並行システムのオブジェクト指向設計に適したプログラミング言語の開発

上田賀一[†] 平井譲[†,☆]

対象世界のモデル化手法の1つとして，モデル化対象をモノ単位でとらえるオブジェクト指向方法論が注目されている．より自然なモデル化には，対象世界が有する並行性などの性質もあわせてとらえることが不可欠である．しかし，そのような性質が実装言語で直接実現されていることは少なく，開発者は，設計および実装工程においてそれらの性質を実現するための何らかの機構を用意しなければならない．これは大きな負担となり，追跡性を欠いたモデルとなる．本研究では，並行システムにおける重要な性質として，モノが持つ能動的/受動的性質の二面性に着目する．そして，能動的振舞いと受動的振舞いをあわせ持つモノの記述が可能な並行オブジェクト指向インタプリタ言語 Coos を開発する．本言語の並行オブジェクトモデルは従来の多重スレッドモデルを特化したものであり，1つのオブジェクトに3種類のスレッドを内包したものである．本言語は単一スレッドモデルが持つ記述容易性に加え，デッドロックを回避する言語となっている．さらに能動的振舞いのモデル化のために状態遷移形式による記述を可能とした．また，その状態は外部観測可能であり，オブジェクトどうしの同期に用いることも可能とした．

A Programming Language for Object-oriented Concurrent System Design

Yoshikazu Ueda[†] and Yuzuru Hirai[†,☆]

In software development, the object-oriented methodology is one of the useful modeling techniques, which is to map the things in the target world into the unit 'object'. In this case, it is indispensable to think about very natural character such as concurrency existing in the target world for more natural modeling. Most of the programming languages, however, do not have such a character. Therefore, the developers should prepare some mechanisms to replace it in the design and the implementation phases. This work is a heavy load for the developers, and the program is very different from the model obtained at the analysis and design phases. As a result, the understandability, reusability and productivity of software decrease. This study focuses the active/passive character of a thing in concurrent system. And, the concurrent object-oriented interpreter language Coos has been developed to describe object with both active behavior and passive one. The concurrent object model of this language is specialization of the traditional multi-thread model, and three kinds of threads are involved to one object. As a result, this language has both descriptive easiness and safety. Because the state chart was useful for modeling active behaviors in design phase, this thread is describable in state-transition-style. Moreover, the state of the object is observable from the outside, and can be used to synchronize with another.

1. はじめに

年々その要求が複雑/大規模化しているソフトウェア開発の生産性を高めるためには，対象世界および問題領域をいかに把握するか，すなわち，問題領域をどのようにモデル化するかが重要である[1]．このモデル化手法の1つとして，オブジェクト指向方法論がより自然なモデル化を行う技術として注目され，適用が進んでいる[2),3)]．この手法の特徴は，モデル化対象をモノ単位でとらえることである．この際に，並行性などの対象世界がごく自然に有する数々の性質もあわせてとらえることにより，さらに自然なモデル化が可能となる[4]．しかし，そのような性質を実装言語が直接持つことは少ない．そのため，開発者は設計および実装工程において，開発対象の本質的な部分とともに，対象世界が自然に持つ性質を実現する何らかの機構も用

† 茨城大学工学部
Faculty of Engineering, Ibaraki University
☆ 現在，三菱電機株式会社
Presently with Mitsubishi Electric Corporation

意しなければならない．これは特にプロトタイプモデリングを行う際に重大な障壁となる．開発者にとっては大きな負担となり，その結果作成されたソフトウェアは分析/設計工程で得たモデルとは異なるものとなる[5]．これは，ソフトウェアの理解容易性，再利用性を損ね，生産性低下の原因となる．

本研究では，並行システムを対象とし，そのソフトウェアシステムにおける重要な性質として，モノの持つ能動的/受動的性質の二面性に着目する．対象システム内に存在するモノは，条件判断などにより自分自身の振舞いを決定する能動的側面と，他のモノからの作業依頼によって何らかの処理を行う受動的側面を兼ね備えていると考える．これらの性質をそれぞれ把握することで，複雑になりがちな並行システムを適切にとらえることが可能になると考える．

本論文では，並行システムのモデル化において着目する性質について述べ，アプローチを考慮し，能動的振舞いと受動的振舞いをあわせ持つモノの記述が可能な並行オブジェクト指向プログラミング言語 Coos を開発する．本言語は前述の考え方に基づき，並行システムの開発において，対象の本質に開発者の意識を集中させることで，分析から実装までのよりスムーズな移行を目指した言語である．

また，本論文では，本言語を用いた並行システムの開発例を示し，関連研究との比較を行う．最後に，まとめと今後の課題について述べる．

2. 並行システムのモデリング

並行システムのモデル化にあたり，着目する性質と本研究のアプローチについて説明する．

2.1 並 行 性

並行性は，構成要素の並行的な処理をまとめることで全体の挙動が示される並行システムを開発するためには欠くことのできない性質であり，オブジェクト指向パラダイムにおいて，早くから意識された性質である．しかしながら，ほとんどの言語が並行性を持たないか，言語レベルでスレッドを用意することにより，ユーザに明示的に並行性の管理を行わせる．時間的，空間的な性能に対する要求が厳しい場合には，ユーザが陽に並行性を管理することも必要であるが，一般には，対象世界をより正確にとらえるための並行オブジェクトの存在は有用である．

実装言語での並行オブジェクトのサポートのほかに，方法論やモデリング言語で並行性をサポートするものがある[13]．ツールやモデル図などにより並行システムの開発を行いやすくするもので，設計工程でのモデル変換を支援するものである．

本研究では，これら 2 つのアプローチのいずれもが必要であると考える．たとえ言語が対象世界で必要とされる概念や性質をすべて持っていたとしても，言語だけで良いソフトウェア開発が行えるわけではなく，方法論やモデリング言語による開発支援が必要であると考える．また逆に，強力な方法論があったとしても，実装言語そのものに対象世界の自然な性質を持たせた方が，よりスムーズなソフトウェア開発が行えると考える．

2.2 能動的/受動的性質

並行性を有するモノは能動的側面と受動的側面の両方を兼ね備えることに着目する．

2.2.1 能動的性質

現実世界には自分自身の判断や周りの状況に応じて，自ら振舞いを起こす能動的なモノが多数存在する．先の並行性を持つモノを考えたとき，そのような能動的性質は必ず考慮されなければならない．

本研究のシステムでは，この性質は 1 つのオブジェクトにスレッドやプロセスを永続的に割り当てることで実現する．そのスレッドは，通常オブジェクトの能動的な振舞いを記述したメソッドの処理を行う．

2.2.2 受動的性質

現実世界の受動的性質とは，他のモノから動作を持ち掛けられることにより何らかの振舞いを起こすことである．これは従来のオブジェクト指向パラダイムにおけるオブジェクトにメッセージを送ってメソッドを起動するという一連の動作に置き換えてとらえることができる．実際，Eriksson ら[1]や丸山[6]は，受動的なオブジェクトは従来のオブジェクト同様，スレッドを持たず，他のオブジェクトが持つスレッドによって処理が行われる非並行オブジェクトであるとしている．

本研究では，受動的性質についてこれらとは異なる捉え方をする．すなわち，オブジェクト内部にスレッドを持ち能動的な振舞いをする並行オブジェクトにも，受動的な振舞いが存在すると考える．

2.3 モデリングと実装のつながり

本研究では，能動/受動オブジェクトあるいはエージェント/オブジェクトといった能動/受動的特性でオブジェクトを区別しモデル化するのではなく，すべてのオブジェクトが能動/受動的側面の両方をあわせ持つこととした．これにより，現実世界のオブジェクトのままのより自然なモデル化ができ，オブジェクトが細分化され過ぎることを抑えることができる．また，能動/受動的特性でオブジェクトを区別することを要求しないため，OMT 法をはじめとする多くのオブジェ

クト指向方法論において，そのモデリング方法を変更することなく，追加的思考でモデル化することが可能となる．

モデリングを柔軟にしても，実装言語が能動/受動的側面をあわせ持つ並行オブジェクトを記述できなければ，実装言語に合わせた再モデリングが必要となる．そこで本研究では，能動/受動的側面をあわせ持つ並行オブジェクトを記述できる並行オブジェクト指向言語 Coos を開発した．これにより，設計工程と実装工程のそれぞれでモデルをとらえる概念を，能動/受動的側面をあわせ持つ並行オブジェクトという単位に揃えることができ，各工程のモデルの差異を抑えることができる．このことは，モデリングと実装のつながりをより滑らかなものにするといえる．

3. 並行オブジェクト指向言語 Coos

本章では，本研究で着目する性質を考慮してモデル化した並行システムに適した実装を行うために開発した，並行オブジェクト指向プログラミング言語 Coos について述べる．

3.1 設計方針

- 能動/受動的振舞いをあわせ持つ並行オブジェクトのサポート
  対象世界を自然に記述するために，1 つの並行オブジェクトに能動的振舞いと受動的振舞いを定義できるようにする．
- プログラムコードの記述容易性とデッドロックに対する安全性の両立
  並行型言語では逐次型言語には見られなかった並行計算特有の問題が発生する．複雑なメッセージ送信形態の存在や，データに対する排他制御の必要性などがそうである．一般的には，専用の表記を用意し，ユーザに明示的に指定させる．しかし，そのようなプログラムはしばしば読みにくいものとなる．本言語では，できるだけそのような表記を導入しないようにする．しかし，その結果頻繁にデッドロックを起こすようでは意味がない．したがって本言語では，前述の記述容易性を確保しつつ，デッドロックを極力避けられる機構を用意する．
- 状態遷移形式での記述の実現
  能動的振舞いのモデル化において状態図が有効であることから，状態遷移形式でも記述できるようにする．
- 高い移植性の確保
  本言語上で起動するアプリケーションプログラム，そして言語処理系本体にも高い移植性を持たせることにする．具体的には，インタプリタ方式を採り，言語処理系自体もカーネルスレッドなどの OS 依存の機能を使用しないようにする．

3.2 文法

前節の設計方針に基づいて本言語の開発を行った．ここでは，以降で本言語について述べるにあたり，必要となる本言語の基本的な文法について説明する．

3.2.1 メッセージ送信と代入

メッセージ送信文は以下の形式をとる．

receiver message parameter: valueObj ...

メッセージ受信側のメソッドでは引数名でもって渡された値の参照が可能であり，引数が 1 つしかない場合は，引数名を省略することも可能である．また，メッセージ送信は連鎖させることも可能であるが，送信順序は括弧を用いて明示的に示さなければならない．

代入は `<-` で行われる．本言語は基本的にメッセージ送信と代入操作により処理が進む．

3.2.2 クラスの定義

クラスは `class` で定義し，クラス名の後ろに“`extend` スーパクラス名”を付けることによりスーパクラスを指定できる．省略した場合は，Actor クラスがスーパクラスになる．クラスの内部では，以下のものが定義できる．

- 属性：`attribute` で定義する．属性を使用するときは‘`&`’が名前の頭に付く．なお，属性はインスタンス外部から直接アクセスすることはできない．
- 定数属性：`const` で定義する．定義時に 1 度だけ値を代入可能である．以降，変更することはできない．
- メソッド：`method` で定義する．必要となる引数名を指定できる．

さらに，それらの定義の先頭に `share` を指定するとそれぞれクラス（定数）属性，クラスメソッドが定義できる．また，これら以外にも後述する状態と状態遷移が定義可能である．

クラスに対し，`new` メッセージを送ることでインスタンスを生成することができる．

3.3 Coos の並行オブジェクトモデル

本言語の中心的な役割をなす並行オブジェクトのモデルについて述べる．オブジェクト指向言語では，共有データはオブジェクトの属性に，それに対する操作は同じオブジェクトのメソッドとしてカプセル化される．このため，共有データの一貫性を保障することが容易になる．一貫性を保つ具体的な方法[3]は，1 度に 1 つのメッセージの実行しか許さない単一スレッドモ

デルと，同時に複数のメッセージの実行を許す多重スレッドモデルによって異なる．

単一スレッドモデルでは，到着順にメッセージの一時保存を行うメッセージキューをオブジェクトに用意し，その順に対応するメソッドの起動を行う．その実行が終了するまで次のメッセージは処理されず，メソッドに対する原子性を保障する．これによって，オブジェクトの属性に対する排他的な操作が実現でき，属性の一貫性が保障される．その反面，単一スレッドモデルでは実際の並行性が減少し，全体としてデッドロックしやすくなる．それを回避するためにプログラムに工夫を凝らさなければならなくなる．

一方，多重スレッドモデルでは，デッドロックは発生しにくいが，1つのオブジェクト内にも並行性が存在し，オブジェクトの属性に対してユーザの責任で排他制御を行い，属性に対する一貫性を保障しなければならない．そのため排他制御が埋め込まれたプログラムは分かりにくいものとなる．

本言語では，メソッドの原子性を確保しながら，なおかつデッドロックの回避を行うため，両方のスレッドモデルを融合したモデルを採用する．具体的には，多重スレッドモデルを基に，スレッドを次の3種類に分け，それぞれに明確な役割を持たせた．

- メッセージハンドラスレッド
  メッセージキューに格納されたメッセージを1つずつ順に処理するスレッドである．メソッドの原子性を保障するため，このスレッドは1つのオブジェクトに付き1つしか存在せず，単一スレッドモデルと同様の処理を実現する．
- メッセージレシーバスレッド
  メッセージを受け取り，メッセージキューに格納するスレッドである．この処理は，単一スレッドモデルではメッセージ受信の際に言語処理系が行っているが，このメッセージ受取り処理を独立させたことにより，単一スレッドモデルと同じ形態で，デッドロックをある程度回避することが可能になる．
- オブジェクトボディスレッド
  能動的振舞いの処理を行うスレッドである．多重スレッドモデル同様，他の2種類のスレッドとは独立に並行に動作する．

オブジェクトボディスレッドとメッセージ処理にかかわる2つのスレッドはオブジェクト内の並行スレッドとなるため，同一属性へのアクセスに対するデータの一貫性の保証が問題となる．本研究では，それぞれのスレッドの役割を定め，オブジェクトボディスレッドは能動的振舞いを処理し，メッセージ処理にかかわるスレッドは受動的振舞いを処理するために用意している．そのため，オブジェクトの属性はいずれかの管理下において利用されるように，ユーザによりモデル化され，実装されることが望まれる．少なくとも属性の更新はいずれかのスレッドだけから行われることが必要である．このことは先の問題を解決するだけでなく，理解容易なモデル化の観点からも必要なことである．

3.4 デッドロックの回避

並行オブジェクト指向言語におけるデッドロックを回避するには，メソッドの原子性を保ちつつ，待ち状態をできるだけ発生させないようにすればよい．

3.4.1 読み出しメソッド

本言語では，メッセージ受信時に割り当てられるレシーバスレッドを利用して，読み出しメソッドの特別な処理の実現を行う．読み出しメソッドに対するメッセージをキューに格納せず，レシーバスレッドが直接そのメソッドを起動する．これを `messageReceiver` メソッドの再定義により実現できるようにしている．ただし，再定義における読み出しメソッドの選定と同一属性のアクセスに対するメッセージ順序の一貫性の保証はユーザが再定義プログラムにおいて行わなければならない．

3.4.2 自己再帰的なメッセージ

並行オブジェクト指向言語では，自分自身への再帰的なメッセージ送信を他のメッセージと同様に処理した場合，必ずデッドロックを引き起こしてしまうという問題がある．そのため，自己再帰的なメッセージは特別に処理しなければならない．

自分自身への直接的なメッセージ送信は，プログラムをより理解しやすくするためで，並行に実行されることを期待して行われるものではないと見なす．したがって，自身へのメッセージ送信は従来のオブジェクト指向言語と同様に，同じスレッド上でメソッドを直接実行することとした．

他のオブジェクトを介した間接的なメッセージ送信については，返信を必要とするメッセージ送信において，メッセージ送信の経路情報の添付を行う．メッセージを受け取ったレシーバスレッドは添付された経路情報を基に再帰的なメッセージを識別し，レシーバスレッドがハンドラスレッドに成り代わることで，そのメッセージの処理を特別に行う．なお，この回避手法は空間的，時間的コストが大きいので，現在の実装ではこのデッドロックの回避を起動時オプションで選択するようにしている．

3.5 メッセージの処理

- 動的なメッセージ処理機構
  メッセージの処理を行うレシーバスレッドとハンドラスレッドの処理内容は，ユーザが自由に変更可能である．これは前述したデッドロックの回避のためだけでなく，様々なメッセージ処理を実現するために利用することができる．並行オブジェクトがメッセージを受信した際に，レシーバスレッドにより起動されるのが messageReceiver メソッドである．メッセージを処理する際に，ハンドラスレッドにより起動されるのが messageHandler メソッドである．messageReceiver メソッド，messageHandler メソッドにより，本言語は柔軟なメッセージ処理を実現している．これらのメソッドは CLOS[7]における MOP の一部を分かりやすい形でユーザに提供したものととらえることができる．
- 3 つのメッセージ送信形態
  並行オブジェクトへのメッセージ送信は，過去型，現在型，未来型の 3 種類の形態がある[10]．それぞれの形態はメッセージ送信に対する返信のあり方により分類され，本言語では次のように判別される．

  過去型： メッセージを送信するだけの形態であり，返信に関する処理は行わない．返信される値を必要としないメッセージ送信文は過去型となる．

  ```
  "Hello" println;
  ```

  現在型： メッセージ送信後，送信先からの返信があるまで送信元の実行をブロックして待つ形態である．メッセージ連鎖がこれにあたり，メッセージ送信文を括弧で括ることで明示的に記述することができる．

  ```
  (((5 + 4) * (2 - 1)) / 3)
  ```

  未来型： メッセージ送信後，実行を継続し，送信先からの返信が必要になった時点で実行をブロックして待つ形態である．返信される値を変数に代入するという代入文とメッセージ送信文との組合せが未来型となる．代入された値が必要となるまで実行を継続できる．

  ```
  var <- anObj getValue;
  ```

  実際のプログラミングでは，過去型や未来型の送信形態において，送信先オブジェクトと同期させたい場合には，明示的に括弧を付けて現在型にすることで可能である．

  ```
  var <- (anObj getValue);
  ```

  このように，本言語では特別な記法を導入することなく，従来のプログラミング言語で自然に見られた記法により形態は判別される．
  また，現在型/未来型の送信形態において，送り主オブジェクトを別のオブジェクトに送って代わりに返信してもらう，という処理の委譲もできる．
- 実行結果の返信
  メソッドの実行終了にともなう返信は‘^’によって行うことができる．また，値の返信後もメソッドの処理を続ける返信には‘^^’を用いる[8]．

3.6 状態遷移形式による記述

能動的な振舞いを行うボディスレッドの処理内容を状態遷移形式で定義できるようにした．状態はクラス定義内で「state @状態名」によって定義される．記述例を以下に示す．

```
state @playing
  do: { cdplayerUI displayCurrentTime };
state @pausing
  entry: { cdplayerUI lightOnPause }
  exit:  { cdplayerUI lightOffPause };
```

状態にはアクティビティとして，entry，exit，do がブロックオブジェクトと組で指定できる．それぞれは，状態入場時，退場時，継続中を意味する．なお，この 3 つ以外のアクティビティを指定することはできない．遷移は，「transition @状態名 -> @遷移先状態名」で定義される．記述例を以下に示す．

```
transition @initial -> @playing
  cond:   { true }
  action: { self playCD };
transition @playing -> @pausing
  cond:   { self isPaused }
  action: { self pauseCD };
transition @pausing -> @playing
  cond:   { self isPaused }
  action: { self resumeCD };
transition @pausing -> @final
  cond:   { self isStopped };
```

状態名は state で定義したものを指定する．なお，@initial と @final は予約状態で，それぞれ初期状態と終了状態を示す．状態遷移は初期状態から始まり，終了状態になるとボディスレッドが解放される．

遷移には，cond と action をそれぞれ指定できる．前者は遷移条件であり，ステートチャートのイベントとガード条件に相当するものである．指定されたブロックオブジェクトを評価し，真なら状態遷移が起こる．後者はアクションであり，遷移の際に必ず指定の

表 1 Actor クラスのメソッド
Table 1 Methods in Actor class.

メソッド名	説 明
initialize	クラスからインスタンスが生成された直後に起動される初期化専用メソッドである.
messageReceiver	引数：sender: message: parameter: レシーバスレッドの実際の処理内容が定義されている．currentState メソッドを読み出しメソッドにする処理と引数で受け取ったメッセージをメッセージキューに格納する処理が既定義されている．このメソッドはレシーバスレッドのみが利用可能であり，ユーザが起動することはできない．例外として，messageReceiver メソッドからのみスーパクラスのそれを起動できる.
messageHandler	引数：sender: message: parameter: ハンドラスレッドの処理の一部が定義されている．Actor クラスでは，後述の startMethod メソッドを用いてメソッド起動を行うよう定義されている．messageHandler メソッドからスーパクラスのそれが起動できるという例外を除いてどのメソッドからも起動できない.
objectBody	Actor のサブクラスで，ボディスレッドが実行する処理内容を定義する抽象メソッドである．このメソッドはボディスレッドのみが利用可能であり，ユーザが起動することはできない.
startBody	ボディスレッドの割り当てを行うメソッドである．レシーバには self 以外に指定できない．このメソッドが起動されて初めて並行オブジェクトは能動的な振舞いを行う．なお，1 つのクラスに状態遷移と objectBody メソッドの 2 つが定義されていた場合，objectBody メソッドは無視され，状態遷移の処理が行われる.
startMethod	引数：name: メソッド名 parameter: ハッシュ表 メソッドの起動を行うメソッドである．このメソッドもレシーバには self 以外指定できない.
currentState	現在の状態名を文字列オブジェクトで返すメソッドである．状態遷移が定義されていない場合は null オブジェクトが返される．このメソッドは読み出しメソッドとして処理される．レシーバが並行オブジェクトであること以外の起動制限はない.
stateTrans	引数：to: 遷移先状態名 action: 遷移アクション メソッド側から状態遷移を起こすメソッドである．レシーバには self 以外指定できない.

表 2 Actor クラスの属性
Table 2 Attribute in Actor class.

属性名	説 明
messageQueue	受信したメッセージを格納するためのリストオブジェクトである．messageReceiver，messageHandler メソッド内からのみ利用可能である.

ブロックオブジェクトが評価/実行される.

以上，状態遷移記述として状態と遷移の基本的な部分を記述可能にした．これらの状態と遷移は継承可能であり，また，サブクラスにて再定義が可能である.

状態遷移は次のように実現されている．定義された状態や遷移を基に，ボディスレッドは，現在の状態が持っている遷移の遷移条件を評価し，すべて偽なら現在の状態の継続アクションを評価し，真の遷移があれば，

- 現在の状態の退場アクション
- 遷移アクション
- 遷移先の状態の入場アクション

の順で評価を行い，状態遷移を行う.

なお，入/退場アクションおよび遷移アクション評価中は状態遷移は起こらない．したがって，これらのアクションはデッドロック回避のために実行時間の短い処理が望まれる.

本言語での状態（と遷移）は，前述したように，設計からのスムーズな移行を考慮して導入されたものである．そのため，実装のためにアドホックに利用される状態遷移とは異なり，対象の本質を表す情報であり，その状態は外部から観測しても有用であることが多い．そこで，並行オブジェクトが持つ特殊なメソッドとして，現在の状態名を文字列で返す `currentState` メソッドを用意した．相手の状態により自分の振舞いを決定するという処理に利用できる．また，オブジェクトがある状態になるまで待つということも記述できる.

```
signal[@red @green]
```

このように並行オブジェクトが格納されている変数名に状態名のリストを続けて記述することにより，オブジェクトが指定の状態になるまで待つことを表現でき，オブジェクトの状態によってメッセージの送信を遅延させるという処理に利用できる.

3.7 並行オブジェクトと組込みオブジェクト

すべての並行オブジェクトは抽象クラス `Actor` をスーパクラスに持つ．並行オブジェクトに関する特殊なメソッドおよび属性は，この `Actor` クラスで定義されている．メソッドを**表 1** に，属性を**表 2** に示す.

また，本言語では，実用性を考え，すべてのオブジェクトを並行オブジェクトとしているわけではない．数値やリストなどの基本的な値やデータ構造は非並行オブジェクトであり，それらは振舞いを再定義すること

表 3 組込みオブジェクト一覧
Table 3 Built-in objects.

クラス名	説　明
Object	本言語の継承木のルートに位置する抽象クラスである．インスタンスの生成を行うクラスメソッド (new) や，インスタンスそのものの比較を行うメソッド (=) などの全オブジェクトの基本的なメソッドが定義されている．
Actor	全並行オブジェクトのスーパクラスとなる抽象クラスである．
Integer	整数クラスである．四則演算などの演算や，比較が行える．
String	文字列クラスである．文字列の加工などが行える．
Hash	ハッシュ表クラスである．整数もしくは文字列をキーにあらゆるインスタンスが格納できる．
List	リストクラスである．要素の格納や取り出しなどのリスト操作が行える．
Block	プログラムコードをまとめるクラスである．まとめたプログラムを評価できる．
True, False	それぞれ真と偽を表すクラスである．条件分岐の処理に利用する．
Null	このクラスのインスタンスは比較の一部以外のメッセージを受け付けない．

のない具象クラスとして，言語処理系と一体化した組込みオブジェクトとなっている．また，非並行オブジェクトのメソッドは，並行オブジェクト側のスレッドを用いて実行される．本言語で用意されている組込みオブジェクトを**表 3** に示す．

4. 並行システムの開発例

本章では，簡単な例を通して，並行性を有するモノが持つ能動性と受動性の二面性を考慮した並行システムのオブジェクト指向設計について考察し，先の言語を用いて実装する．なお，この章では新たな設計法を提示するのではなく，能動/受動の両性質を認識した並行システムのモデル化が既存の設計法によってどのように行われるかについて考察する．ここで取り上げる例は，片側 1 車線の道路における自動車の挙動を調べることを目的とした交通シミュレーションであり，適用するオブジェクト指向方法論は，OMT 法[2]を基に UML [1),9)]も参考にしたものである．

4.1 交通シミュレーションの要件

交通シミュレーション例（**図 1**）の要件を以下に示す．

- 片側 1 車線の道路における自動車の挙動のシミュレーション（片側車線のみ）を行う．
- 道路上を走るのは普通車のみである．
- 普通車は，前方車両との車間を適切に保つよう，加速/減速を行いながら走行する．
- 普通車から見た走行位置には左側/中央/右側があり，通常中央を走行する．
- 普通車はときどき道路左に左折する．つまり，左側に寄った後，減速，停止して道路上からいなくなることがある．
- 普通車は，前方車両が左側で減速中の場合，右側から追い抜きを行う．

4.2 分析・設計

まず，対象システムの分析を行う．本例では，クラス図を作成することによって，対象システムの基本的な構造をとらえる（**図 2**）．要件中，クラスとなりうるのは道路と普通車だけである．そのうち，普通車は能動的なクラスとなり，前後車両と 1 対 1 の関連を持つ．そして，普通車は道路の上を走る．

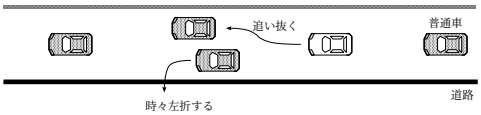

図 1 例：交通シミュレーション
Fig. 1 Example: traffic simulation.

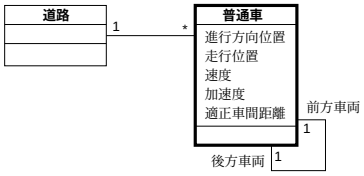

図 2 交通シミュレーションのクラス図（分析）
Fig. 2 Class diagram of traffic simulation in analysis.

この段階で，能動的性質と受動的性質について考察する．明らかに，普通車はそれぞれ並行に動作する存在であり，状況に応じた走行は能動的な振舞いである．それに対し追い抜かれるという行為は，もしそれにより何らかの動作をともなうなら受動的な振舞いであるといえる．問題はそれらの振舞いをどうモデル化するかである．

次に，普通車の並びの管理法を決定しなければならない．分析モデルでは，普通車自身がそれぞれの前後車両を知っていると想定されている．しかし，普通車は左折や追い抜きによりその並びが変化する．関連がその動作に従い自動的に変化したりすることはないので，道路クラスによって普通車の並びの一括管理を行い，普通車が直接関連を持つのは前方車両だけにした．これにより，道路と普通車クラスの詳細な役割が決定

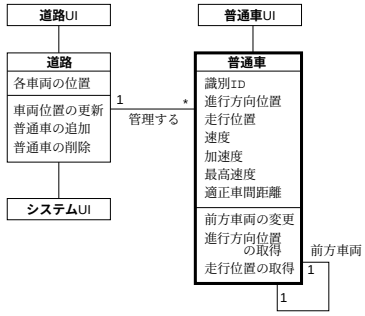

図 3 交通シミュレーションのクラス図（設計）
Fig. 3 Class diagram of traffic simulation in design.

する．

普通車

- 状況に応じた走行をする．
- 追い抜きをした場合，道路にそのことを伝える．
- 道路から外れた（ユーザインタフェース部と関連する）場合も道路に伝え，自分自身を削除する

道路

- 普通車の生成（または削除）を行う．
- 普通車の並び順を管理し，並びに変更があれば関係する車両に伝える．
- ユーザからの普通車の生成要求を受け付ける．

設計されたクラス図を**図 3** に示す．道路，普通車の表示を司る各々のユーザインタフェースクラスと，ユーザからのシステムに対する要求を受け付けるシステムユーザインタフェースクラスがそれぞれ追加されている．

さらに，動的モデルの作成を行う．本研究で着目する能動的/受動的性質は，モノの振舞いに関する性質であり，この動的モデルの作成を通してとらえられるものである．まずは受動的な振舞いのモデル化に関しては，前述したように，従来のオブジェクト指向パラダイムでのメッセージのやりとりとしてとらえることができる．したがって，メッセージを受け取る動作は受動的振舞いである．

メッセージのやりとりはシーケンス図を用いてモデル化され，要件および先に明確にしたそれぞれの役割から作成した本例のシーケンス図を**図 4** に示す．

シーケンス図より，普通車オブジェクトが受け取るメッセージは位置の取得と前方車両の変更の 2 つである．すなわち，これらが普通車の受動的な振舞いとなり，それらのメッセージに応答するメソッドを用意する．

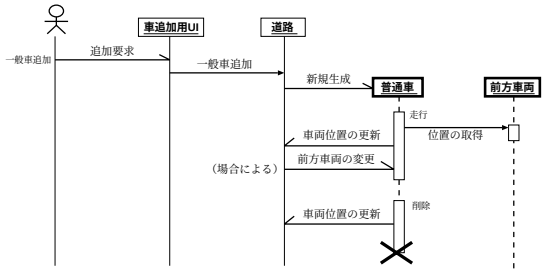

図 4 交通シミュレーションのシーケンス図
Fig. 4 Sequence diagram of traffic simulation.

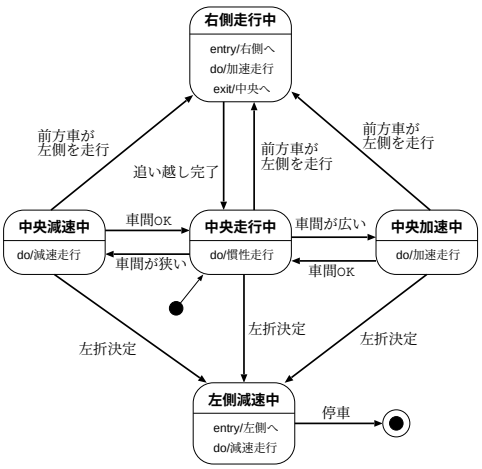

図 5 普通車クラスの状態図
Fig. 5 State chart of car class.

シーケンス図で表されるメッセージの一連の流れには，必ず発生源がある．並行システムにおいて，その発生源は必ず能動的なオブジェクトもしくはユーザなどシステム外部の能動的なモノである．つまり，図 4 の普通車オブジェクトにおける走行や削除など自然発生的な動作は，すべて能動的な振舞いから発生する動作であるといえる．このように受動的な振舞いが示されているシーケンス図から，能動的な振舞いを把握するためのヒントを得ることができる．

次に，要件や洗い出した役割，シーケンス図のヒントから普通車の能動的な振舞いについてモデル化を行う．

OMT 法をはじめとする多くの方法論において，1 つのオブジェクトの振舞いをモデル化するためのモデル図として，状態図がよく用いられる．そこで，状態図を用いて能動的な振舞いをモデル化する（**図 5**）．

4.3 普通車クラスの実装

クラス図（図 3）を基に，普通車クラスの属性（関連を含む）とメソッドを定義する（**図 6**）．各位置情報を返す `getLocation` メソッドと `getPosition` メ

```
class Car {
  attribute id;
  attribute position;
  attribute location;
  attribute speed;
  attribute accel;
  const maxSpeed <- 16;
  const space    <- 32;
  attribute foreCar;
  attribute road;
  attribute carUI;

  method initialize [num fcar road] {
    &id <- num;
    &position <- 0;
    &location <- 0;
    &speed <- 0;
    &accel <- 1;
    &foreCar <- car;
    &road <- road;
    self startBody;
  };

  method changeForeCar [car] {
    &foreCar <- car;
  };

  method getPostion [] {
    ^&position;  };

  method getLocation [] {
    ^&location;
  };
};
```

図 6 普通車クラスの定義（属性/メソッド）
Fig. 6 Definitions of attributes/methods of car class.

```
state @runningCenter
  do:   { self inertiaRun };

state @accelingCenter
  do:   { self acceleRun };

state @decelingCenter
  do:   { self deceleRun };

state @runningRight
  entry:{ &location <- -1 }
  do:   { self acceleRun  }
  exit: { &location <- 0  };

state @decelingLeft
  entry:{ &location <- 1 }
  do:   { self deceleRun };
```

図 7 普通車クラスの定義（状態）
Fig. 7 Definitions of states of car class.

ソッドについては，これらを読み出しメソッドとして `messageReceiver` メソッドを定義する方がよいが，ここでは省略している．

次に，状態図（図 5）から状態と遷移の定義を行う（**図 7**，**図 8**）．普通車の走行を処理しているそれぞれの状態の継続アクティビティは，メソッドとして定義することにした．

最後に状態遷移において必要となったメソッドについて定義する（**図 9**）．

以上の定義を `Car` クラスの定義にまとめ，残りの道路クラスと各々の UI クラスを作成すると交通シミュレーションシステムが完成する．作成したシステムの実行例を**図 10** に示す．

4.4 評　　価

交通シミュレーションの開発例を通して，モデル作成および本言語に対し評価を行う．

4.4.1 モデル作成

能動的振舞いと受動的振舞いをそれぞれ独立にモデル化できるため，対象をより正確に，詳細に把握することが可能になり，動的モデル作成における段階的開発が可能になった．2 つの振舞いは 1 つのモノの異なる側面であり，各々は互いに関連している．そのため，一方をより詳細にとらえると，他方に対してモデル化のきっかけを得ることができる．具体的には，あるオブジェクトの能動的振舞いが他に対してメッセージを送信すると，それは別のオブジェクトの受動的振舞いになり，反対に，あるオブジェクトへのメッセージ送信の源をたどると，別のオブジェクトの能動的振舞いが判明する．

例においても，まず要件から受動的振舞いをとらえ，シーケンス図を作成し，その結果から能動的振舞い

```
# Initial state
transition @initial -> @runningCenter
  ;

# Running in central way
transition @runningCenter -> @decelingCenter
  cond: {((&foreCar getPosition) - &position) < &space };

transition @runningCenter -> @accelingCenter
  cond: {((&foreCar getPosition) - &position) > (&space * 2) };

transition @runningCenter -> @runningRight
  cond: {(&foreCar getLocation) == 1 };

transition @runningCenter -> @decelingLeft
  cond: {(self isLeftTurn) == true };

# Decelerating in central way
transition @decelingCenter -> @runingCenter
  cond: {((foreCar getPositon) - &position) > &space };

transition @decelingCenter -> @runningRight
  cond: {(&foreCar getLocation) == 1 };

transition @decelingCenter -> @decelingLeft
  cond: {(self isLeftTurn) == true };

# Accelerating in cetral way
transition @accelingCenter -> @runningCenter
  cond: {((&foreCar getPosition) - &position) < (&space * 2) };

transition @accelingCenter -> @runningRight
  cond: {(&foreCar getLocation) == 1 };

transition @accelingCenter -> @decelingLeft
  cond: {(self isLeftTurn) == true };

# Running in right way
transition @runningRight -> @runningCenter
  cond:   {((&foreCargetPosition) - &position) < 0 }
  action: { &road changeOrder id: &id };

# Decelerating in left way
transition @decelingLeft -> @final
  cond: { &speed <= 0 };
```

図 8 普通車クラスの定義（遷移）
Fig. 8 Definitions of transitions of car class.

```
method isLeftTurn [] {
  ((100 random) == 0) if
    then: { ^true }
    else: { ^false };
};

method inertiaRun [] {
  &position <- &position - &speed;
  &carUI put pos: &position loc: &location;
};

method acceleRun [] {
  &speed <- &speed + &accel;
  &position <- &position - &speed;
  &carUI put pos: &position loc: &location;
};

method deceleRun [] {
  &speed <- &speed - &accel;
  &position <- &position - &speed;
  &carUI put pos: &position loc: &location;
};
```

図 9 普通車クラスの定義（メソッド）
Fig. 9 Definitions of methods of car class.

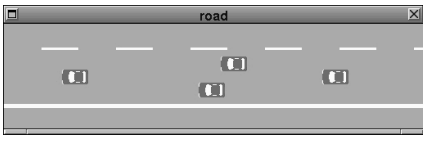

図 10 交通シミュレーションの実行例
Fig. 10 Execution example of traffic simulation.

の一部を抽出した．そして，その結果作成した状態図が，遷移条件もしくはアクションなどにおいて他のオブジェクトに対しメッセージを送信する場合は，それが源になって新たなシーケンス図を作成できる．このように，両方の観点を行ったり来たりしながら，対象の漸時的なモデル化が可能である．

4.4.2 実装言語

とらえた 2 つの振舞いをそれぞれ別々に記述できることにより，分かりやすいプログラムを作成でき，また，実装時における開発者の負担を減少させることが可能になった．能動的振舞いと受動的振舞いの分離はオブジェクト内における処理のモジュール化を促進するといえる．

例においても，特に状態のアクティビティに対して，それぞれ 1 つのメソッドとして切り分ける傾向にある．アクティビティは 1 つの処理として独立できる可能性が高いため，オブジェクト内の機能分割を分かりやすい形で行うことが可能である．

当然のことながら，それは，対象をいかに上手にモデル化できたかに依存する．しかし，再設計が必要になる場合でも，本言語の記述容易性は，スムーズなモデルの再設計，再実装を可能とする．

5. 関連研究

本章では，本研究と関連する研究について比較を行い，本研究の位置付けを明確にする．

5.1 ABCL/1

ABCL/1[10]は実用的な並行オブジェクト指向言語の祖と呼べる言語である．本言語でも採用しているメッセージ送信の過去型，現在型，未来型を最初に提唱し，速達モードの導入など，現実世界に存在する多様なメッセージのやりとりの自然な形での実現を目指した言語である．また，ABCL/1 はそれを裏付ける数学的モデルが定義されており，計算モデルとしても完成度が高い言語である．

本言語は ABCL/1 から始まった単一スレッドモデルを基礎とする並行オブジェクト指向言語の流れの新たな形ととらえることができる．本言語は ABCL/1 のメッセージ処理機構を発展させ，レシーバスレッドとハンドラスレッドの処理をユーザが記述できることにより柔軟なメッセージ処理の実現を試みている．また，ABCL/1 のオブジェクトは本研究での受動的なオブジェクトであり，本言語では，そのオブジェクトモデルに能動的な性質を付加している．

5.2 y

並行オブジェクト指向言語 y[11]は，計算モデル ψ に基づいたネットワークソフトウェアを記述することを主な目的とした言語である．

y では本言語における `objectBody` メソッドと同等の，オブジェクトの生成から消滅までのライフサイクルを記述できるシナリオを持ち，オブジェクトの能動的な性質に着目している．しかし，他のオブジェクトからのメッセージの処理についてもシナリオで記述し，本研究のように能動的な性質と受動的な性質の二面性については考慮されていない．また，デッドロックについても特に注意が払われておらず，たとえば，自分自身へメッセージを送信すると必ずデッドロックする．本言語ではその問題に対し，より実用的な解決策を示している．

5.3 抽象状態と p6

本研究では，能動的な振舞いを状態遷移でとらえ，その状態遷移が外部から観測できることは有用であるとした．これと類似の概念を提唱するものに抽象状態とそれに基づく並列オブジェクト指向言語 p6[12]がある．

抽象状態では，オブジェクトがとる意味のある（時に抽象的な）状態は外部に公開し，利用できることが望ましいとしている．そして p6 言語では抽象状態を効果的に使用し，並行言語においてこれまで難解にならざるをえなかった種々の同期について自然な形で表現することを可能としている．

本言語における状態は，主に設計工程における能動的な振舞いをとらえた結果として着目するものである．そのためオブジェクトの振舞いに直接関係しており，抽象状態のようにオブジェクトの内部状態と抽象状態との写像を明示的に行う必要はない．その反面，本来なら外部に公開しなくてもよい情報までも公開する可能性がある．

抽象状態と p6 が示したオブジェクト間の同期問題に関する結果は，本言語においても有効に適用できる可能性があり，その検討を行う必要があると考える．

5.4 ROOM 法

ROOM 法[13]は，並行システムの一種であるリアルタイムシステムに特化したオブジェクト指向開発方法論である．アクタと呼ばれる自律的なモノを中心にモデル化を進め，そのモデル化には状態図を独自拡張したものを使用する．さらに，モデル化過程はツールによって分析から実装まで支援されている．

本研究は方法論の構築を目指したものではないが，状態図に着目し，それを基に実装まで行うという点が本研究と共通している．ただし ROOM 法では受動的な性質との両立については意識されていない．

6. ま と め

本研究では，対象世界をより自然にとらえ，トレーサビリティの高い効果的なソフトウェア開発を実現するためには，現実世界に存在する概念を分析工程から実装工程まで一貫して保持することが重要であることを主張した．そして，並行性を有するソフトウェアシステムの開発において重視する性質として，1つのモノに内在する能動的性質と受動的性質の二面性に着目した．これらの性質を考慮に入れたソフトウェアの設計について考察し，さらに，これらの性質を言語仕様に取り入れた並行オブジェクト指向言語 Coos の開発を行った．

本言語は，設計工程における状態図の有用性から状態遷移の記述を言語レベルで行えるようにし，それによって能動的な振舞いを表現できるようにした．また，能動的な振舞いはメッセージ受信によるメソッドの起動で表現される．これらにより，設計モデルから大きな変換をすることなくプログラムが記述でき，設計から実装へのスムーズな移行を実現することができた．

本言語ではデッドロック回避のために messageReceiver メソッドの再定義を可能にしているが，データの一貫性はユーザが保証しなければならないという問題がある．実装工程だけでこの保証を得ることは労力を要すると考えられるが，本研究で示した読み出しメソッドにかかわる messageReceiver メソッドの再定義では，設計工程のシーケンス図などでメッセージの順序関係を詳細にとらえることにより，保証のある messageReceiver メソッドの再定義自体はそれほど困難ではないと考えている．しかしながら，今後あらゆる状況に照らし合わせてこのことを検証していく必要がある．

受動的な振舞いと能動的な振舞いの関係のさらなる検討も今後の課題としてあげられる．

このことに関しては，本論文中で受動的な振舞いと能動的な振舞いの2つは互いに関連があり，片方からもう片方を抽出できるということを示唆した．

しかし，2つの性質の関係はそれだけでなく，1つのオブジェクト内において，能動的な振舞いが受動的な振舞いに，あるいはその逆に影響を与えるという状況が十分に起こりうる．そして，それらの振舞いは互いで1つの動作を行うと考えられる．より自然なモデル化を目指すには，この2つの振舞い間の協調的な動作についても明確にとらえる必要がある．

また，能動的な振舞いは状態図でモデル化するが，その状態図におけるイベントはたいていの場合他のオブジェクトから発生するものである．しかしそれを受け取ることは，オブジェクトの能動的な振舞いと見なすことになる．このように，現在，状態図は能動的な振舞いと受動的な振舞いの一部の双方をとらえていることになっている．モデルから本言語への変換をよりスムーズに行うために，これら2つの振舞いのとらえ方を再検討し，それぞれを明確に定義する必要がある．

参 考 文 献

1) Eriksson, H.E. and Penker, M., 杉本宣男，落合　修，武田久美子（監訳）：UMLガイドブック，トッパン (1998).
2) Rumbaugh, J. ほか，羽生田栄一（監訳）：オブジェクト指向方法論 OMT モデル化と設計，トッパン (1992).
3) 所真理雄，松岡　聡，垂水浩幸（編）：オブジェクト指向コンピューティング，岩波書店 (1993).
4) 増原英彦：並列・分散環境上でのオブジェクト指向プログラミング技術，オブジェクト指向最前線 '98，pp.211–221，朝倉書店 (1998).
5) 金村星吉，上田賀一：並行オブジェクト指向言語 COOL の開発，電子情報通信学会技術研究報告（COMP），Vol.96, No.487, pp.17–24 (1997).
6) 丸山勝己：並列オブジェクト指向言語 COOL，情報処理学会論文誌，Vol.34, No.5, pp.963–972 (1993).
7) 井田昌之，元吉文男，大久保清貴（編）：Common Lisp オブジェクトシステム—CLOS とその周辺，*bit* 別冊，共立出版 (1991).
8) Yokote, Y.: *The Design and Implementation of ConcurrentSmalltalk*, World Scientific Co. Pte. Ltd. (1990).
9) Fowler, M. and Scott, K., 羽生田栄一（監訳）：UML モデリングのエッセンス —標準オブジェクトモデリング言語の適用，アジソン・ウェスレイ・パブリッシャーズ・ジャパン (1998).
10) 米澤明憲，柴山悦哉，Briot, J.-P., 本田康晃，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, pp.9–23 (1986).
11) Choi, K.-R., 小島一人，野呂昌満，原田賢一：並行オブジェクト指向言語 y の通信機構，第 50 回情報処理学会全国大会論文集（5），pp.63–64 (1995).
12) 久野　靖，大木敦雄：抽象状態に基づく並列オブジェクト指向言語 p6，情報処理学会論文誌，Vol.38, No.3, pp.563–573 (1997).
13) ObjecTime Limited: ObjecTime, Technical Papers, http://www.objectime.com/otl/

(平成 11 年 11 月 11 日受付)
(平成 12 年 5 月 11 日採録)

上田 賀一（正会員）

1961 年生．1989 年名古屋工業大学大学院工学研究科電気情報工学専攻博士後期課程修了．同年同大学工学部電気情報工学科助手．1990 年茨城大学工学部情報工学科講師．現在に至る．ソフトウェア工学，プログラミング言語に関する研究に従事．工学博士．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE Computer Society 各会員．

平井 譲（正会員）

1973 年生．1999 年茨城大学大学院理工学研究科情報工学専攻博士前期課程修了．同年三菱電機（株）入社．在学中，ソフトウェア開発法やオブジェクト指向言語の研究に取り組む．