、さらにそれぞれの結果を共通のドメインへ持ち寄って、そこで $+$ の計算を行なうことになる。

5.2.2 名前の最適化

ドメインへの経路が幾通りもあると、そのドメインへ複数個の名前が付いてしまう。例えば、図 5.2 でドメイン B からドメイン C への名前は、2 つのドメインを往復することも経路であるから、

$$[y], [\bar{w}.z], [y.\bar{y}.y], [\bar{w}.w.y], [\bar{x}.x.y], \dots$$

のように、無限に存在する。しかし、同じドメインを指しているこれらの名前には単純な規則があり、ルールによって相互に変換が可能である。いま、ルールの全集合を $\mathcal{R}$ とし、ドメイン d におけるルールの集合 $R_d \subset \mathcal{R}$ を考える。 R_d の元は d を起点とした 2 つのドメイン名で構成され、それらのドメイン名は異なった経路により同じドメインを指していることを意味する。

$$R_d \subset \{(n_1, n_2) | n_1, n_2 \in N_d,$$

$$\langle d, n_1 \rangle = \langle d, n_2 \rangle\}$$

ここで、右辺は無限の元を持つ集合であるが、 R_d は有限である。

同じドメインへ異なった経路が存在するということは、その経路の途中でループが存在するということである。また、隣接したドメイン間を往復することもループに含めて考えることができる。そこで、ルールはその系のループを基にして作られる。例えば、図 5.2 に示すドメイン B におけるルールは次のようなものを考えることができる。

$$R_B = \{([y.z.w], []), ([\bar{w}.z.\bar{y}], []),$$

$$([y.z], [\bar{w}]), ([\bar{w}.z], [y]),$$

$$([y.\bar{y}], []), ([\bar{w}.w], []),$$

$$([\bar{x}.x], [])\}.$$

ここで、最初の 4 つのルールはドメイン B-C-D-B を通過するループによるもので、残りの 3 つのルールはポートの往復によるものである。このような

ルールが他のドメインにも存在する。

このルールを用いて、名前をより短いものへ変換する関数を定義する。ある名前に対するルールの適用は次のように定義することができる。

$$\times : \mathcal{N} \times \mathcal{R} \rightarrow \mathcal{N}$$

$$n \times (n_1, n_2) =$$

$$\text{if } n = n_1 : n_r \text{ then } n_2 : n_r$$

$$\text{else } \perp$$

これは n_1 が n の先頭からの部分列であるならば、 n_1 と一致する部分の代わりに n_2 を繋いだものを返し、それ以外は定義されていない。この動作例は以下のようなになる。

$$[x.y.z.\bar{w}] \times ([x.y], [\bar{z}]) = [\bar{z}.z.w].$$

$$[z.y.z.\bar{w}] \times ([x.y], [\bar{z}]) = \perp.$$

次に、ドメイン d を起点とする名前 n が与えられたとき、 n に適用可能なルールを全て適用する関数を定義する。

$$\text{reduce} : \mathcal{D} \times \mathcal{N} \rightarrow \mathcal{N}$$

$$\text{reduce}(d, n) =$$

$$\text{if } \exists (n_1, n_2) \in R_d \wedge n = n_1 : n_r$$

$$\text{then } \text{reduce}(d, n \times (n_1, n_2))$$

$$\text{else } n.$$

例えば、ドメイン B における $[\bar{w}.\bar{z}]$, $[y.\bar{y}.y]$, $[\bar{w}.w.y]$, $[\bar{x}.x.y]$ 等はこの演算により、すべて $[y]$ に変換される。

ルールの適用する関数 reduce が必ず停止するために、ルールは循環しないようにする必要がある。例えば、 $\{([x.y], [z]), ([z], [x.y])\}$ の場合は、2つ

のルールが循環しているためにこの適用は停止しない。そこで、必ず停止する条件として、 R_d は

$$\forall n \in N_d, r_i \in R_d$$

$$n : n_r \neq n \times r_1 \times r_2 \times \cdots \times r_k.$$

を満足する必要がある。すなわち、ある名前 n に任意のルールを一度以上適用して行き、 n の部分列と等しくなることが無いことである。

また、ルールに合流性が存在しない場合には、*reduce* が返す名前は、ルールを適用する順序に依存して一意に定まらない可能性がある。例えば、図 5.2 において、ルールの集合が $R_B = \{([y.z.w], []), ([y.z], [\bar{w}])\}$ であった場合、 $reduce(B, [y.z.w.\bar{x}])$ は $[\bar{x}], [\bar{w}.w.\bar{x}]$ のいずれかの解を返す。このとき、 R_B に $([\bar{w}.w], [])$ というルールを加えると 1 つの解に落ち着く。このような合流性を持つためのルールの集合への条件を与える。任意の名前 $n \in N_d$ に対して適用が可能な 2 つのルール列 a^*, b^*

$$R_d^* = \{r_1 \times r_2 \times \cdots \times r_k \mid r_i \in R_d\}$$

$$a^*, b^* \in R_d^*$$

が与えられたとき、

$$\exists c^* \in R_d^*$$

$$[a^* \times c^* = b^*] \vee [a^* = b^* \times c^*].$$

を満足するように R_d を決めることにより、合流性を満たすことができる。

ここで定義した *reduce* は一つのドメインで行なわれるものであった。しかし、この処理をさらに経路上のすべてのドメインでの協調処理とすることにより、より短い経路が得られる。その準備として、先に定義した tr^* を拡張して、経路中のドメインで変換ルールの適用を行なう関数 *send* を定義する。

$$send : \mathcal{D} \times \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{D} \times \mathcal{N}$$

$$send(d, [], n_2) = (d, n_2)$$

$$send(d, n_1, n_2) = \text{let } n_r = reduce(d, n_1)$$

$$send(\langle d, hd(n_r) \rangle, tl(n_r), tr(d, hd(n_r), n_2)).$$

ここで、 $send$ は名前と起点の両方を返す多値関数である。 $send$ は tr^* と同じ働きをするが、ここでは離れたドメインへ自分自身への名前 $[]$ を渡すことにより、そのドメインへの経路を短くする働きとして用いる。例えば、図 5.2 において $send$ を用いて A を起点とした $[x.y.z] \rightarrow []$ を渡してみると、

$$\begin{aligned} & send(A, [x.y.z], []) \\ &= send(\langle A, x \rangle, [y.z], tr(A, x, [])) \\ &= send(B, [y.z], [\bar{x}]) \\ &\quad ; reduce(B, [y.z]) = [\bar{w}] \text{ なので} \\ &= send(\langle B, \bar{w} \rangle, [], tr(B, \bar{w}, [\bar{x}])) \\ &= send(D, [], [w.\bar{x}]) \\ &= (D, [w.\bar{x}]). \end{aligned}$$

となる。ここで得た $[w.\bar{x}]$ は、 D を起点とした A へのより短い名前である。 $send$ を繰り返し用いて $reduce^*$ を定義する。

$$reduce^* : \mathcal{D} \times \mathcal{N} \rightarrow \mathcal{N}$$

$$reduce^*(d_1, n)$$

$$= \text{let } (d_2, n_{back}) = send(d_1, n, [])$$

$$(d_1, n_{go}) = send(d_2, n_{back}, [])$$

$$\text{if } n = n_{go} \text{ then } n$$

$$\text{else } reduce^*(d_1, n_{go}).$$

これは、 $send$ の呼び出しを交互に行ない、その名前が変化しなくなるまで繰り返している。

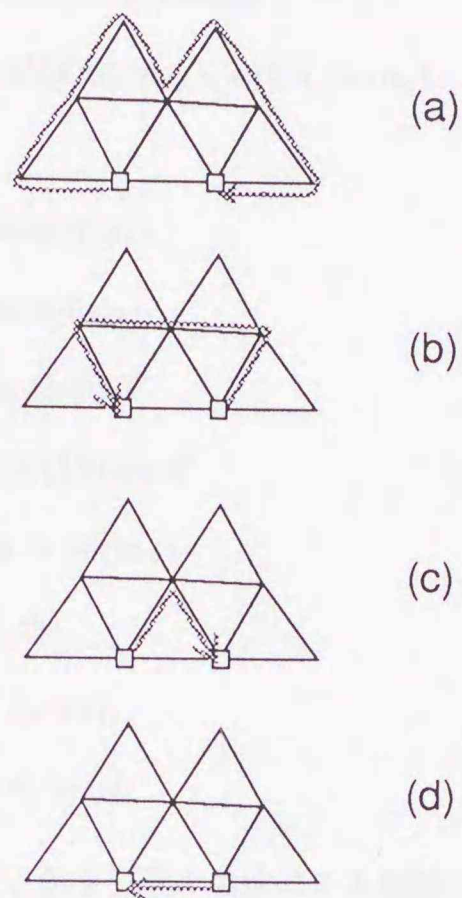


Figure 5.4: 経路の最適化

例えば図 5.4において、各ドメインには小さな三角形のループに関するルールのみがあったとする。そこで、(a) のような名前を $reduce^*$ に与えると、 $send$ が呼ばれるごとに (b)(c)(d) の順に名前が変化する。

5.2.3 名前の同一性

与えられた2つのドメイン名が同じドメインを指しているかどうかを判定する関数を定義する。もし、それらの名前の起点において、それらを同一化できるようなルールがすでに存在するならば、この関数は一つのドメイン内で動作する。しかし、各ルールが不完全である場合には2つの名前が通過している各ドメインの協調計算により求める。以下にそのような関数を定義する。

$eq, eq1 : \mathcal{D} \times \mathcal{N} \times \mathcal{N} \rightarrow \text{boolean}$

$eq(d, n_1, n_2) = eq1(d, n_1, n_2) \vee eq1(d, n_2, n_1).$

$eq1(d, n_1, n_2) =$

 let $n_{r1} = \text{reduce}(d, n_1),$

$n_{r2} = \text{reduce}(d, n_2),$

 if $n_{r1} = n_{r2}$ then T

 else if $n_{r1} = []$ then F

 else let $x = \text{hd}(n_{r1})$

$eq1(\langle d, x \rangle,$

$\text{tr}(d, x, n_{r1}),$

$\text{tr}(d, x, n_{r2})).$

$eq1$ は eq の補助関数で、第 2 引数の名前による経路のドメインに対して、2 つの名前が等しいかどうか判定できるまで、順に聞いている。 eq は $eq1$ の引数を入れ替えて、すなわち 2 つの名前上のドメインすべてにその判定を要求している。例として、図 5.2 においてドメイン A からドメイン C への 2 つの名前 $[x.y], [x.\overline{w}.z]$ が同じものを指しているかどうかを調べてみる。

$eq1(A, [x.y], [x.\overline{w}.z])$

$= eq1(\langle A, x \rangle, \text{tr}(A, x, [x.y]),$

$\text{tr}(A, x, [x.\overline{w}.z]))$

$= eq1(B, [y], [\overline{w}.z])$

 ; $[y] = \text{reduce}(B, [\overline{w}.z])$ なので

$= \text{T}$

eq の協調アルゴリズムとしての特徴は、各ドメインにおけるルールを協調的に適用して全体が動作していることである。つまり、ドメイン A にはドメイン B-C-D-B のループに関するルールは存在していないので 2 つの名前の

同一性を判定できないが、ルールを持っているドメイン B に処理を移譲することにより、同一性が判定できる。

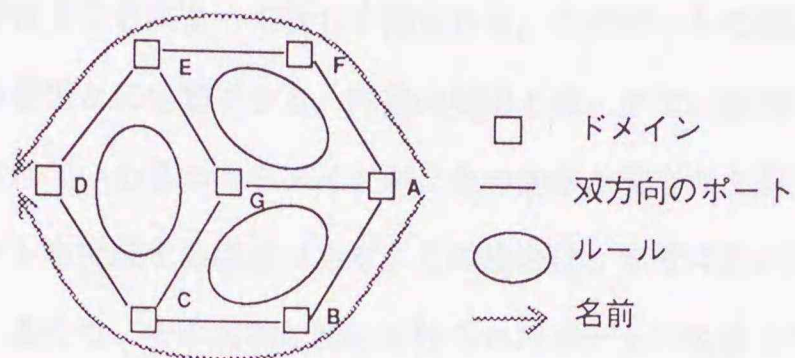


Figure 5.5: 不完全なルール

各ドメインが持っているルールが不十分である場合には、この eq が正しく動作しない。図 5.5 で、3 つのループに関するルールがそのループ上のドメインにのみ存在しているとしよう。つまり、ドメイン A は 2 つのループを知っており、ドメイン G は 3 つのループを知っている。ここでのルールが不完全であるのは、一番外側のループ A-B-C-D-E-F-A に関するルールが存在しないことである。この場合、2 つの A から D への名前 (経路が A-B-C-D と A-F-E-D によるもの) が同一であることは、 eq から判定できない。しかし、G にある 3 つのループを合成すると、外側のループを知ることができるので、これらの名前を G に移動すると判定が可能である。 eq は 2 つの名前の経路上のドメインにのみ判定を依頼するが、この例のようにそれらの経路上にないドメインへ依頼することにより、ルールの不完全性を補うことが可能となる。ルールを多く持つか、判定アルゴリズムを複雑にするかは記憶域と処理速度のトレードオフの問題である。

5.2.4 ルールの形成

ルールはそのドメインに含まれているループと対応し、そのループはポートが接続されることによって新しく作られる。このポートの接続は内部の要因と外部の要因とに分類できる。内部の要因とは、すでに別のポート列を通じて接続されている2つのドメインが、その効率や安全性を向上させる目的などでポートを接続する場合である。この場合は、すでに互いのドメイン名は知っているの、その名前と新しく作られたポートの名前とでルールを作ることができる。外部の要因としては、例えばシステムを操作している人間がポートを接続した場合である。この場合でも、各ドメインがすでに互いのドメイン名を知っているのであれば、それと新しいポート名とを人間が結び付ければ、ルールは生成できる。それ以外の場合は、2つのドメインが初めて接続されたとみなされるので、ループは生じない。実際には、極めて離れたところで2つのドメインが接続されている可能性もあるが、それを知るためには大きなコストがかかる。開放系では完全に接続されていないことを知るためには無限のコストが必要であり、それらすべてを知らなくてもよいとする立場をとるのが開放系の考え方である。

より積極的にループを検出する方法も考えられる。例えば、あるドメインからブロードキャストメッセージを全てのポートを通じたドメインに送り、それを受け取ったドメインは再びすべてのポートを通じたドメインへブロードキャストメッセージを送る。もしループがあれば、いずれ発信者へメッセージが届けられるはずである。この方法が停止する為には、メッセージに距離の情報も含め、ポートを通過する度にそれを減らすようにすれば、ある距離より小さいループを得ることができる。

5.2.5 論理的なポート

ユーザ向けの名前付けを考えると、実際に存在するポートとは別の切り口から名前付けを行ないたい場合がある。そのために、物理的には存在しない論理的なポート（別名ポート）を考えることができる。別名ポートにより生じたルールは、別名ポートを物理的なポートに置き換える方向に適用される。例えば、木構造の名前はユーザにとって利用しやすく、多くのシステムで用いられている。DPM で名前付けされたシステムを、別名ポートを適当に接続することにより木構造の見方が可能である。最初にユーザが用いる名前は別名ポートを介したものであるが、各ドメインでのルールにより物理的なポートを通じた名前に変換されて、使用される。これは、高レベルの抽象的な名前がより低レベルの具体的なものへと順次変換されるモデル [5] と同様である。

これとは別に、系全体のルールの軽減や、よく使う名前を実際より短いものにする目的でも、物理的には存在しないポート（急行ポート）を考えることができる。急行ポートが別名ポートと異なっているのはルールの適用の方向である。この場合、ルールは積極的にこの急行ポートを用いるように適用される。急行ポートを通じてのデータの転送は、それと等価な物理ポートの列を通じて行なわれるが、途中のドメインではルールの適用などの処理は一切行なわれず、通過するのみである。

長さが N であるループが存在する場合、ループ上のドメインでのルールの記憶域や変換速度は $O(N)$ のコストがかかる。これが、 N 個のドメインに存在するので、 $O(N^2)$ のコストになり、大きなループでは問題となる場合がある。そこで、図 5.6 のように急行ポートを用いて小さなループに分解すると、急行ポートが接続されていないドメインでは N が小さなループとしてルールを作ることができ、コストの軽減に寄与する。さらに、遠く離れたドメインへ

の長い名前も、急行ポートを通すことによって、より短いものへ変換できる。

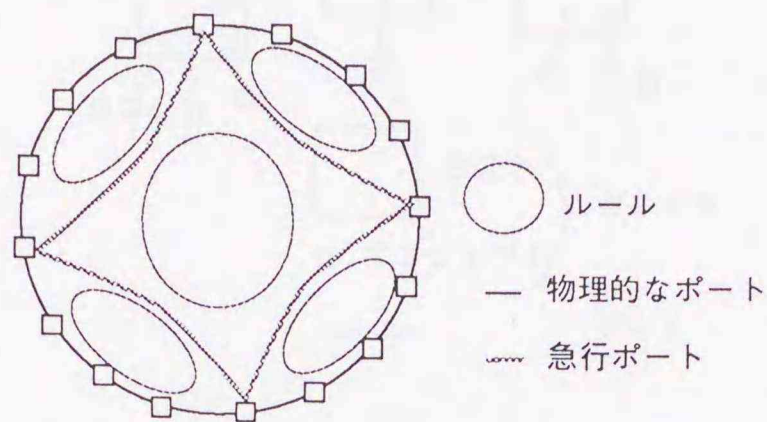


Figure 5.6: 急行ポート

5.3 DPM による名前付けの例と比較

DPM を実際のシステムの名前管理として用いた場合について考察し、他の方式と比較する。

5.3.1 オブジェクト ID

多くのシステムが分散環境をオブジェクト指向パラダイムによって捉え、分散された資源をオブジェクトとして統一的に扱っている [29]。オブジェクト ID はそのオブジェクトを参照する為に用いられ、環境全体でユニークになるように付けられている。経路探索の効率を重視する場合には、オブジェクトが存在する位置の情報がオブジェクト ID の構造の中に直接埋め込まれる。オブジェクトマイグレーションの効率も重視する場合には [15]、中間層を設けることもある。MUSE[19] では、木構造の名前を相対的に付けた、不定長のオブジェクト ID を採用している。名前の相対性により木構造の縦の方向に開放的な性質が含まれる。ここでは、DPM により ID が付けられたオブジェクトへのメッセージ送付と、オブジェクトマイグレーションについて述べる。

図 5.7 は、DPM によるオブジェクト ID の付け方を示している。オブジェ

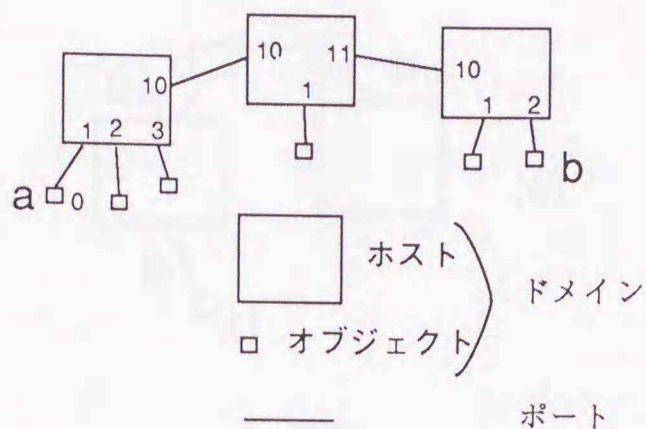


Figure 5.7: オブジェクト ID

クトとそれが存在するホストとがドメインに対応し、それらをポートが接続している。この図で、オブジェクト a がオブジェクト b を指す場合、その ID は [0.10.11.2] となる。メッセージは受け手を示す ID の経路上のドメインを経由して伝えられる。このような、隣接した媒体を通じて離れたオブジェクトへメッセージが送られる例は [13, 16, 8] 等にも見られる。メッセージの引数として他のオブジェクト ID を渡すような場合、その ID を受け手から見たものに変換する必要がある。そこで、前に定義した *send* をメッセージ送出に用いる。すなわち、 $send(d, n_1, n_2)$ は d を起点とする n_1 に対して n_2 というオブジェクトを渡していることになる。*send* はさらにメッセージ名や複数の名前等を渡せるように拡張し、用いられる。すなわち、メッセージの引数に他のオブジェクト ID を渡す際には、*send* が移動する各ドメインでそのオブジェクト ID を常に相対的に正しくなるように変換して渡される。

オブジェクトマイグレーションは次のように行なわれる。図 5.8(a) で、ホスト D_s に存在するオブジェクト O がホスト D_d にマイグレートする。オブジェクト O と移動先のホスト D_d 間にポート p_d を接続する (図 5.8(b))。その結果、現在のホスト D_s と D_d 、 O を結ぶループができ、それに関するルールを D_s と O に作成する。次に、 O と D_s とをつなぐポート p_s を切断し (図 5.8(c))、 p_s

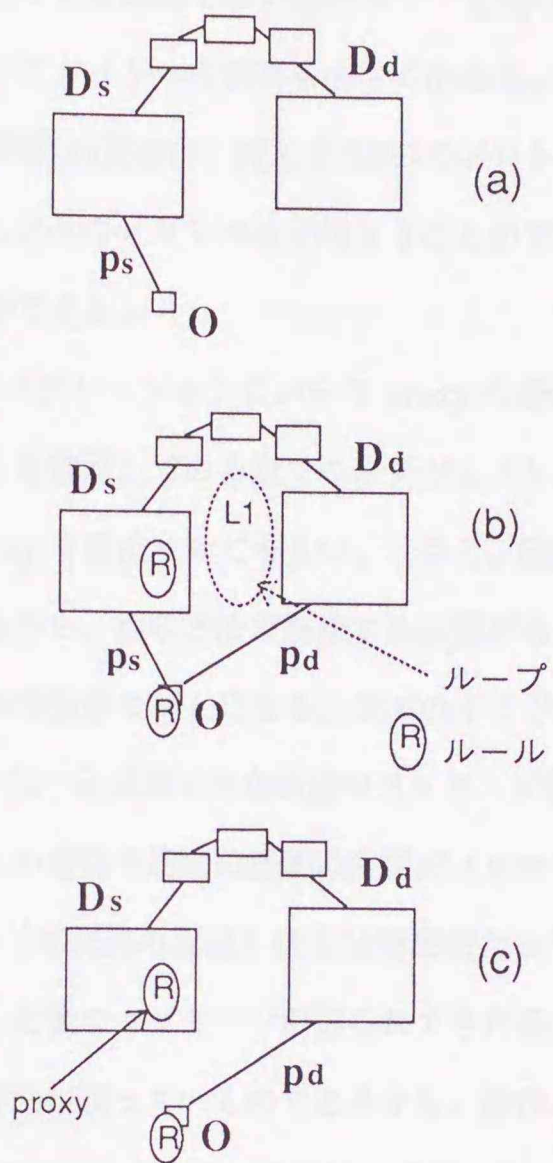


Figure 5.8: オブジェクトマイグレーション

を迂回して p_d を用いるように D_s と O におけるルールを修正する。 D_s におけるこのルールは proxy[2] の働きをし、ここに送られてきたメッセージは新しいホストへ送り直される。

DPM でのオブジェクト ID は、途中の経路が動的に変更されることから、動的に変化する。あるオブジェクトへの ID を獲得し、その ID を固定的に用いても、メッセージの送出の際に途中のドメインである程度効率のよい経路を用いるので、問題はない。さらに、メッセージのやり取りのたびに相手の ID を更新するように各オブジェクトが動作すれば、より効率のよい結果が得

られる。途中のドメインを経由しない急行ポートを用いるように ID が更新されれば、その間のドメインへの負荷も小さく出来る。相手の ID(つまりインスタンス変数の内容)の更新は、例えば RPC のプロトコルと対で用いることにより、オブジェクトのメタレベルで行なうことができ、オブジェクトレベルから隠すことができる。

オブジェクトマイグレーションにおいて proxy の寿命が問題となる。移動したオブジェクトを参照している全てのオブジェクトがこの移動を知ったときには、この proxy を消滅させてもよい。しかし、開放系ではこれを知ることが不可能であるから、別の方法で解決する必要がある。この解決には人間社会の住所変更の作法がヒントになる。全てのオブジェクトは自分が知っているオブジェクトに一定間隔で存在確認のメッセージを送る(年賀状)。その際、オブジェクトの移動や途中の経路の変更がメッセージの返答によって伝えられる。proxy(郵便局の配送)は十分時間がたった後で自発的に消滅する。もし、消滅した後でメッセージが送られてきた場合は、それを送出したオブジェクトの作法が誤っているのであるから、動作は保証されない。また、データベース(市役所)のように大量にオブジェクトを捉えている場合には、その全てにこのようなメッセージを送ることは不可能であるため、データベースに登録したオブジェクトは自分が移動した際にデータベースへその移動を伝える必要がある(住所変更届)。

一方、オブジェクトが図 5.8 と異なって、移動体通信のようにシステムから完全に切り離されて移動される場合 [15] には、移動の前後で共通な指標となる地点を設定することが必要である。システムから切り離されるときに、その指標への経路をオブジェクトが覚えておき、新たにシステムのどこかに接続された際に、そこから同じ指標への経路を得る。オブジェクトは2つの経路を合成することによって、どのような移動が行なわれたのかを知ること

ができ、それによって proxy を作成することができる。ここで問題なのは、そこで用いた指標が移動の前後で確実に同じものであることの保証である。例えば、移動に要する時間や距離が最初に見積もったものより遙かに超えた場合、同一の指標を設定できないかも知れない。これは、DPM での範囲を超えており、暗号化されたキーによる確率的な方法などの別のレベルでの保証が必要となる。

メッセージの引数が暗号化されたり、引数の型の情報が判らない場合には、引数に含まれるオブジェクト ID をうまく変換することができない。send によるオブジェクト ID の変換は、それが通過する各ドメインで少しずつ行なうので、より効率のよい名前の獲得、経路の動的な変更に強い。この性質を犠牲にして、最初にメッセージの送り手で ID を一度に変換してしまうことも可能である。これにより、途中の変換の処理を省略できるが、送り手だけの知識で変換を行なうので、そこで得た名前は全体の知識によるものよりも冗長なものになる可能性がある。

5.3.2 通信路設定問題

ネットワーク状に接続された各ノードがユニークに名前付けられており、ノードのペアを最短で結ぶ経路を探索する問題がある。DPM でこのような問題を考えると、あるノードに対しての経路はそこへの名前自身であるから、経路探索はすでに解決されており、ある与えられた経路を最短にすることになる。それを大域的にではなく、経路上のノードすなわちドメインの協調計算によって行なう。

このアルゴリズムは通信路設定問題 [18, 7] にも適用できる。例えば、このような最短経路を複数のドメインの組みが探索しているとしよう。そのとき、一つのポートを使用できる容量が決められており、その容量を超えた接続は別のポートへ迂回する。あらかじめそれぞれのドメイン間の最短経路は

静的に計算できるので、実行時には途中のポートが重なった場合の修正を局所的に行なえばよい。図 5.9 では、(a) において 2 つのポートが重なっている
ので、そこを迂回する。迂回の為に、これまでのルールの適用を逆に行なう
ことになる。次に (b) で経路の無駄な部分を最適化して、(c) を得る。複数の
迂回路がある場合、そのどちらを選択するのかを決めるためには、さらに
別のルールが必要であろう。DPM を用いると、あるポートが事故などによ
り使用不能になったときも、迂回路と同様な方法により、動的に復旧するこ
とができる。そのためには、ルールの適用順序を動的に変化させる必要があ
る。この変化は局所的に行なわれ、全体への影響は波が伝搬するように伝え
られる。

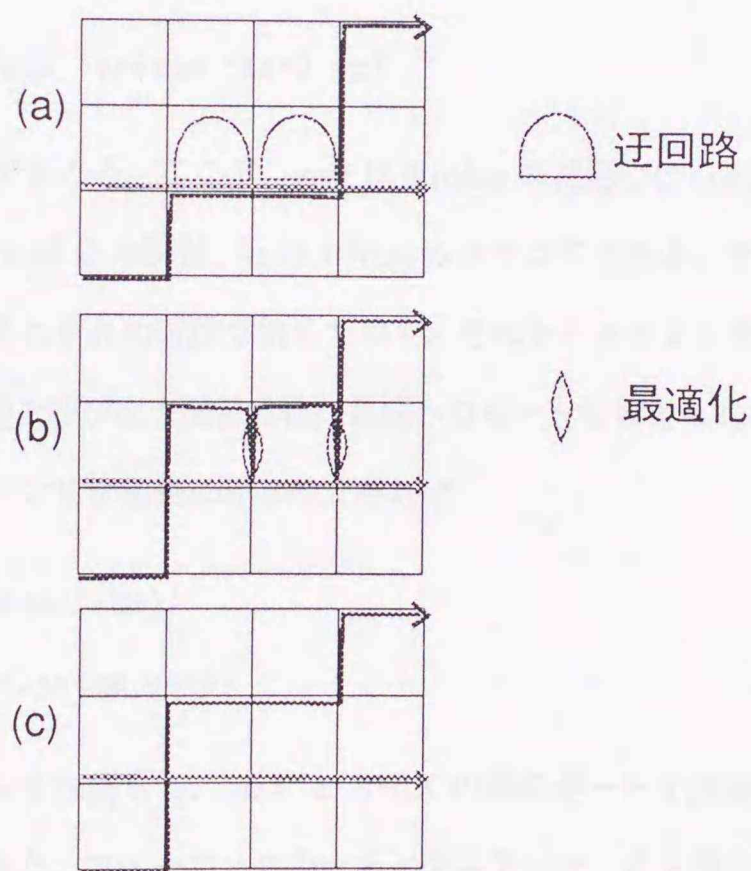


Figure 5.9: 通信路設定問題

マルチステージネゴシエーション [1] は複数のエージェントの協調によっ

て全体に与えられた制約を充足させる [18] プロトコルであり、ここでの例と似ているユニークな名前が付けられた通信回線設定問題を分散協調処理により解いている。このような違いはあるものの、このような DAI における高度なプロトコルを DPM に導入する必要があるだろう。

5.3.3 多重環境間の関数名

大規模で複雑なプログラミングでは、単一言語ではなく、複数の記述言語の組み合わせによって行なわれることが多い。それぞれの言語での名前空間は閉じており、特殊な関数によって他の言語を呼び出している。例えば、UNIX 上の C 言語で記述された LISP インタプリタの上で Prolog が動作しているとしよう。この Prolog から UNIX のコマンド ls を呼ぶ場合には、

```
(eval (C-CALL 'system "ls") *r)
```

と記述するであろう。ここで、eval は Prolog の述語、C-CALL は LISP の関数、system は C の関数、ls は UNIX のコマンドである。それぞれの関数名の空間はそれぞれの言語で閉じており、それをドメインと考えることができ、他の言語を呼び出す関数は他の言語へのポートと考えられる。LISP から UNIX のコマンドを頻繁に呼ぶのであれば、

```
(defun system (com)
  (C-CALL 'system com))
```

というルールを作成して、LISP と UNIX の間にポートを接続することが可能である。また、コールバックルーチンやエラーハンドラ等の関数を言語間で渡す場合、ポートを通した名前の変換が必要になる。このように多重言語間の関数呼び出しは DPM としてモデル化でき、DPM での考察が役に立つであろう。

さらに、異なったバージョン間のプログラミングや、プロジェクトが異なったモジュールなどのように、そのままリンクすると名前が衝突してしまう場合にも、DPM を用いてモデル化することにより、問題を解決できる可能性がある。これは今後の課題である。

5.3.4 IP アドレス

IP[17] のアドレスは、ネットワーク名とネットワーク内のホスト名に分離でき、ネットワーク名は大域的にユニークである必要がある。このユニーク性はシステムでは保証できず、社会的な組織（例えば SRI-NIC 等）により保証している。もし、ユニーク性が崩れた場合、このプロトコルは正しく動作しない。そこで、システムがこのユニーク性を保証するような管理を行なうか、DPM のように大域的なユニーク性を必要としない方法をとる必要がある。

IP でのアドレスから経路を得るアルゴリズムを簡単に述べる。ある与えられたネットワーク n 上のマシンに対してデータを送りたい場合、IP は次のように動作する。

- n が現在のネットワークと同じであれば、そのネットワークの機能を用いて与えられたマシンへデータを送る。
- n が現在のネットワークに接続されているどれかのゲートウェイの方向にあることが判っているならば、そのゲートウェイにデータを送り、これを再帰的に実行する。
- それが判らなければ、 n の存在を知っているところに行き着くまで、ランダムにどれかのゲートウェイに送ってみる。それが成功したか否かは、コントロールメッセージによって伝えられ、成功したゲートウェイを覚える。

この最後の項目は全体の構造が階層化されていれば、親をデフォルトとし

てそこに送ることができる。しかし、基本的には迷路を探索する方法と同じであり、一度探索して得た経路の情報を覚えておくことにより、実用的な速度を得ている。

Table 5.2: IP アドレスと DPM との対応

IP	DPM
ネットワーク	ドメイン
ゲートウェイ	ポート

DPM をこのような大規模ネットワークに適用してみると、その対応は表 5.2 となり、それは次のような利点と欠点を持っている。

- 利点
- DPM は全体での名前のユニーク性を要求しないので、そのための管理を必要としない。
 - DPM は経路を名前として用いるために、経路の探索や蓄積等の大域的な管理を必要としない。
- 欠点
- 資源の名前がそれぞれのドメインから相対的であるために、人間による管理は難しい。
 - 名前が不定長である。
 - 同一性を保証するために必要なコストがかかる。

IP ではネットワーク名は大域的であるため、その情報を IP の外部の機構（例えばファイル、文書、口述）を用いても伝えることができる。しかし、DPM ではそれは不可能で、逆にすべての管理を DPM のなかで行なう必要がある。これは結果的に管理全体が自動的に行なわれることを意味する。

次に、不定長の構造のままどの程度の名前を表現できるか考えてみよう。ポート名はゲートウェイの名前であるから、それらを区別できる程度の短い

ビット数でよい。さらに、ポート名の付け方は各ドメインで自由に行なつてよいので、ゲートウェイが接続されている数に応じて、そこで用いるサイズを固定する。任意のビット列は、各ドメインで各自のポート名のサイズの分だけ先頭から取り出し、残りは次のドメインで解釈される。

IP アドレスは全体で 32bits であり、そのうち最長 24bits をネットワーク名に割り当てている。それと比較するために、ネットワーク内のホストの名前として 8bits、ネットワークへの経路を 19bits、経路の有効ビットを示す size を 5bits として、アドレスを全体で 32bits になるように表現してみる。

```

8 bit      5bit      19 bit
|.....|.....|.....|
ホスト名  size  ポート名のビット列

```

例えばあるネットワーク名が [5.12.1.0.6] で、その経路上の各ドメインのポートのサイズがそれぞれ (3, 4, 1, 2, 3) であったとき、その表現は、

```

00000000  01101      000000 110 00 1 1100 101
host      size = 13   ----  6   0 1  12   5

```

となる。size が 0 の場合は、ドメイン名が [] であるので、このネットワーク上の局所的な名前である。この方法で、経路上のすべてのネットワークでのゲートウェイの数が 1-2 の時は各ドメインで 1 ビットずつポート名に使われるので 19 ホップの名前を、3-4 の時は 2 ビットずつ使われるので 9 ホップを表現できる。RIP[17] おけるルーティングテーブルは 16 ホップを無限として扱っているので、DPM の方法は同程度の規模のネットワークに対して妥当な数字である。

名前を移動している最中にビット長があふれた場合には、その名前に対して *reduce** を適用して、名前を短くすることを試みる。それでもあふれる場

合は、経路の途中に急行ポートを動的に接続し、そのポートを利用するように名前を変換して短くする。もし、このような急行ポートが全てのドメインとの間で接続された場合には、全てのドメインが急行ポートの名前で名前付けされたことになり、そこへは実際のどのポート（ゲートウェイ）へ送れば良いかがルールにより知ることができ、IPが行なっている経路の管理と似た結果になる。つまり、急行ポートを適当に用いることは、完全に相対的な名前付けと大域的な名前付けとの中間に位置すると考えられる。急行ポートの動的な接続や切断を行なうことにより、どの程度まで大規模な範囲を効率よく名前付けができるのかは、今後の課題である。

DPMが行なう名前の変換に要する計算量は、名前の長さに比例したオーダーである[4]。変換処理においては、ルールをそのままの形ではなく、自分のドメインの近傍の構造をサブグラフとして保存しておき、与えられた名前を用いてそのグラフ上を移動して、冗長な部分を削除していくものである。実際には、ルールが適用されるのは名前の最初の部分であり、さらに、名前は通常短くなって行くので、それほど多いものではない。

IPのルーティングはネットワークが大規模になるにつれ、高度なプロトコルが用いられるようになってきた。EGP[11]は隣接したゲートウェイの情報を基にルーティングを行なうプロトコルであり、その立場はDPMと極めて似ている。そのため、EGPで用いられているアルゴリズムがどの程度、DPMにも適用できるのかは今後の課題とする。

5.3.5 uucp 表記との比較

uucpを用いて公衆回線などで接続されたネットワークでは、直接接続されていないホストへの名前付けを、例えばhost1!host2!host3のように、途中のホスト名を繋ぐことにより行なっている。これは、DPMと同じ方法である。しかし、人間による相対的なアドレスの扱いが難しいために、現在で

はほとんど用いられてはいない。uucp の方法と DPM とが異なっているのは次の点である。

1. DPM は各ドメインがアドレスの変換規則を持っている。
2. DPM ではメッセージの引数に使われた他の名前も転送によって変換する。
3. 変換ルールの適用を動的に決定する。

1. により、与えられた経路より効率のよい経路が用いられるかも知れない。2. は、例えば他人のメールアドレスを教える場合、それを受け手から見たアドレスに自動的に変換する機能である。このために、メールのヘッダ部分にシステムで変換して欲しいアドレスを付加し、そこを、メールの本文で参照する。3. は、途中のホストがダウンした場合などの迂回路の設定に利用できる。

uucp 方式の欠点であったユーザにとって扱いが困難であった問題は、別名ポートによりユーザの自由な構造を作ることができるので大きな問題ではない。

Chapter 6

分散プログラミング環境

前章までに分散ソフトウェアの問題点とその解決のモデルを提案してきた。この章では、その具体的な実現として、実際に分散ソフトウェアの構築を支援するプログラミング環境を述べる。

6.1 Kamui 環境

非均質な分散ソフトウェアの構築を支援するために我々は Kamui 環境というプログラミング環境を作成している。Kamui 環境が単なる分散プログラミング記述言語ではないのは、複数の記述言語にまたがったソフトウェアの構築を想定しているからである。

Kamui 環境の構築における目標は、研究の最初の段階から存在していたわけではなく、何度かの段階をへて現在に至っている。最初の節で現在の Kamui 環境の研究の経緯を述べる。それにより Kamui 環境の研究の目標と動機が明らかになるであろう。さらに、2 節以降ではより具体的にそれらの内容について前章までで述べた分散ソフトウェアでの問題点がどのように Kamui 環境の中で実現されているのかを見て行く。

6.1.1 経緯

最初に並行に動作するオブジェクトの協調動作をうまくモデル化する並列計算モデル Kamui88 が渡辺ら [42] によって提案された。このモデルはオブジェクトのグループによる仕事を場という概念でくくったものであり、あるオブジェクトが何らかのメッセージが必要であれば、それに関連した場にオブジェクト自ら入ることにより、そのメッセージを受けることができる。場を通じて送られるメッセージはそこに属しているすべてのオブジェクトに伝えられる。このモデルはメッセージの受理の権利をそれを受信するオブジェクトに与えた点が良い。渡辺はモデルの提案と共に Pilika88 と呼ばれる処理系も実現している。この処理系は 1 つの lisp 内で疑似並列で動作するものであり、モデルの記述力の確認の為に作成された。

次にこの Kamui88 モデルを実際の分散環境で動作させる試みがなされた。それが Kamui-C Ver.1 (KC1)[26] である。この KC1 は C 言語へのプリプロセッサとして実現された。その時点で処理系自身の実現やクラス階層の扱い、さらに他の記述言語で書かれたオブジェクトとの通信などが問題となり、それらの解決が主な研究テーマとなってきた。KC1 では、クラス階層によりオブジェクトの型付けるのではなく、メッセージの集合をプロトコルとして定義し、そのプロトコルの集合によってオブジェクトの型とした。その方法により、他の記述言語とも型が付けられながら通信が可能となり、Kamui-Lisp の試作も同時に行なわれた。

その後、プリプロセッサと C 言語自身の記述能力の限界からターゲット言語を C++ へ変更した (KC2)。これによって Kamui-C 自身の構文が整理され、さらに、実行時の誤り検出などが可能となった。この拡張では特にモデル上の変更は行なわれていない。

実際に KC2 による分散ソフトウェアを構築する段階で次のような新しい

問題が生じてきた。一つはオブジェクトの部品としての側面を強調するために、オブジェクトの送出するメッセージの型についての情報が不可欠であること [28]、これは武村らの Kamui-Toolkit[41] の作成で必要となったことである。もう一つは、KC2 の処理系内部の問題であるが、動的に増加するオブジェクトに対してそれらの名前をどのように管理するか [27] である。この解決のためにドメインポートモデルが提案された。現在の Kamui-C のバージョン (KC3)[43] ではオブジェクトの送出メッセージによる型宣言の機能を持っている。またその実現として各オブジェクトにつけられた ID は DPM による開放的な管理を行なっている。これらの機能によって、ソースプログラム上の大規模性だけでなく、分散的に見た大規模なプログラミングが可能となった。

Kamui-Lisp は KC2 に対応するものが濱田らによって作成された [25] が、KC3 に対応する開放的な名前管理と送出メッセージの型情報を持つものはまだ作成されていない。

また、オブジェクトの送出メッセージによる型宣言により、複数のオブジェクトの接続を考えることが可能となった。オブジェクトの接続と同時にそれぞれのオブジェクトへの初期化メッセージも必要となる。プログラミングの方法論として、オブジェクトを部品として作成するフェーズと、オブジェクトを接続して使用するフェーズに分離することが考えられる。前者はオブジェクトの部品としての性質を公開する機能が必要であり、これをオブジェクト記述言語と呼ぶ。このオブジェクト記述言語はそのオブジェクトの性質により様々な言語を考えることができ、その役割をするのが Kamui-C と Kamui-Lisp である。それに対して、後者のオブジェクトの接続とその初期化メッセージのフェーズの為にオブジェクト操作言語というものが必要と考える。これは UNIX のコマンドとシェルインタプリタの関係に似ている。シェルインタプ

リタはコマンドをパイプライン構造に接続し起動させる機能を持っている。UNIX のコマンドの場合には型は存在しなかったが、我々のオブジェクトには入出力の型が付けられているので、このオブジェクト操作言語では型検査を行なうことができる。松岡はオブジェクト操作言語として KamuiScript を開発した。

6.1.2 型システム

Kamui 環境では分散された非均質なオブジェクトを用いたプログラミングを支援する。そこで、そのようなプログラミングでの誤りをできるだけ少なくできるように、非均質なオブジェクトに対して型をつける。それは 4 章で述べたように、メッセージの集合をプロトコルとし、その入出力の集合によってオブジェクトに型が付けられる。

このような型検査はオブジェクトの合成時に行なわれる。

6.1.3 プログラム言語

Kamui 環境でのプログラミングは次のようなものを想定している。

1. プロトコルの設計
2. オブジェクトの作成
3. オブジェクトの合成

ここで、2. はオブジェクト記述言語によって行なわれ、3. はオブジェクト操作言語が担当する。このような、流れは実際のアプリケーションのプログラミングによって得られたものである。

6.1.4 オブジェクト ID の管理

大部分の分散システムにおいてオブジェクト ID は大域的にユニークに付けられている。しかし、Kamui 環境では一つの試みとして、3 章で述べた開

放系名前付けモデルによってオブジェクト ID を管理している。

6.2 記述言語

6.2.1 Kamui-C

Kamui-C[26] は、C++の上に作られた Kamui オブジェクト記述言語である。Kamui-C はプリプロセスとして Kamui 環境のオブジェクト群から情報を得て、コンパイル時点での環境で動作できるような C++のプログラムを生成する。その後 C++コンパイラによりコンパイルされ、Kamui-C の実行時ライブラリとリンクされて実行可能な状態となる。

Kamui-C のプログラム例を以下に示す。

KamuiC

```
defprotocol pCounter {  
  send set(int aValue);  
  send up();  
  send down();  
  int get();  
}  
  
defclass cCounter [ pCounter ] {  
  int value;  
}  
  
defmethods cCounter pCounter {  
  send set(int aValue) { value = aValue; }  
  send up() { value++; }  
  send down() { value--; }
```



```

int get() { return value; }
}

```

最初の行の KamuiC は、このファイルが Kamui-C で記述されたものであることを示している。defprotocol ではメッセージプロトコルを宣言している。この宣言でコンパイラと Kamui 環境とのやり取りが行なわれ、環境のプロトコルデータベースとの比較、更新などが行なわれる。メッセージの型を示す send は非同期メッセージの送出の意味で、それ以外の型（ここでは int）は同期メッセージ (RPC) を意味するとともに、その返り値の型も示している。また、ここで void を、値は返さないが同期をとるメッセージの型として用いることができる。

defclass では Kamui-C のクラスを宣言している。ここでのクラス名は、Kamui-C が実行される UNIX プロセスの中で閉じている。次の [pCounter] はこのクラスが pCounter の仕様を満足するオブジェクトであることを表わしている。

defmethods では、メッセージプロトコル単位でメソッドをまとめて記述する。ここでは、クラス cCounter におけるビュー pCounter に関するメソッドを記述している。メソッドの本体には、Kamui-C の処理系はなにも手を加えておらず、C++ のプログラムを直接記述する。

```

defprotocol pTimer {
  send reset(int aTime);
  send up();
}

```

```

defprotocol pAlarm {

```



```

    send signal();
}

defclass cTimer [ pTimer , -pAlarm bell]
    cCounter {
        int time;
    }

defmethods cTimer pTimer {
    send reset(int aTime) {
        pCounter_set(0);
        time = aTime; }
    send up() {
        pCounter_up();
        if (value == time)
            bell.signal();
    }
}

```

クラス cTimer は cCounter よりインスタンス変数、メソッド、プロトコルを継承し、内部変数 time、プロトコル pTimer と出力プロトコル pAlarm を新たに追加している。pCounter_set、pCounter_up はプロトコル pCounter のメソッドを直接呼んでいる。reset(10) により初期化すると、メッセージ pTimer:up を 10 回受けた後、外部のオブジェクト bell にメッセージを送出する。

例えば、pCounter を持つオブジェクトの値を常に表示し、上下のボタン

でその値を変更できる GUI の表示ツールがあったとする。そのオブジェクトに cCounter のインスタンスを接続した際には、通常のカウンタとして動作する (図 6.1)。さらに、cTimer のインスタンスと入れ替え、プロトコル pTimer としてクロック発生オブジェクトと接続し、Bell と pAlarm によって接続することにより、目覚まし時計として使用できる (図 6.2)。

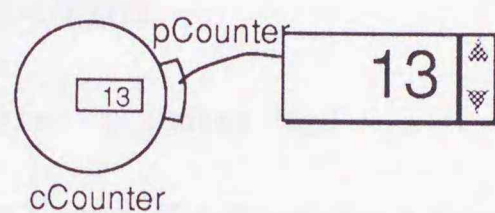


Figure 6.1: cCounter オブジェクトの動作例

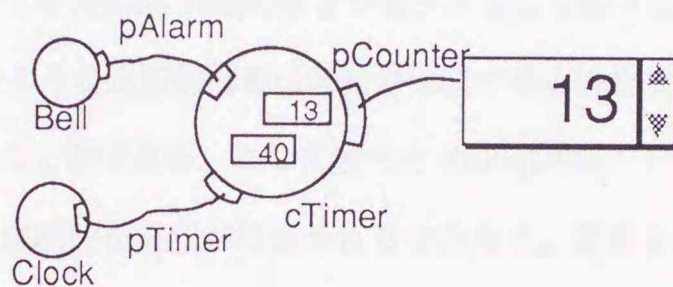


Figure 6.2: cTimer オブジェクトの動作例

6.2.2 Kamui-Lisp

Kamui-Lisp[25] は Lisp 上のオブジェクト記述言語である。現在は TAO のクラスとして実現されている。前節の Kamui-C の例と同じオブジェクトを記述してみると次のようになる。

```
(defKamuiClass cCounter (pCounter)
  (value))
```



```
(defKamuiMethods (cCounter pCounter)
```

```
  (set (v) (setq value v))
```

```
  (up () (incf value))
```

```
  (down () (decf value))
```

```
  (get () value))
```

また、メッセージを送る関数は、

```
(Kamui-Send counter 'pCounter 'up)
```

となる。さらに、プロトコル宣言やデータベースアクセスなどの関数が用意されている。

3節で述べたオブジェクトの型を満足していれば、どのようなオブジェクトの実現であっても Kamui 環境のオブジェクトとして扱うことができる。そのため、Lisp のように比較的簡単に言語が作成できる枠組では、多くの記述言語を作成することができる。ここに述べた Kamui-Lisp は一つの例であり、今後も幾つかの実験的な言語が作成されるであろう。重要なことは、そのような言語が全て型付けされて相互動作できることである。

Lisp 等の記号処理言語の特徴の一つは動的に関数を拡張できることである。すなわち、このような言語をベースとしたオブジェクトではメソッドが動的に拡張できる。こういったオブジェクトへの型付けは基本的には不可能であるが、この動的な部分を局所化することにより、安全性を向上することができる。例えば、メッセージプロトコル pRaw はメッセージ名を文字列の形で送るものであり、動的なメソッドはこのプロトコル内で作成されるものとしよう。その pRaw で送られたメッセージは Smalltalk の場合のように毎回探索され、実行時に誤り検出が行なわれる。しかし、動的な性質を持つオブジェクトであっても、通常のプロトコルで送られたメッセージに関しては従来と同じ処理が行なわれる。

6.2.3 KamuiScript

KamuiScript は Kamui 環境でのオブジェクトを操作するために提案された。ここで、操作とは以下のようなものである。

1. オブジェクトの生成
2. オブジェクトの登録、検索
3. オブジェクト間の接続
4. 起動メッセージの送出

これらは、Kamui-C の main プログラムで行なわれていたものであった。つまり、これらの操作の若干の変更の度にプログラムを再コンパイルをしていた。しかし、我々が実際に使用する段階で、このような操作は対話的に行なう方がすぐれていることが理解できた。それが、KamuiScript の作成の動機である。

オブジェクトの生成はクラスを指定することによって行なわれる。さらに、どのマシン上のどのプロセス内に作成するかの情報も付加する必要がある。各オブジェクト間で他のオブジェクトの共有が行なわれる場合、最初にそのオブジェクトに名前を付け、それによって検索がおこなわれる。つぎに、知ることができたオブジェクトを相互に接続する。接続とは、各オブジェクトのインスタンス変数への代入のことである。オブジェクトは接続が行なわれただけでは動作しないので、次にそれらに起動メッセージを送る。それによって全体の動作が始まる。

KamuiScript は操作する対象のオブジェクトの種類によって様々な用途に用いることができる。オブジェクトが GUI などのツールキット等であれば、KamuiScript はユーザインタフェース作成用言語である。GUI を用いたアプリケーションのなかにはリソースファイルによって GUI の様々な変更をユー

ザに許しているものがある。KamuiScript で記述されたファイルはこのよう
なりソースファイルとしてもつかえるのである。

6.3 環境支援

Kamui 環境でのプログラミングのために幾つかの実行支援オブジェクト
がある。

6.3.1 オブジェクトの登録

分散されて作成されたオブジェクトを共有するために、それぞれの間で共
通の名前を付けてそれによって参照する。そのために、各 Kamui-C のプロセ
ス内に gDB と呼ばれるデータベースが用意されている。これ自身も Kamui-C
によって記述されている。

参照は次の様に行なわれる。

```
obj = gDB.get("name");
```

各プロセス上の gDB は相互に参照されていると、

```
obj = pDB(gDB.get("dbName")).get("name");
```

のように、参照を連続して行なうことができる。pDB() は ID 型から pDB 型
へのキャストである。名前の中に ':' の文字が含まれている場合には、それ
を分割して上記の動作をする。つまり、

```
obj = gDB.get("dbName:name");
```

は、同様の動作をする。

Chapter 7

応用システム

この章では Kamui 環境を用いた応用プログラムについて述べる。応用プログラムはそれだけで十分研究の価値のあるものであるが、ここでは、その概要と Kamui 環境との関連について述べる。

7.1 Kamui-Toolkit

これまでの分散ソフトウェアの多くは、分散したハードウェアを主体としたものであった。しかし、ネットワークの普及は新しいタイプのコンピュータアプリケーションを生み出すこととなる。それは、CSCW と呼ばれるコンピュータを通じた複数人の仕事の支援である。Kamui 環境のような分散ソフトウェア作成支援ツールにより、このような CSCW アプリケーションの作成がより容易になることであろう。

ここで述べる Kamui-Toolkit[41] はこのような CSCW アプリケーションの作成をライブラリのレベルから直接支援しようとするものである。それは、GUI を重視したアプリケーションの作成の支援の為に、良く使われる GUI 部品をライブラリとしたツールキットを用いる場合に似ている。

最初、このような CSCW アプリケーションの場合でも従来の GUI ツールとプロセス間通信の機能によって作成可能であるように思われた。しかし、

実時間で密に結合されたユーザ間の支援のためには、不十分な事態が生じた。

1. 複数のマウス（複数のユーザによって制御されている）を扱う必要がある。
2. 編集の結果が即座に他の画面へ反映される必要がある。

このような問題の解決の為に、従来とは異なった方法によるツールキットの作成が望まれるのである。

7.1.1 Kamui 環境との関係

Kamui-Toolkit を Kamui 環境上に作成するにあたって、オブジェクト群の関係とそれらの接続を抽象化する必要がでてきた。ここでは、すでにグループとなっているオブジェクト群に対して、新たに別のオブジェクトがそのグループに参加する場合、グループからオブジェクトが抜ける場合などの処理が行なわれる。それは、Window に新しい部品を貼ったりはがしたりする場合や、ユーザが CSCW 活動に参加したり、そこから抜けたりする場合などである。このような動機から、オブジェクトの部品化の機能を向上させる必要が出てきた。さらに、オブジェクトグループに対する考え方が必要となった。

また、マウスなどが発生するメッセージは特定の系列が意味を持っている。つまり、このようなプログラムは実時間で動作するものであり、オブジェクトの同期やメッセージの系列を表現することが重要である。その結果、系列記述のモデルが必要となった。

7.2 HyperDESK

Kamui 環境は単純な分散システムだけではなく、開放分散システムに対してもプログラミング可能である。その理由はオブジェクト ID などの管理を集中的に行なわずに済む、開放的な名前付けを採用しているからである。

CSCW アプリケーションが開放分散へと対応した場合、それはどのようなアプリケーションになるのでしょうか。

HyperDESK はユーザの DESKTOP や各情報を 1 枚のカードに対応させ、それらのリンクの網によって情報の伝達や CSCW の作業を行なうものである。概念的には T.Nelson が進めている Zanadu プロジェクトに近いものである。

7.2.1 ハイパーテキスト

HyperDESK の構成要素はカードとリンクである。各カードから幾つかのカードへ方向のついたリンクが張られている。

ハイパーテキスト構造とは、文書中に他の文書へのリンクを張ることによって文書間のつながりをより高度に行なおうとするものである。文書中の他の文書の引用はその文書へのリンクとして表現され、それを辿ることによって実際の文書へアクセス可能となる。また、注釈のような文書への挿入も注釈の本体へのリンクを張ることによって実現する。さらに、目次や索引等からも実際の文書の特定の箇所へリンクを張ることによって、それらへ直接（ページを参照するよりも）アクセスができる。

Zanadu プロジェクトはこの考えを世界的規模で推し進めて、すべての文書をこの Zanadu データベース上で表現し、その引用関係を構成しようとするものである。この大きな利点は著作物の不当な複製を防止し、読者が引用を辿ることによって自動的に課金されるシステムが作れる点である。しかし、このシステムの構成があまりにも大規模であることから、現実に稼働するまでにはまだ多くの歳月が必要とみている意見もある。

HyperDESK はこのようなハイパーテキスト構造を CSCW のプラットフォームとして用いようとするものである。各情報はカードで表現され、それぞれのカードがリンクによって接続されている。あるカードを見ている人

が2人以上存在すれば、そのカード上で実時間の協調動作が生じる。この協調動作をどのように支援するかは HyperDESK とは別の次元の問題である。ここは例えば Kamui-Toolkit のようなもので組まれたアプリケーションの役目である。HyperDESK は一人の人が複数の仕事をしたり、その仕事を渡り歩いたりする点を、カードを辿ることによって実現しようというものである。

7.2.2 実現イメージ

例えば、電話番号の管理を HyperDESK で行なう場合を考えてみる。ある人物（例えば大学のサークルの後輩）に電話をかけたい、としよう。その場合、最初に HyperDESK では自分のホームカードにある大学というボタンを押す。それによって自分の大学用のカードへ進む。そこには、自分の属している講座、学科、学年、サークル、事務、他の学科などの項目がある。このカードも自分固有のものであるから、これらの項目から張られているリンクは他人のリンクとは異なっている。次に、サークルの項目を選ぶと、そのサークルのカードになる。このサークルのカードはサークルに属している人全体と同じカードを共有している。その時点で何人かのサークルの仲間がそのカードを見に着ており、そこで直接交信することが可能である。また、サークル全体への連絡の為に掲示板が使われるかもしれない。何か連絡がある場合には、そこへ直接書き込むことでよい。目的の後輩に電話をかけるため、サークルの住所録のカードへ進む。そこにはその後輩の項目があるので、そこから、後輩のホームカードへ進む。もし、その後輩がどこかでそのホームカードを見ていた場合、自動的に電話がかけられる。もし、その後輩がいない場合には、その留守番電話にメッセージを残すことができる。このとき、ある時点からシステムが勝手に電話をかけたのであるが、その仕掛けは隠されている。すなわちカードをめくることが電話番号を回したことに相当するのである。

7.2.3 メディアの統合

前節の話のなかで、電話の他に電子掲示板やチャットなどの通信機能がシステムの内部に隠されていることに注意して欲しい。つまり、様々なコミュニケーション手段が HyperDESK の制御の下で統合されるのである。

これまで異なっていたメディアであったものが区別なく利用できる。例えば、これまでは、コンピュータシステムである人が login していればその人と直接 talk コマンドを用いて交信し、そうでない場合には E-mail を用いるというように、似たような動作をするものでも異なったコマンドを用いる必要があった。HyperDESK ではこの選択はシステムが勝手にやってくれる。

7.2.4 分散データベース

HyperDESK は分散データベースとしての側面を持っている。我々はそこに大規模なシステムに対応できるように一つの制限を設ける。

これまでの分散データベース等の研究ではデータベースへの操作は読みだしと書き込みの2種類であった。トランザクションを定義し、そのなかでのデータの読みだしと書き込みとの無矛盾を保証するために、トランザクションのスケジューリングを行なう必要があるが、それは大規模な分散システムでは困難である。

実働している極めて大規模な分散データベースの例としてネットワークニュースがあげられる。これは隣のサイトとのやり取りを定義することだけで、全体が動作している。しかし、すべてのサイトで同じデータのビューがあるわけではない。これは良く経験することであるが、実際に投稿した記事の順序が保存されずに伝えられるのである。ニュースシステムが許している基本操作は、読みだし、情報の追加、情報の削除の3つである。書き込みが追加と削除の2つに分かれている。もし、情報の追加の順序があまり重用ではない場合には、ニュースシステムのように追加に関して排他制御を行なわ

なくても良い。また、情報の削除はニュースシステムではその記事を投稿した人のみその権限が許されているので、これもトランザクションの排他は必要がない。

HyperDESK ではこのように情報の追加に関して半順序が保存されるだけで十分であれば、それを排他制御無しで実行する。また、情報の削除は常に一人だけが行なう。例えば A,B,C,D の四人で議論していたとする。最初に A の提案 a に対して B と C の二人が意見 b,c を述べたとする。その場合、各人の意見が見える順序は図 7.1 のように異なっている。

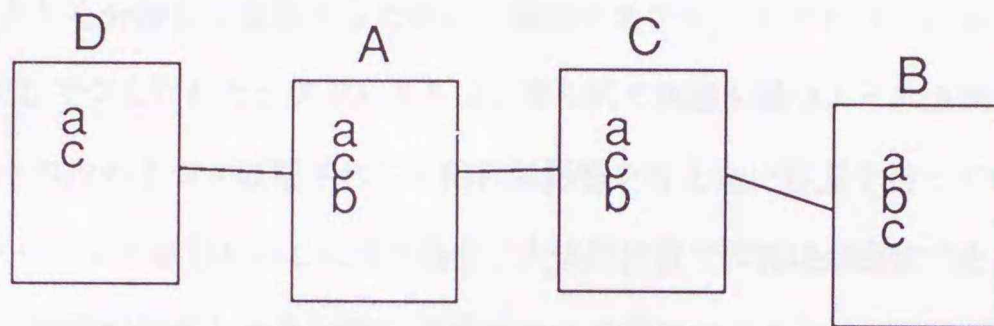


Figure 7.1: 各人のビューの違い

ここで D にはまだ b が届いていない状態である。A が c,b に対しさらに意見 a' を述べたとき、D には c よりも先に a' が届いてはいけない。現在のニュースシステムでは各サイト間で順序を保存しない方式で記事を交換するためこの順序の入れ替えが生じることがある。HyperDESK ではこの半順序を保証するために、サイト間の順序を保存する。つまり、もし A: で $c \leftarrow a'$ ならば、a' を新たに伝えようとするところ (例えば D) でも $c \leftarrow a'$ でなければならない。つまり、そこに c がなければ、a' を送る前に c を送る必要がある。これによって全体の半順序が保存される。

7.2.5 新しい構築モデルの必要性

このアプリケーションの作成は比較的最近始められたので、Kamui 環境への影響はこれから出てくるところである。現在考えられている問題は、分散されたカードをどのようにして効率よくアクセスするかである。我々はそれをオブジェクトのクローンとして考える。離れたところのオブジェクトにアクセス（内容の読みだし）する場合に、そのクローンを自分に近いところに作り、それらの間でクローンリンク (CL) を貼る。CL は、通常のオブジェクト間のリンク (RL) とは性質が異なっている。CL は仮想的には 1 つのオブジェクトを分散して実現するために、複数のオブジェクトをつなぐものである。CL でつながれたオブジェクトは、常に同じ状態を保つように協調し、また、その内の 1 つが破壊されても他には影響を与えない性質を持っている。

クローンの状態を同じに保つ場合、大規模分散での実現が困難である。そこで、前節で定義した各状態に半順序のみで満足するように規制を弛める。それによって、大規模分散での実現は極めて容易となる。

RL を通じて送られてきたメッセージは 3 種類に分類される。

読みだし オブジェクトの内部状態を変化させない。

追加 オブジェクトに情報の追加を行なう。クローン間で情報の追加の半順序のみが保存される。

削除 クローン間で矛盾が生じないために、全体で一人にしか削除の権利は与えられない。

RL から送られたメッセージが追加と削除の時には、そのメッセージを CL を通じて別のクローンに送り直す。CL を通じたそれらのメッセージは特別な実行モードでメソッドを起動する。それは、そのメソッド内から他のオブジェクトへメッセージを送出しないことである。つまり、それは自分の状態を更

新するのみに用いられる。

CL は事故による破壊に強くなるために、ループを形成している必要がある。大規模になるとそれらのループは小さいものが組み合わせられた形になるのが望ましい。一つのオブジェクトに送られたメッセージを CL を通じて他のクローンに再送される場合、そのメッセージが重複なく、できるだけ少ない時間で伝えられる必要がある。それらの処理が開放系の名前付けの方法と同様に局所的な処理により行なわれる必要がある。

Chapter 8

おわりに

分散ソフトウェアの構築モデルを Kamui 環境の作成を通じて提案した。
ここで扱われたのは、

1. 非均質分散ソフトウェアにおける型
2. 並行オブジェクトの部品化
3. 通信の構造化
4. 開放系の名前管理

である。これをまとめると、1. は非均質分散ソフトウェアにおいては、クラスなどによって型を付けることができず、実際に流れるメッセージの種類によって型を付けることである。さらに、大規模で多人数のプログラミングにおいてメッセージ名が衝突する可能性もあるので、モジュールプログラミングと同様の方法により、プロトコルとメッセージ名との組みでメッセージを表現する。2. は、オブジェクトの独立性が高くなってくると、それらを部品として考えるには受信可能なメッセージの種類による型だけではなく、どのようなメッセージを送出する可能性があるのかについても型として表現する必要がある。3. は、さらに単純なメッセージだけではなくメッセージが行き来する状態についても考える必要がある。4. は、システムが大規模になって

くると、それを従来の分散モデルで捉えることが不可能となり、開放系として考える必要が生じてくる。開放系として最初に考える必要があるのは、開放系における名前付けの問題である。すなわち、開放している資源にどのようにして全体で整合のとれた名前を付けることができるかである。ここでは、相対的な経路を利用してそれを名前とし、名前の移動、同一性の判定、最適化などの処理を各分散システムの協調によって行なわれる。

以上のように提案された構築モデルに対して、その記述力を試すために分散ソフトウェア構築支援環境 Kamui 環境を構築した。Kamui 環境として、Kamui-C と Kamui-lisp の2つのオブジェクト記述言語を作成し、それらが協調動作できるように支援オブジェクトを定義した。

さらに、環境としての記述力を試す目的で Kamui-Toolkit と HyperDESK のアプリケーションを試作した。これらの構築モデルが妥当なものであることが、このようなアプリケーションの記述において示された。

今後、さらに進んだ問題点を抽出し、それを解決する構築モデルを提案し、それによって新しいプログラミング言語へと発展させて行きたい。実際、現時点の Kamui-C などの記述言語はまだメッセージの順序に基づいた抽象化は行っていない。しかし、アプリケーション記述の経験から、このような抽象化がプログラミングの見通しを高めることがすでに判っている。

さらに、作成されたプログラミング環境の配布等も行ないたい。そのためにも、より強固なシステムの作成が要求される。

実際のシステムの作成と構築モデルの提案とがバランスよく行なうことによって、新しい問題の発見につなげることができた。つまり、最初から自明の問題に取り組んだわけではなく、それまでに理解されている構築モデルにしたがってプログラミングを行ない、それによって新たに問題を発見してきたのである。現在進行中であるアプリケーションの作成によって新たな問

題が見つけれられるであろうし、それによってまた新たな構築モデルが考えられることになるであろう。

今後考えなければならない問題は以下のようなものである。

1. オブジェクトの複製

2. 永続的な属性

1. は離れたオブジェクトへのアクセスを軽減させたり、予期しないオブジェクトの破壊などに強くなるため、オブジェクトの分散した複製を作ることである。これは、計算処理としては2重に行なっているため無駄であるが、その分、安全で高速なアクセスが提供される。達成されなければならない点は、複製されたオブジェクト間での内容の一致を保証することと、自分が指しているオブジェクトが破壊された際にいかにして別の複製オブジェクトを知ることができるかである。これは、HyperDESK の効率のよい実現にぜひとも必要な技術である。2. はさらに長い寿命をもつオブジェクトをどのように作成するかである。具体的には、オブジェクトがメモリ上からファイルなどの2次記憶へと移動させることである。オブジェクトベースと呼ばれる研究が関係するので、その成果をどのように分散プログラミング支援環境へ取り込むのが課題とされている。

謝辞

本研究を進めるにあたり、御指導、貴重な御助言を頂きました北海道大学工学部情報工学科宮本衛市教授、赤間清助教授、三谷和史助手、渡辺慎哉助手に深く感謝いたします。また、有益なコメントを頂いた言語情報工学講座の皆様にも深く感謝いたします。

Bibliography

- [1] Conry, S.E., Meyer, R.A., and Lesser, V.R.: Multistage Negotiation in Distributed Planning, *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, 1988, pp. 367-384.
- [2] Decouchant, D.: Design of a Distributed Object Manager for the Smalltalk-80 System. *Proc. OOPSLA'86*, 1986.
- [3] 原田康德, 浜田昇, 渡辺慎哉, 宮本衛市: 分散型環境 kamui について, 情報処理学会記号処理研究会報告, 90-SYM-57, 1990.
- [4] 原田康德, 宮本衛市: 開放系名前付けモデルを導入した開放分散型言語 Kamui-C, ソフトウェア科学会第 8 回全国大会論文集, 1991.
- [5] Hauzeur, B.M.: A Model for Naming, Addressing and Routing, *ACM Trans. Office Information Systems*, Vol.4, No.4(1986), pp. 293-311.
- [6] Hewitt, C. and Jong, P.: Open Systems, chapter 6, *On Conceptual Modelling*, Springer-Verlag, 1982, pp. 147-164.
- [7] 湊賢治, 奥村康行, 岸本了造: 分散協調マルチエージェントモデルによる通信網の複数回線設定制御, 情報処理学会研究報告, 90-SF-37 90-PL-27, 1990.

- [8] 村上健一郎, 明石修, 天海良治, 奥野博: 計算モデル linda の tao への導入—近傍系理論に基づく計算モデルのための動的計算場について, 情報処理学会記号処理研究会報告, 90-SYM-57, 1990.
- [9] 沼岡千里, 所真理雄: 分散人工知能とプログラミング言語, 人工知能学会誌, Vol.5, No.4(1990), pp. 441-421.
- [10] Numaoka, C. and Tokoro, M.: Open mind : conversational model for open systems, 日本ソフトウェア科学会オブジェクト指向計算に関するワークショップ論文集, 1990.
- [11] Rosen, E.C.: EXTERIOR GATEWAY PROTOCOL (EGP), RFC827, 1982.
- [12] 清水謙多朗, 前川守, 芦原評: 分散オペレーティング・システムにおける名前管理, コンピュータソフトウェア, Vol.6, No.3(1989), pp. 19-34.
- [13] 高田敏弘: 細粒度オブジェクトによる並列計算システムへ向けて—近傍場を介したメッセージ送信, ソフトウェア科学会第7回全国大会論文集, 1990.
- [14] 竹内郁雄: コネクティクスとはなにか, *Computer Today*, Vol.8, No.2(1991), pp. 52-62.
- [15] Teraoka, F., Yokote, Y., and Tokoro, M.: A Network Architecture Providing Host Migration Transparency, Technical Report SCSL-TM-91-009, Sony Computer Science Laboratory Inc., 1991.
- [16] Tokoro, M.: Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment, *The 2nd*

- IEEE Workshop on Future Trends in Distributed Computing Systems*, 1990.
- [17] XEROX: Internet Transport Protocols, XSI 028112, XEROX Corporation, 1981.
 - [18] 横尾真, 石田享: ATMS を用いた分散制約充足問題, 情報処理学会論文誌, Vol.31, No.1(1990), pp. 106-114.
 - [19] Yokote, Y., Mitsuzawa, A., Teraoka, F., Fujinami, N., and Tokoro, M.: *Continuous-Grained Objects in the Muse Operating System*. Technical Report SCSK-TM-91-008, Sony Computer Science Laboratory Inc., February 1991.
 - [20] America, P.: A Parallel Object-Oriented Language with Inheritance and Subtyping, *Proc. ECOOP/OOPSLA '90*, 1990.
 - [21] Atkinson, R. and Hewitt, C.: Synchronization in actor systems, *4th SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, 1977.
 - [22] Black, A., Hutchinson, N., Jul, E., Levy, E., and Carter, L.: Distribution and abstract types in emerald, *IEEE Trans. Softw. Eng.*, SE-13, No.1(1987), pp.65-76.
 - [23] Canning, P., Cook, W., Hill, W., and Olthoff, W.: Interfaces for strongly-typed object-oriented programming, *Proc. OOPSLA '89*, 1989.
 - [24] Danforth, S. and Tomlinson, S.: Type theories and object-oriented programming. *ACM Comput. Surv.*, Vol.20, No.1(1988), pp.29-72.
 - [25] 浜田昇, 原田康徳, 渡辺慎哉, 宮本衛市: Kamui 環境におけるオブジェクト記述について, 情報処理学会記号処理研究会報告, 90-SYM-57, 1990.

- [26] 原田康德, 渡辺慎哉, 三谷和史, 宮本衛市: イベントグループを導入した
並行分散オブジェクト 指向言語 kamui-c について, ソフトウェア科学会
第 6 回全国大会論文集, 1989.
- [27] 原田康德, 浜田昇, 渡辺慎哉, 宮本衛市: 開放分散に対応した kamui 環境,
情報処理学会研究報告, 90-SF-37 90-PL-27, 1990.
- [28] 原田康德, 宮本衛市: 送出メッセージの型宣言機構に基づいた オブジェ
クトの部品化について (君はどんな言葉を話すの?), ソフトウェア科
学会第 7 回全国大会論文集, 1990.
- [29] 原田康德, 浜田昇, 渡辺慎哉, 宮本衛市: 分散型環境 kamui について, 情
報処理学会記号処理研究会報告, 90-SYM-57, 1990.
- [30] 本田康晃, 渡滋, 大沢英一, 所真理雄: 成長するオブジェクトのモデル, ソ
フトウェア科学会第 7 回全国大会論文集, 1990.
- [31] Johnson, R., Graver, J., and Zurawski, L.: Ts : an optimizing compiler
for smalltalk, *Proc. OOPSLA '88*, 1988.
- [32] Jones, M. and Rashid, R.: Mach and matchmaker: kernel and language
support for object-oriented distributed systems, *Proc. OOPSLA '86*,
1986.
- [33] 久野靖: 多重継承と強い型付けを持つオブジェクト指向言語 misty, コン
ピュータソフトウェア, Vol.6, No.3(1989), pp.9-18,
- [34] Madsen, O. and Magnusson, B.: Strong typing of object-oriented lan-
guages revisited, *Proc. ECOOP/OOPSLA '90*, 1990.
- [35] Meyer, B.: *Object-oriented Software Construction*, *SERIES IN COM-
PUTER SCIENCE*, PRENTICE HALL, 1988.

- [36] Nierstrasz, O. and Papathomas, M. : Viewing Objects as Patterns of Communicating Agents, *Proc. ECOOP/OOPSLA '90*, 1990.
- [37] Notkin, D.: Proxies: a software structure for accommodating heterogeneity, *Softw. Pract. Exper.*, Vol.20, No.4(1990), pp.357-364.
- [38] Raj, R., Tempero, E., Levy, H., Black, A., and Hutchinson, N.: Emerald: a general-purpose programming language, *Softw. Pract. Exper.*, Vol.21, No.1(1991), pp.91-118.
- [39] Shilling, J. and Sweeney, P.: Three steps to views: extending the object-oriented paradigm, *Proc. OOPSLA '89*, 1989.
- [40] Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, 1986.
- [41] 武村巧司, 原田康德, 渡辺慎哉, 宮本衛市: Kamui toolkit-分散環境における toolkit, 情報処理北海道シンポジウム '91 情報処理学会北海道支部, 1991.
- [42] 渡辺慎哉, 原田康德, 三谷和史, 宮本衛市: 場とイベントによる並列計算モデル-kamui88, コンピュータソフトウェア, Vol.6, No.1(1989), pp.41-55.
- [43] 原田康德, 渡辺慎哉, 宮本衛市: 開放系名前付けモデルを導入した開放分散型言語 Kamui-C, ソフトウェア科学会第 8 回全国大会論文集, 1991.
- [44] Wegner, P.: Dimension of object-based language design, *Proc. OOPSLA '87*, 1987.
- [45] Wirfs-Brock, A. and Wilkerson, E.: An overview of modular smalltalk, *Proc. OOPSLA '88*, 1988.

- [46] 横手靖彦, 所真理雄: 並行オブジェクト指向言語 ConcurrentSmalltalk, コンピュータソフトウェア, Vol.2, No.4(1985), pp.2-18.
- [47] 米澤明憲, 柴山悦哉, Briot, J.P., 本田康章, 高田敏弘: オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1, コンピュータソフトウェア, Vol.3, No.3(1986), pp.9-23.
- [48] Robin Milner : A Calculus of Communicating Systems, *LNCS 92*, 1980.
- [49] 檜崎, 吉田: 並列協調処理モデル Cellula の実現と評価, 日本ソフトウェア科学会第6回大会論文集, 1989.
- [50] Matsuoka, S. , Kawai, S. : Using Tuple Space Communication in Distributed Object-Oriented Languages, *Proc. OOPSLA '88*, 1988.
- [51] Carriero, N. , Gelernter, D. : Linda in Context, *Comm. ACM*, Vol.32, No.4(1989), pp.444-458.
- [52] 渡辺 慎哉, et.al.: 並列オブジェクト指向モデルにおけるイベント処理順序保存機構, 日本ソフトウェア科学会第6回大会論文集, D4-2, 1989.
- [53] 川口 雄一, et.al.: 放送とメッセージ到達順序に関する CCS の拡張, 日本ソフトウェア科学会第7回大会論文集, A4-2, 1990.
- [54] 戸村 哲, et.al.: プロセス代数モデルに基づく並行オブジェクト指向言語の意味定義, コンピュータソフトウェア, Vol.7, No.2, 1990.
- [55] Six, H., Voss, J.: DIWA: A Hierarchical Object-oriented Model for Dialog Design, *Engineering for Human-Computer Interaction*, Ed. G.Cockton, IFIP, 1990.

- [56] Logrippo, L., et.al.: An Interpreter for LOTOS, A Specification
Language for Distributed Systems, *Softw. Pract. Exper.*, Vol.18,
No.4(1988), pp.365-385.

Appendix A

Kanmi-C 言語仕様

A.1 文法

A.1.1 プログラム

Kanmi-C のプログラムは次の通り。

プログラム

プログラムは、変数宣言、関数宣言、関数体、および main 関数からなる。

変数宣言は、変数名、変数の型、および変数の初期値からなる。

関数宣言は、関数名、関数の引数、および関数の戻り値からなる。

関数体は、関数の実行体であり、関数の引数と関数の戻り値とを対応させる。

main 関数は、プログラムの実行体であり、プログラムの実行を開始する。

プログラムの実行は、main 関数の実行から始まる。

変数

変数は、変数名と変数の型からなる。

関数

関数は、関数名、関数の引数、および関数の戻り値からなる。

関数体

Appendix A

Kamui-C 言語仕様

A.1 文法

A.1.1 プログラム

Kamui-C のプログラムは先頭が

```
KamuiC
```

から始められているファイル (.k で終了する名前) である。これは、将来同じファイルに異なった言語の記述を許すことができるようにしている。

その他に各ファイルで共通となる情報を記述するためのヘッダファイル (.h で終了する名前) がある。次のデータ型宣言、プロトコル宣言は通常このヘッダファイルないで宣言され、全体で共通に利用される。

C++ のプログラムの記述と混合するために、

```
C {  
    <C++ のプログラム>  
}
```

が用意されている。Kamui-C で用意されていない機能はこれによって直接 C++ を記述する。

A.1.2 データ型宣言

メッセージの引数などに使われる型を宣言する構文である。これには2つのタイプがある。

```
deftype <TypeName>
```

これは、固定長で特別な処理が必要ないデータの宣言である。例えば、新しく点のデータを宣言してみよう。

```
C {  
    typedef struct {  
        int x,y;  
    } point;  
}  
deftype point
```

ここで、deftypeはC++のデータの宣言は行なっていないく、別に定義しなければならない点に注意してほしい。

次の例は、X-Window のイベントのデータ型の宣言である。

```
C {  
    #include <X11/Xlib.h>  
}  
deftype XEvent
```

このように、あらかじめC++のデータ型であるものもKamui-Cのデータ型として宣言可能である。

次に用意されている構文は、特殊な構造を持つデータの宣言である。これまでに、不定長のデータ構造などが定義されている。その構文は、


```
deftype <TypeName> { <toRemote> } { <toLocal> }
```

である。ここで、<toRemote> <toLocal>は、それぞれのデータ構造をネットワークの共通な形式に変換するプログラムをC++によって直接記述する。この記述法は Kamui-C の実行時ライブラリに依存するため、ここではくわしくは述べない。それらの情報は、<Kamui3/libprotocol.h>などの使用法を参考にして欲しい。

Kamui-C が最初から用意しているデータ型は、

int

ID

Str

である。int は整数である。ID はオブジェクト ID である。Str は最後が 0 で終了する文字列へのポインタである。Str は不定長のデータである。さらに、<Kamui3/libprotocol.h>には、

```
typedef struct vdatabody {  
    int size;  
    int buf[1];  
} *vdata;
```

が、宣言されている。これは、大きさがsize 個のint の配列である。これにより、不定長のデータが表現できる。

A.1.3 プロトコル宣言

幾つか関係のあるメッセージをプロトコルとして共通に宣言する。

```
defprotocol <ProtocolName> {  
    <Message> ;
```



```

    <Message> ;
    ...
}

```

ここで、<Message>は、

```

<MesType> <MesName> ( <Type> <ArgName> , ... )

```

である。<MesType>、<ArgType>は前節で述べたデータ型の他に、ここで宣言したプロトコルも型として利用できる。この場合は実際にはそのプロトコルを受信可能なオブジェクトのIDが渡される。さらに、<MesType>には、非同期メッセージの型としてsend と、値の必要のないメッセージの型としてvoid ができる。C++で用いることができる int * のような構成された型は直接は書くことができず、一度typedef int * intp; として型に名前を付けてから用いる。

例として、

```

defprotocol pSample {
    send a(int x);
    void b();
    Str  c(pSample obj);
}

```

は、3つのメッセージを含んでいるプロトコルの宣言である。send a(int x) は整数を引数として送る非同期型のメッセージである。void b() は引数と値がない同期型のメッセージである。Str c(pSample obj) は、オブジェクトIDを引数として送り、値として文字列を返すメッセージである。ここで渡されるオブジェクトIDは、pSample というプロトコルを受け付けることができる、という型が付けられたものである。

プロトコルの宣言の内部で別のプロトコルを型として使うことが可能なため、それらの相互参照が生じる。そのために、仮の宣言が用意されている。

```
defprotocol <ProtocolName>
```

これは、この名前がプロトコル型であることのみを宣言している。その内部は後で宣言するので、相互参照が可能となる。C++のクラス宣言にも同様のことがある。

以上の宣言がファイル間で共通の宣言を行なう構文である。

A.1.4 クラス宣言

クラスの宣言は、

```
defclass <ClassName> [ <Protocols> ] <SuperClass> {  
  <InstVars>  
}
```

である。<SuperClass>は省略可能である。<Protocols>は、入出力のビューに応じた宣言で以下のようなものである。

```
<ProtocolType> , - <ProtocolType> <VarName> ...
```

<Protocols>の例は、

```
pDraw , pWindow , -pEvent event
```

である。これは、受信のプロトコルとして、pDraw,pWindow を宣言し、送信のプロトコルとして-pEvent event を宣言している。送信のプロトコルの宣言はそのままインスタンス変数の名前の宣言にもなる。ここでは、event という名前の変数が宣言している。

<InstVars>はインスタンス変数やC++のメソッドをC++の構文を直接用いて宣言する。この<InstVars>の部分にオブジェクトの初期化の為のメソッドを定義することができる。

```
void startUp() {  
    初期化  
}
```

この型のメソッドを<InstVars>に定義することにより、オブジェクトが生成される際に、これが呼び出される。変数の初期化はC++の機能として可能であるので、<InstVars>に直接記述してもよい。しかし、C++の機能であるコンストラクタによる初期化を行なってはならない。コンストラクタはKamui-Cですでに使用されている。

例をあげる。

```
defclass cClass [ pTest , -pWindow win ] {  
    int x = 0;  
    int *array;  
    void startUp() {  
        array = new int [30];  
    }  
}
```

宣言されているインスタンス変数は、x, array, win の3つである。その中で、x は直接初期化され、array は初期化メソッドの中で初期化されている。

A.1.5 メソッド宣言

メソッドは、プロトコル単位で宣言する。

```
defmethods <ClassName> <ProtocolName> {
```



```

    <Method> ;
    <Method> ;
    ...
}

```

<Method>はメッセージの宣言とメソッドとの組みである。メソッドの本体はC++の構文を用いる。

```

defmethods cClass pSample {
    send a(int x) {
        printf("%d\n",x);
    }
    void b() {
    }
    Str c(pSample obj) {
        return "string";
    }
}

```

A.1.6 メッセージの送出

メッセージの送出は、プロトコル型の変数を通じて行なう。

```

pSample x;
x.a(10);

```

この例は、プロトコル型pSample xの変数から指されているオブジェクトに対して、メッセージa(10)を送っている。

変数の型と異なるプロトコルのメッセージを送る場合や、プロトコルの情報がついていないID型の変数からメッセージを送る場合には、一度送った

いプロトコルの変数へ代入してから、それを通じて行なうか、そのプロトコルへキャストしてメッセージを送出する。

```
ID x;
```

```
pSample(x).a(10);
```

これは、前の例を同様にプロトコル型 `pSample x` のメッセージ `a(10)` を送っている。`x` は ID 型であるからキャストを行なっている。

A.1.7 プロトコルの本体

プロトコル型の実行時ライブラリの為にリンクされるファイルのどれか一つにプロトコルの本体を記述する必要がある。

```
protocolBody
```

これは、プロトコルの本体（データの型変換などを行なっている）のプログラムを生成する。`protocolBody` が生成するプログラムは大きいので、この記述だけを含むソースを用意することで、コンパイルの速度は早くなる。通常 `protocol.k` というファイル名で、

```
KamuiC
```

```
include "protocol.h"
```

```
protocolBody
```

のような内容のファイルを用意し、それをコンパイルしリンクする。このファイルが再コンパイルされるのは、プロトコルの変更があったときだけである。

A.1.8 メインプログラム

メインプログラムを記述する構文は、

```
main {  
  
}
```


である。ここで、以下の変数がすでに宣言されており、その値を使うことができる。

`int argc` コマンドの引数の個数。

`char **argv` コマンドの引数。以上は Kamui-C 自身へのオプションの処理が終わってから呼び出されるので、それらを除いたものが渡される。

`int serv` このプロセスがサーバモードで実行しているときに真になる。

`pDB gDB` 各プロセスに存在しているデータベース。オブジェクトを名前前で登録できる。隣接したプロセス上の `gDB` を登録することにより、階層的なオブジェクトのアクセスができる。

`pKamuiC gKamuiC` Kamui-C プロセス固有の処理を行なうオブジェクト。実際には `gDB` と同じオブジェクトを指しているが、プロトコルが異なるため、別の変数から指されている。

最後の2つは、実行時ライブラリにより起動時に用意されているオブジェクトである。くわしくは後で述べる。

A.2 実行時環境

A.2.1 起動の流れ

Kamui-C コンパイラによって作成された実行モジュールは次のような順で動作する。

1. コマンドに与えられた起動用オプションの処理。

そのオプションは接続する相手のホスト名とポート名の設定（省略するとデフォルトが使用される）と、デバックレベルの設定などである。

2. 他のプロセスとの接続。

最初、指定されたサーバとの接続を試み、失敗すると自分がサーバとなって動作する。

3. ルートオブジェクトの生成。

オブジェクトデータベースと、プロセスに関する処理を行なうオブジェクトで、gDB, gKamuiC から指されているオブジェクトである。

4. ユーザのメインプログラムの処理。

main { }で宣言された部分である。

5. イベントとI/O等の処理の無限ループ。

ソケットや他のI/Oを監視し、データがこなければプロセスのイベントキューの内容を実行する。

A.2.2 起動時のオプション

Kamui-Cで作成されたコマンドには、次のようなオプションが最初から用意されている。

-h hostName 最初に接続するソケットのホストを指定する。これが指定されなければ、環境変数KAMUIHOSTの値が用いられる。

-p portNumber 最初に接続するソケットのポート番号を設定する。これが指定されなければ、環境変数KAMUIPORTの値が用いられる。まず、指定されたホストに対し自分がクライアントとして接続を試み、それに失敗すると自分はサーバとなる。

-d デバッグレベルを1にする。これが指定されなければ、環境変数KAMUIDEBUGの値が用いられる。

これらのオプションの処理の後、変数`argc`, `argv` の値は次の引数を指すようになっている。

A.2.3 ルートオブジェクトの機能

Kamui-C のプロセスにはそれぞれ、ルートオブジェクトが一つずつ生成される。それは、プロセスの終了、他のプロセスとの接続、などの機能と、オブジェクトの管理の機能がある。それらは、それぞれ`pKamuiC`, `pDB` というプロトコルで処理される。ルートオブジェクトは一つであるが、2つのプロトコルを用いているため、それぞれ別の変数`gKamuiC`, `gDB` によって指されている。

`pKamuiC` のなかでユーザが使用することができるのは次のようなものである。

`ID New(Str className);` 指定したクラスのオブジェクトの生成。別のプロセス上の `gKamuiC` に対して行なうことにより、そこへのオブジェクトを生成できる。

`send quit();` それが属しているプロセスの終了。これは、正しく後始末をおこなっている。

`send connectTo(pKamuiC other);` 他のプロセスと直接ポートを接続する。
これは内部的に、`connectMe,connectToForce` を呼んでいる。

`send debugLevel(int level);` プロセスのデバッグレベルを変更する。

`gDB` は、オブジェクトに名前を付けて管理する機能を持っている。各プロセス上の `gDB` が互いに参照されているため、そのアクセスをたどることにより、離れたプロセス上のオブジェクトも参照できる。`pDB` では次のようなメッセージが開放されている。

`send add(Str name, ID object);` オブジェクトを名前で登録する。

`ID get(Str name);` 登録されたオブジェクトを得る。もし、見つからない場合には ID が 0 であるオブジェクトが返る。

`send remove(Str name);` 登録されているオブジェクトを名前で削除する。

`send removeByID(ID object);` 登録されているオブジェクトを ID で削除する。

`add, get` ので用いられるオブジェクト名に `':'` の文字が含まれている場合、それらを経路と見なして、連続したアクセスが行なわれる。名前の先頭から `':'` までの名前によって、登録されているオブジェクトを得る。そのオブジェクトは隣接したプロセス上の `gDB` であるので、さらに `':'` 以降の名前について同様の要求を行なう。これが、各プロセス上の `gDB` に対して連続して行なわれるので、直接自分が管理されていないオブジェクトでも得ることができる。

A.2.4 I/O 処理の追加

I/O 処理の無限ループでは、UNIX のシステムコール `select` を用いて、I/O の要求を待っている。ユーザが新しい I/O を接続したい場合 (例えば X-Server、標準入力との接続) には、次のようにして行なう。

```
C {  
  
#include <kamui3/hPortTbl.h>  
  
pCommand gCommand;  
  
struct stdinPort : Port {  
    stdinPort() : (0) {  
gPortTbl.Add(this);  
};
```



```

    virtual int doPort();
};

int stdinPort::doPort() {
    char strbuf[256];

    gets(strbuf);

    gCommand.enter(strbuf);

    return 1;
}
}

```

ここで、gCommand は文字入力が行なわれた際に呼び出したいオブジェクトを指しているものとする。Port のサブクラスとしてあるクラスを作成し、そのファイル記述子の番号によって初期化する。ここでは、標準入力なので 0 である。doPort は select によって入力が生じた際に呼び出されるメソッドであり、通常は 1 を返すようにする。しかし、この入力に異常が生じた場合には 0 を返し、このストリームを閉じることもできる。

