システムソフトウェアと
オペレーティング・システム 92－1
（２００３．２．２７）

エージェント指向オペレーティングシステム AG における リフレクティブエージェントの実現手法

瀧本 栄二 † ††　滝沢 泰久 †　毛利 公一 †††　大久保 英嗣 †††

† 株式会社 ATR 適応コミュニケーション研究所
†† 立命館大学大学院理工学研究科
††† 立命館大学理工学部

エージェント指向オペレーティングシステム AG は, 環境の変化に動的に適応可能なオペレーティングシステムである. AG では, オペレーティングシステムの構成要素として, 自律動作するリフレクティブエージェントを用いている. 本稿では, 各エージェントが環境の変化に適応するための手法であるリフレクションの実現手法について述べる. また, リフレクションの結果として創発される, システム全体の適応性に関する考察について述べる.

A Construction Method of Reflective Agents on Agent Oriented Operating System AG

Eiji Takimoto † ††　Yasuhisa Takizawa †　Kouichi Mouri †††　Eiji Okubo †††

† ATR International Adaptive Communications Reserch Laboratories,
†† Graduate School of Science and Engineering, Ritsumeikan University,　††† Faculty of Science and Engineering, Ritsumeikan University

An agent oriented operating system AG can adapt to changes of environment dynamically. AG uses reflective agents which act autonomously as operating system's components. In this paper, a construction method of reflective agents is described. And a study of adaptivity of whole systems which will be emergent as a result of reflection is also described.

1　はじめに

現在, 我々は, 分散システムにおける環境の動的な変化に適応可能とすることを目的としたエージェント指向オペレーティングシステム (以下 OS と記す) AG[1] を開発している. AGでは, OS の適応性を実現するために, その構成要素としてエージェントを用いている.

分散環境では, 計算機やデバイスの追加, 削除等のシステム構成の変化と, 計算機負荷の変化やデバイスの競合などの実行環境の変化が発生する. これらの変化が発生する時刻を予測することは困難である. また, 変化によって発生する影響は, 大域的なものではなく, 一部のシステム構成や特定資源に関するものなど, 局所的であることが多い. このような, 環境の変化に OS 全体が適応するのではなく, 影響を受ける部分のみが適応することで, 急激な変化にも対応することが可能になると考えられる.

AGにおけるエージェントは, このような環境に柔軟に対応するために, リフレクション (自己反映) を用いている. リフレクションとは, プログラムコードやその解釈・実行機構をプログラム自身から参照・変更することである. リフレクションを用いることで, エージェントは自らの動作を変更し, その環境に適した処理を行うことが可能となる.

本稿では, 適応性を実現するために必要となる, リフレクティブエージェントを実現するための手法について述べる. 本手法では, 即応型スレッドと熟考型スレッドと呼ぶ 2 種類のスレッドを用いてエージェントを構成する. 即応型スレッドは, 外部からの要求を処理する. 熟考型スレッドは, 環境の変化に応じて, 即応型スレッドの動作を制御する. このようにエージェントを構成することで, 容易にリフレクティブエージェントの実現と拡張を行うことができる. さらに, 本稿では, リフレクティブエージェントを用いた, システムとしての適応性に関する考察についても述べる.

以下, 本稿では, 2 章で AGの概要について述べ, 3 章で我々が提案するリフレクティブエージェントの構成手法について述べる. 4 章では, 他の研究事例との比較を行い, 最後に 5 章で本稿のまとめと今後の課題について述べる.

2　AG の概要

AG は, 図 1 に示すように, 複数のエージェントとオブジェクトによって構成されている.

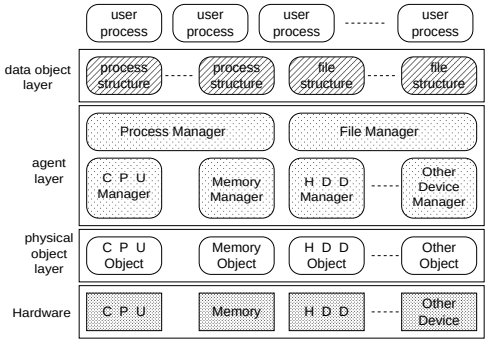

図 1　システム構成

オブジェクトには, 計算機資源を抽象化する物理オブジェクトと, システム内で利用するデータ構造であるデータオブジェクトの 2 種類がある. データオブジェクトは, それを利用するエージェントによって管理される. 各物理オブジェクトは, CPU, メモリ, 各種周辺デバイスに 1 対 1 で対応付けられる. 物理オブジェクトは, 資源の状況を表す内部変数と, 資源操作のためのメソッド群を持つ. 資源管理は, エージェントが物理オブジェクトを操作することで行われる. すなわち, AG では, 資源の管理と操作を物理オブジェクトとエージェントに分けることで, ポリシとメカニズムの分離を実現している.

各エージェントは, C 言語で記述され, それぞれ異なる仮想アドレス空間を持つプロセスとして実現される. 1 つのエージェントは, 複数のスレッドを持つことが可能である. エージェントをプロセス化することで, エージェントを独立したものとし, 移動性の実現や動的な追加, 変更, 削除などを容易にする. また, 複数のスレッドを動作させることで, システム全体のみならず, エージェント単位で複雑な処理の実現を可能とする.

OS としての機能は, 複数のエージェントの協調処理により実現される. AG では, ある機能を実現する複数のエージェントを動作させることができる. ユーザやエージェントは, 自分の要求がどのエージェントによって処理されているかを意識する必要がない. 物理オブジェクトに関しても同様に, 特定のエージェントに 1 対 1 で結びつけられ

る必要がない. すなわち, 他計算機上のオブジェクトを選択し, 操作することで, 任意の計算機の資源を管理することができる.

3 リフレクティブエージェント

エージェントが環境の変化に対して適応的に動作するためには, エージェントは自律的に自身の振る舞いを変更できなければならない. ここで, 振る舞いとは, 外部から対象を眺めたときに観測できる事象を指す. AGでは, 自らの振る舞いを制御することができるエージェントをリフレクティブエージェントと呼ぶ. リフレクティブエージェントを構築するためには, エージェントを適用する環境やそれに関連したエージェントの性質について明らかにする必要がある.

以下, 本章では, リフレクティブエージェントの環境と性質, そしてそれらに基づいた構成とリフレクションの実現手法について述べる.

3.1 エージェントの環境と性質

OS の動作は, イベント駆動で処理を開始し, 複数のモジュールやメソッドを逐次的に呼び出すことで行なわれる. 処理のオーバヘッドは, アプリケーションの実行の妨げとならないように小さくなければならない. AGにおけるエージェントは, OS の一部であるため, このような即応的な性質を持つことを保証しなければならない.

一方, リフレクションや自律動作を実現する場合, このような即応的な性質とは異なる性質が必要となる. 例えば, リフレクションを行なうためには, 環境を観測する必要がある. また, 観測結果の内容から, どのように自身の動作を変更すべきかを決定しなければならない. 通常, このような処理はコストが高く, 即応的に行なうことが困難である. したがって, リフレクションのための処理を, 即応的に動作するモジュールなどの内部に組み込むことは適切でない. リフレクションのような非即応的なモジュールと, それを起動するためのイベントを別に用意する方法がある. しかし, この方法には, そのイベントをいつ, 誰が起こすのかが静的に決められてしまうという欠点がある. さらに, エージェントがイベントに拘束されてしまい, 柔軟性が低下するということも考えられる. したがって, OS という動作環境を考えた場合, エージェントの基本的な動作とリフレクションのための動作は, 完全に分離しなければならない.

一般に, エージェントが認識できる環境が複雑であればあるほど, エージェントの適応性や自律性を向上させることができる. しかし, AGが対象とする流動的に構成が変化する分散システムでは, すべての環境をエージェントが認識することは困難である. また, エージェントが環境の変化に対して過度に反応すると, それだけシステムにかかる負荷が高くなる. したがって, エージェントが観測すべき環境は, エージェントが実現する機能とリフレクション手法に応じて単純化すべきである.

AGでは, エージェントは, いくつかのエージェントと協調しながら資源を抽象化するオブジェクトを操作することで, ユーザの要求を処理する. したがって, エージェントにとって必要十分となる環境とは, ユーザ, 協調関係にある他エージェント, 管理対象となるオブジェクトであるといえる.

実際のエージェントと環境との関わりのうち, ユーザと他エージェントとの関係は, 通信関係と置き換えることができる. なぜなら, ユーザやエージェントは, 必要とするエージェントとのみ通信を行うためである. 通信の有無を関係の有無と捉えると, オブジェクトを除く環境の変化とは, 通信内容の変化であると見做すことができる. 通信状態から環境の変化を捉えるようにすることで, 必要のないエージェントなどを考慮せずに済む. ただし, オブジェクトは, それ自身が何らかの通信メッセージを発行しないため, エージェントはオブジェクトの状態を観測しなければならない.

3.2 リフレクティブエージェントの構成

以上の考察から, エージェントには, 受け取った要求を処理する即応的部分と, 環境のモニタリングやリフレクションそのものを自律的に判断して行なう熟考的部分の両方が必要であることが分かる. そこで, AGでは, 身体性認知科学の分野で用いられる Subsumption Architecture の考え方を導入し, 以下に挙げる構成要素を用いて, 図 2 に示す構成でエージェントを実現する.

環境データ 環境に関するデータ群である. 本データ群は, 環境からの要求やその処理結果などのログデータや, 環境を観測して得られたエージェント毎に異なるデータの集合である.

メソッドテーブル メソッドのアドレスと機能を保持するテーブルである. メソッドテーブルは, すべてのスレッドによって共有される.

即応型スレッド　ユーザと他のエージェントからの要求や割り込み処理など, 即時性を要する処理を行なうスレッドである. 即応型スレッドは, 受け取ったイベントに対応するメソッドを, メソッドテーブルを参照して呼び出すことでその処理を行う.

熟考型スレッド　主に環境の観測, 環境データの更新, 環境に基づいたメソッドテーブルの変更を行なう. また, GC(Garbage Collection) のように, 特定のイベントに依存しない処理を行う際のトリガを発行する.

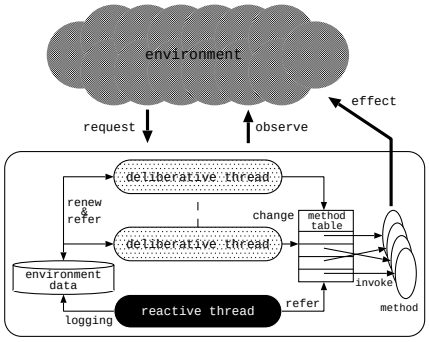

図 2　リフレクティブエージェントの構成

3.3　リフレクションの実現手法

3.1 節で述べたように, OS におけるエージェントは, 即応的な側面と非即応的な側面を同時に実現しなければならない. *AG* では, 即応型スレッドと熟考型スレッドの 2 種類のスレッドを用いることで, この問題を解決している. さらに, Subsumption Architecture を採用し, 熟考型スレッドを階層的に構成することで, エージェントの振る舞いを拡張することを容易にしている.

Subsumption Architecture は, 階層型アーキテクチャに基づく知的ロボット実現のためのアーキテクチャである. 従来の方法では, 入力から出力までの一連の動作を機能毎にモジュール化し, 逐次的に処理が行われる. これに対して, Subsumption Architecture では, 並列的に活性化される多数の層によってエージェントを構成する. これらの層は, それぞれ何らかのエージェントの内部行動を実現したものである. 高位の層はより低位の層に依存する形で作られ, 低位の層を包摂することができる. 各層は, 結線を通して非同期メッセージを送るモジュールの組から構成されており, これらのモジュールは拡張有限状態機械を用いて構築される. 各層は, これらの結線によって送られるメッセージを置換したり抑制することで, 下位層の動作を制御することができる. また, Subsumption Architcture では, すでに存在する層の上に, 新たな動作を定義する層を追加することで, エージェントの拡張を行う.

AG では, Subsumption Architecture における階層型アーキテクチャを, スレッドを用いて実現する. ソフトウェアであるスレッド内のデータフローを外部から制御することは困難である. したがって, 本手法では, エージェントの振る舞いに最も影響する部分であるメソッド呼出しに関する部分をメソッドテーブルとして抽出している. メソッドテーブルは、メソッドへのポインタを格納しており、リフレクションは, このメソッドテーブルを操作することで実現される. 本手法では、メソッドテーブルの操作でリフレクションを実現するために、同じ機能を実現するメソッドをあらかじめ複数用意しておく。各メソッドは、一般に使われるものや特定の状況下において有効なものなどの異なるアルゴリズムを実装している。メソッドテーブルとメソッドの関係を図 3 に示す。

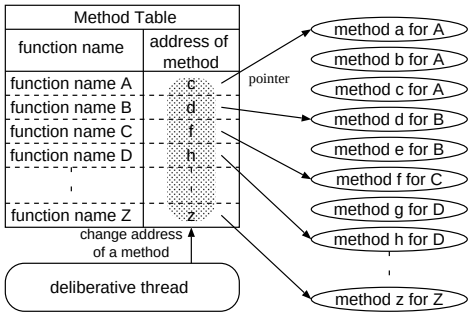

図 3　メソッドテーブルとメソッドの関係

また、図 2 に示すように、複数の熟考型スレッドを階層的に配置し、各階層間で同様の処理を行なうことで、より複雑な処理を実現することが可能となる。即応型スレッドは、外部からの要求を処理するのみであるが、エージェントが行なうべき処理は、即応的に行なわれるものだけではない。例えば、メモリのコンパクションやガーベジコレクションなどは、メモリの使用状況や CPU 負荷などに応じて行なわれるべきである。このような処理は、環境を把握している熟考型スレッドが行なう方が適している。この場合でも、上位の熟考型

スレッドが、下位の熟考型スレッドを同様の方式で制御することで、即応的でない処理に対してもリフレクションを実現することができる。

3.4 リフレクティブエージェントによる適応性の実現

OS のように大規模かつ複雑なソフトウェアでは、システム全体を適切に変化させることが困難である。*AG*では、エージェントを単位としてシステムを変化させることで、この問題を解決している。すなわち、*AG*における適応性とは、エージェントが個々の環境に適応した結果として創発されるものである。エージェントは、他のエージェントを環境として捉える。そのため、1 つのエージェントがリフレクションにより変化した場合、そのエージェントを環境の一部としているエージェントにとって、環境が変化したと捉え、リフレクションを行なうという流れが生じる。この影響の伝播によって、システム全体が適応的に変化していく。影響の伝播による変化の流れを以下に示す。

(1) 環境の変化が発生する。この変化は、ユーザ、またはエージェントからの要求やオブジェクトの変化として察知できる。

(2) エージェントは、この変化に対してリフレクションを行なう必要があれば、リフレクションにより動作を変更し、環境に適応する。特にその必要がなければ、何もしない。

(3) エージェントがリフレクションにより動作を変更したとき、当該リフレクションの影響が他のエージェントに及ぶか否かによって、次の場合に分けられる。

  (a) 変化が他のエージェントに及ばない場合は、その時点でシステム全体としての適応が完了する。

  (b) 変化が他のエージェントに及ぶ場合、これは、他のエージェントへの要求内容の変化という形で観測できる。したがって、要求を受け取ったエージェントは、(2) から処理を繰り返す。

3.5 本手法の特徴

本手法は、Subsumption Architecture を用いた階層型アーキテクチャにより、リフレクティブエージェントを構築する。本手法により、エージェントの持つ即応的な側面と熟考的な側面を明確に分割することが可能である。また、熟考型スレッドを複数用いることで、リフレクティブエージェントの適応性を向上させることができる。さらに、既存のエージェントに新しい熟考型スレッドを階層の最上位に追加することで、容易にエージェントを拡張することができる。

また、リフレクティブエージェントが受け取るメッセージから、環境の変化を察知するという手法により、環境である他エージェントやオブジェクトの状態を常に観測しなければならない従来の手法に比べ、環境の変化を容易に知ることができる。あるエージェント内部でリフレクションが発生したことは、メッセージに影響がない限り、関係する他エージェントには隠蔽され、明示的に通知しなくてもよい。したがって、環境の変化とそれに伴うリフレクションは、その影響を受けるべきエージェント内で局所化することができる。リフレクション自体は、メソッドの変更を行なうだけであるため、エージェントのインタフェースに影響がない。メソッドの変更自体は、即応型スレッドと熟考型スレッドの間で共有されるメソッドテーブルを書き換える処理で行なわれる。書き換え処理自体にかかるオーバヘッドは、非常に小さく、エージェントの処理に与える影響を最小限にすることができる。

エージェントを設計･構築する際には、他のエージェントがどのような変化を発生させるかを考慮せずに行なうことができる。インタフェースの定義さえ変更されなければ、どのような内部メカニズムを持つエージェントであっても、システム全体の動作を損なうことなく動作させることができる。これにより、通常のエージェントでは対応できない状況になった場合などに、対応できるエージェントに切替えることでシステムを適応させることなどが可能となる。

4 関連研究

4.1 従来のリフレクション実現手法

従来のリフレクションの実現手法は, 以下の 3 種類に分類することができる.

(1) 言語レベルでの実現手法

(2) メタレベルアーキテクチャに基づく手法

(3) 再コンパイルによるコード変更手法

(1) の手法では, リフレクションの計算モデルを言語に採り入れ, リフレクションのための記述を可能としている. ABCS/R2[4] は, リフレクションの計算モデルである Hybrid Group Architecture を採り入れた並列オブジェクト指向言語である. ABCL/R2 では, Hybrid Group Architecture により, オブジェクト単位でのリフレクションとオブジェクトグループ単位でのリフレクションを同時に可能としている.

(2) の手法では, メタオブジェクトと呼ばれるオブジェクトの上位概念を導入している. この手法では, オブジェクトとメタオブジェクトの関係を動的に変更することでリフレクションを実現する. Apertos[2] では, ポリシとメカニズムを, それぞれメタオブジェクトとオブジェクトとして実現し, reflector と呼ばれる機構によって, オブジェクトとメタオブジェクトの関係を変更することでリフレクションを実現している.

(3) の手法では, プログラムを再コンパイルし, 実行中のコードと入れ替えることによってリフレクションを実現する.

本稿で述べた手法は, エージェントの内部でポリシとメカニズムを分割しているという点で, (2) の手法に似ている. AG では, エージェント内部のポリシとメカニズムがエージェント内部で閉じている点で異なっている.

4.2 適応型 OS

適応性を実現した OS の事例として、Synthetix[3] がある. Synthetix では, ユーザインタフェースとメカニズムを分離し, それらを環境に合わせて繋ぎ替える (replugging) ことで, システムの動作を変更している. また, 環境は quasi-invariant と呼ばれるもので保持し, その監視は guard が行う.

replugging により, 提供するサービスのメカニズムを変更し, 環境の監視を行う点で, AG に類似している. しかし, Synthetix は, インタフェース単位での変更を行うのみである. AG では, ひとまとまりの機能を実現するエージェント単位で振る舞いを変化させ, かつエージェント間でその影響を与え合うことで, システム全体の振る舞いを適応させることが可能である.

5 おわりに

本稿では、Subsumption Architecture に基づいたリフレクティブエージェントの構築手法について述べた。本手法を用いることで、リフレクションを容易に実現し、そのオーバヘッドも小さいものにすることが可能となる。さらに、階層を追加することで、エージェントの拡張も容易に行なうことができる。

また、リフレクティブエージェントを用いることで、適応的な OS を構築することができる。リフレクティブエージェントによって OS を構成することで、リフレクションによる変化の局所化、実装の簡単化などが期待できる。

今後は、本手法に基づいた AG の実装を通して、本手法の妥当性を検証していく。特に、システム全体の適応性が、どのように創発されるかという点は非常に興味深い。また、2 種類のスレッドのスケジューリング手法、環境データとそこから環境の変化を察知するための手法についても検討を行なっていく予定である。

謝辞

本研究は、通信・放送機構の委託により実施したものである。

参考文献

[1] 瀧本 栄二, 芝 公仁, 大久保 英嗣, “エージェント技術を用いた分散オペレーティングシステムの構成手法”, 情報処理学会研究会報告 2002-OS-89,Vol.2002,No.13,pp.117–123(2002).

[2] Yasuhiko Yokote, “The Apertos Reflective Operating System: The Concept and Its Implementation,” OOPSLA’92 Proceedings, ACM, pp.414–434 (1992).

[3] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. “Fast Concurrent Dynamic Linking for an Adaptive Operating System”. The International Conference on Configurable Distributed Systems (ICCDS’96),1996.

[4] Hidehiko Masaharu, Satoshi Matsuoka, Takuo Watanabe, “ Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently”, In Proceedings of OOPSLA’92,1992.