形式的検証機能を備えたインターネットエージェントプログラミングシステム

櫟 肅之†

本論文では，形式的検証機能を備えたエージェントプログラミングシステムを提案する．その言語は，if-then ルールに基づく単純なセマンティクスを持ち，またインターネットエージェントの動作を記述するには十分な，いくつかのエージェントのアクションが与えられている．形式的検証に関しては，CTL のモデルチェックアルゴリズムを利用する．その入力として，エージェントシステムの振舞いの論理的記述が必要であり，従来はこの記述を人手により作成していたが，ここで導入される言語では，それを自動的に合成することが可能になる．また，検証すべき要求仕様に対して，その仕様に関連するプログラムの部分だけを自動的に抽出する方法を導入し，自動合成される状態遷移の論理的記述を簡約化することも可能にした．この言語に対する処理系は，Java 上のインタプリタとして実装されており，Jade などの他のエージェント開発・実行環境にひけをとらない実行スピードを実現している．

An Internet Agent Programming System with a Formal Verification Facility

Tadashi Araragi†

In this paper, we present an Internet agent programming system with a formal verification facility for the agent programs. The language has a simple semantics of "if-then rule" and a few selected actions of agents which are enough to describe agents' behavior working in the Internet. For the verification, we employ the model checking algorithm of CTL as the base engine. The language is designed so that a logical representation of behavior of agents, an input of a CTL model checker, which was manually created by programmers, is automatically derived from the programs. We also introduce a method of reducing the logical representation of behavior by extracting a part of programs which is essentially related to a given requirement specification to be verified. This system is implemented on Java and shows practical speed of execution compared with other agent systems such as Jade.

1. はじめに

昨今インターネットの利用は急速に拡大し，その利用形態も，これまでのメールや WWW などのような情報交換の手段から，Web サービスのように，ネットワーク上でシステムが連携して，より自動化の進んだサービスを提供する枠組みに発展しつつある．この Web サービスの枠組みを，より細かい個人レベルに推し進めたものが，インターネットエージェントである．そこではエージェントとよばれるソフトウェアが人間に代わってインターネット上で，情報の収集や，電子商取引の契約などを行う．各々のエージェントはそれぞれの目的を持って独立に開発され，ネットワーク上で適切な相手エージェントを見つけ，メッセージの送受信を通して互いに協調し，その目的を達成する．

インターネットエージェントの普及において，大きな課題の 1 つとなるのが，簡単に，間違いのないエージェントを作る開発環境の提供である．エージェントプログラミングは非同期分散プログラミングでもある．非同期分散プログラムでは通信（メッセージ送受信）のタイミングによって無数の可能な実行（列）が存在し，熟練したプログラマでも，自分のプログラムがそれらすべての可能な実行でつねに正しく動作するかを確認することは難しい．一方，エージェントの開発では，しばしば，他で独立に作られたエージェントのサービスを利用することを前提としている．そして，そのような状況では，エージェントにバグがあった場合，その波及範囲は大きく，いったん実行したエージェン

† 日本電信電話株式会社 NTT コミュニケーション科学基礎研究所
NTT Communication Science Laboratories, Nippon Telegraph and Telephone Corporation

トのバグの影響を除くことは容易ではない．したがって，そのようなバグを事前に，完全に除いておくことが必要になる．

上記の問題に対する 1 つのアプローチとして形式的検証手法の利用が考えられる．形式的検証では，システムの抽象的な実行により与えられた要求仕様がすべての場合（可能な実行）において満足されるかを論理的な形で効率良くチェックする．これまでに，さまざまな検証手法，ツールが開発され，分散システムのプロトコルの検証にも利用されてきた．しかし，これらの検証手法を利用するには，いったん，検証用の言語で，システムの動作を論理的な形に表現し直す必要があり，それを間違いなく行うにはやはり相当の熟練が必要となる．また，形式的検証のかかえる大きな問題の 1 つとして，探索空間の爆発があり，これを防ぐためには，対象とするシステムをあらかじめ適切に抽象化する作業が必要になる．一般のエージェントプログラマにとって，これらの作業を間違いなく行うことは非常に難しい．

このような問題を解決するため，本論文では，検証用の言語に翻訳し直す必要のない，直接形式的検証可能なエージェントプログラム言語のデザイン（Erdoes）を考案し，また，探索空間を縮小するためのシステム状態遷移モデルの自動的な簡約化の手法を提案する．提案する言語は，if-then ルールに基づく単純なセマンティクスを持ち，ここでは，そのセマンティクスに基づいて，時相論理 (F)CTL の symbolic モデルチェック[8] に必要なシステム動作（状態遷移）を表す論理式をプログラムから自動合成する．状態遷移モデルの簡約化では，検証する要求仕様が与えられたとき，その仕様に影響を及ぼしうるプログラムの部分のみを自動で抽出する slicing の手法を導入する．

一般に，分散システムの形式的検証では，実行可能なプログラムに対して直接の検証や，有効な抽象化作業の自動化は困難な課題である．ここでは，対象をインターネットエージェントにしぼり，エージェント特有の性質，すなわちエージェントが限られたタイプのメッセージの通信のみで互いをコントロールし，また各エージェントの自律性ゆえに，そのコントロールの関係も疎であることを利用して，これらのことを可能にする．

導入されたエージェントプログラミング言語に対し，Java をベースとしたインタプリタ型の処理系が実装されており，この処理系は Jade や Aglets などの代表的なエージェント開発・実行環境と比べ，遜色のない実行スピードを実現し，いくつかのエージェントサンプルアプリケーションがこの言語で開発されている．

以下，2 章では，まず分散システムとしてのエージェントの計算モデルを定め，そのモデルを反映したプログラミング言語を導入する．3 章では，2 章で導入したエージェントプログラムから，(F)CTL のモデルチェックアルゴリズムの入力となる，エージェントの動作を表す論理式の導出方法を示す．4 章では，この言語に特化した，検証作業効率化のためのプログラム簡約化手続きを説明する．5 章では，導入された手法についての議論を行い，その関連研究を説明する．

2. インターネットエージェントシステムのモデルとそのプログラミング言語

2.1 インターネットエージェントシステムの計算モデル

まず最初に，分散システムの観点から，インターネットエージェントの計算モデルを定める．各エージェントは分散システムの 1 つのプロセスとして扱う．そして，エージェントはグローバルに一意な名前を持つとする．エージェントが複数いる場合，システム全体の動作は，fair な interleaving モデルとして扱う．すなわち，システムの各実行ステップで，エージェントが 1 つだけ選択され，実行される．そして，各エージェントに対して，この選択実行が無限回行われることが保証されている．また，エージェントはつねに実行可能（enable），すなわち選択されたら必ずその実行状態を進めることができるものとする．

エージェントはメッセージベースと呼ばれるメッセージの集合を持つ．ここには，他のエージェントから受信したメッセージと，内部処理の結果をメッセージの形にしたものを保存する．

エージェントは，受信エージェントの名前と送信するメッセージを指定して，他のエージェントにメッセージを送信することができる．通信リンクは FIFO queue で，送信メッセージは受信側でハンドシェイクなしにメッセージベースに受け取られる．また，ここでは，エージェントの内部処理の結果は，メッセージの形で自分のメッセージベースに返されるものとし，1 回の選択実行で，内部処理とその結果のメッセージベースへの返却の 2 つの動作が完了するものとする．エージェントは各々プログラムあるいは状態遷移関係を持ち，メッセージベースの状態（集合としての内容）と現在のプログラムポイントにより，メッセージの送信，やメッセージベースの書き換えなどの次のステップのアトミックな動作を決定する．

2.2 エージェントプログラミング言語（Erdoes）

上記の計算モデルを反映した直接形式的検証可能なプログラミング言語（Erdoes と呼ぶ）を以下に導入する．

[プログラムの構造]

まず，プログラムは複数のサブプログラムからなり，各サブプログラムは if-then ルールの列からなる．サブプログラムの 1 つは実行のエントリを示す main をサブプログラム名として持つ．サブプログラム実行時には原則として先頭の if-then ルールから末尾に向かって順に処理を進める．サブプログラムから別のサブプログラムへの実行の遷移は，下記に説明する *call* アクションにより行われる．

[if-then ルール]

if-then ルールの if 部には，一階述語論理式で，アトム式を and または or の論理結合子で結んだ式が指定される．**メッセージ**は，一階述語論理式のグランドアトム式，すなわち，引数の項の中に変数を含まないアトム式として表される．if 部には，いつも真であることを表す “true” の記述も許される．そして then 以下，および else 以下には，それぞれ複数のアクションが並べられる．アクションの中には，if 部の論理式の変数を含むものもある．ただしその変数は，if 部の論理式を変数の名前の書き換えなしに選言標準形に書き換えたときに，どの選言子にも含まれる変数とする．if-then ルールの実行では，まずその if 部の論理式が現在のメッセージベースの内容から導かれるかをチェックする．ここで if 部の式の変数は存在限定されているものとする．たとえば if 部が if msg(f(?x))（?x は変数）で，現在のメッセージベースに msg(f(g(a，b))) というメッセージがあれば，if 部は導出されたことになり，?x には g(a，b) がバインディングされる．もし if 部の導出に成功したら then 部分のアクションを順に実行し，そうでなければ else 部分のアクションを順に実行する．アクション部分の変数に関しては，if 部の導出が成功したとき，その if 部の式のインスタンシエーションを，アクション部の変数にも適用して，アクションを具体化する．上に述べた，if 部の論理式の選言子にかかわる，アクションが含む変数の条件は，if 部の式が導かれたとき，その導出から，かならずアクション部の変数がインスタンシエートできることを保証するもので，この条件が満たされない場合には，たとえば，if m1(?x) or m2(?y) で，アクション部が?y を含んでいたなら，m1(a) がメッセージベースにあり，if 部が導出されても?y をインスタンシエートできない状況になってしまう．

[アクション]

Erdoes で利用されるアクションは以下のとおりである．

［メッセージベースを操作するアクション］

- *add*(*agt_name*: ϕ)
  エージェントはメッセージ ϕ をエージェント *agt_name* のメッセージベースに加える．このアクションが Erdoes でエージェント間メッセージ通信を行う唯一のアクションである．*agt_name* の部分が空のときは，自分のメッセージベースに ϕ を加えることを意味する．
- *rm*(ϕ)
  エージェントは自分のメッセージベースから，もしメッセージ ϕ があればそれを取り除く．

［実行をコントロールするアクション］

- *call*(*sub_prg_name*)
  エージェントはサブプログラム *sub_prg_name* を呼ぶ．
- *idle*
  エージェントは何もせず，次の実行時も現在実行しているものと同じ if-then ルールを実行する．

［プロセスコントロールのアクション］

- *create*(*new_agt_name*，*sub_prg_name*，*arg*1，...)
  エージェントは名前 *new_agt_name* を持つ新しいエージェントを生成する．このエージェントのプログラムは *sub_prg_name* を *main* とし，その他のサブプログラムはそこから直接，あるいは間接的に呼ばれるものである．また，メッセージベースの初期状態はメッセージ *arg*1，... からなる．
- *stop*(*agt_name*)，*resume*(*agt_name*)，*kill*(*agt_name*)
  これらのアクションはエージェントの動作の停止，再開，エージェントの消滅を行う．

［その他のアクション］

- *go*(*url*)
  url で指定されたホストにエージェント，すなわちそのプログラムと実行状態を移動し，実行を新しいホストで続ける．
- *ex_call*(*method_name*，*arg*1)
  エージェントは外部メソッド *method_name* を引数 *arg*1 で呼ぶ．Erdoes は Java 上で実装されているので，この外部メソッドも基本的には適当なインタフェースを持った Java メソッドとして実装される．このアクションについては 2.3 節で再び詳しく議論する．

2.3 Erdoes の言語の補足説明

エージェントの移動機能とプログラムスタック

現在最も幅広く使われているエージェント開発・実行環境である Jade をはじめとして，多くのエージェント開発・実行システムではエージェントのホスト間の移動機能をサポートしている．Erdoes でも，*go* アクションに見られるように，エージェントのホスト間の移動機能を実現している．この移動では，if-then ルールを単位として，それまでのプログラムの実行状態を記憶し，別のホストに移動後，その実行を継続することができる．

この実行状態は，プログラムスタックで管理される．それは次のように構成される．今，サブプログラム *sub_prg_name* が呼ばれたとき，スタック要素 <*sub_prg_name*, 1> をプログラムスタックに積む．ここで，対の 2 番目の要素 “1” は，最初の if-then ルールを実行することを表している．その後，次の if-then ルールに実行が移ると，この 2 番目の要素は順次 increment される．また，サブプログラムの実行が終了すると通常どおり，対応するスタック要素がスタックの上部から除かれる．このスタックは，形式的検証でもシステムの状態遷移を記述する際の重要な要素となる．

外部メソッド呼び出し

Erdoes では，エージェント間通信を必要としない内部処理は外部メソッド呼び出しのアクション *ex_call* で処理する．すなわち，エージェント間の協調プロトコルの部分のみを Erdoes のプログラミング言語で記述し，その他の部分は Java や C で実装された外部メソッドを呼び出す．ただし，これらのメソッドの呼び出しにおいてその引数は *ex_call* の第 2 引数としてメッセージの形で与えられ，そのメソッドの戻り値もやはりメッセージの形で，そのメソッドを起動したエージェントのメッセージベースに返される．そして外部メソッドの実行が終わるまで，すなわち戻り値がメッセージベースに返されるまで，次の if-then ルールは実行しない．

if-then ルールに含まれる変数の値代入

先にも述べたとおり if 部のテスト式でメッセージの中に変数を使うことができる．Erdoes では変数には “?” を冠頭につけ，定数と区別する．そしてテスト式に含まれる変数を同じ if-then ルールのアクションの記述の中，たとえば *agt_name*, *sub_prg_name*, *new_agt_name* などにも用いることができる．このとき，if-then ルールの実行で，テスト式があるインスタンシエーションで導出可能なとき，その if-then ルール中のアクションの記述に含まれる変数にそのインスタンシエーションに基づく代入をほどこしてアクションを実行する．この代入は実行中の if-then ルールに対してのみ行われる．すなわち，同じ変数が別の現在実行中でない if-then ルールのアクションの中で用いられていても，その変数への代入は行わない．また，複数のインスタンシエーションで導出可能な場合は，そのうちの 1 つが非決定的に選ばれる．この代入を利用して，エージェントは通信するエージェントや呼び出すサブプログラムの名前などを動的に決定することができる．特にエージェント間で，新たに通信する未知のエージェントの名前のやりとりができる．

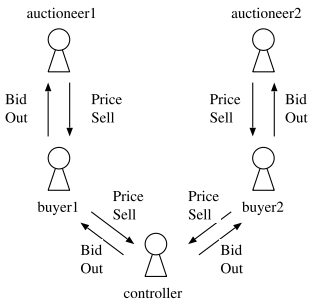

図 1 オークションプロトコルのエージェント構成と交換メッセージ
Fig. 1 Agents and their message exchanges in auction protcol.

2.4 Erdoes 言語のプログラム例

以下に Erdoes 言語のプログラム例を紹介し，その例に基づいて言語の特徴を説明する．

オークション

まず，プログラムのシナリオは以下のとおりである．今，ユーザが欲しい商品を扱うオークションが同時に 2 つ開催されるとし，各々のオークションをとり仕切るエージェントを auctioneer1，auctioneer2 とする．これに対し，ユーザは，**図 1** のように各々のオークションに参加する 2 つのエージェント buyer1，buyer2 と，そのエージェントの動作をコントロールする controller というエージェントを作る．controller は，buyer1，buyer2 から各オークションのせり値の情報をもらい，それらを比べて，安い方から品物をせり落とそうとする．そのプロトコルは，下記のように Erdoes の言語でプログラム記述される．ただし，buyer1 と buyer2 のプログラムの構造は同じなので，buyer1 のプログラムのみを示し，その中の registration サブプログラムの中身の記述も省略した．また，auctioneer1，auctioneer2 は他で独立に作られたエージェントで，相互接続のために，外から見た振舞いが一定の条件を満たすように作られているが，その内部の詳しいプログラ

ムはここでは分からないとする．これに関しては，3.2節でもう一度説明する．

```
/*agt> buyer1 */
sub_p main
{
 if true then call(registration);
 if true then add(auctioneer1: Bid(buyer1, 1000));
 if true then call(bidding);
}

sbu_p registration
{ ...}

sub_p bidding
{
 if Price(?x)
   then add(controller: Price(buyer1, ?x)),
            rm(Price(?x));
 if Sell
   then add(controller: Sell(buyer1)), rm(Sell);
 if Bid(?x)
   then add(auctioneer1: Bid(buyer1, ?x)),
            rm(Bid(?x));
 if Out
   then add(auctioneer1: Out(buyer1)), rm(Out)
   else then call(bidding);
}

/*agt> controller */
sub_p main
{
 if Price(buyer1, ?x) and Price(buyer2, ?y)
   then ex_call(choose Price-choose(?x, ?y));
 if Choose(?au, ?buyer, ?price)
   then add(?buyer: Bid(?price)),
        rm(Price(?buyer, ?x)),
        rm(Choose(?au, ?buyer, ?price));
 if Out
   then add(buyer1: Out), add(buyer2: Out),
            add(: Break);
 if Sell(?b) and Price(?buyer, ?x)
   then add(?buyer: Out), add(: Break);
 if Break then else call(main);
}
```

まず，buyer エージェントは，サブプログラム registration を呼んで，担当するオークションの auctioneer エージェントに登録をし，次に最初のせり値を auctioneer エージェントに伝える（Bid）．そして，せりのルーチン（サブプログラム bidding）に入る．auctioneer エージェントは，集まったせり値から，新しい価格を定め（Price(...)），それを参加している各 buyer に伝える．buyer エージェントは，auctioneer エージェントから受けた値をそのまま自分の名前をつけて controller エージェントに送る（Price(buyer1,...)）．Price(?x) の?x のところには，auctioneer から送られてきた現在の価格がインスタンシエートされる．controller エージェントは buyer1，buyer2 から受け取った値段を外部メソッド Choose を使って比べる．Choose メソッドは，もしオークションから撤退すべきと判断したら Out，そうでなければ次にどちらの buyer にいくらのせり値で競争させるかを決める（Choose(?au, ?buyer, ?price)）ようコーディングされており，その結果をメッセージベースに返す．Out が返ると，buyer1，buyer2 ともにその auctioneer に Out を送り，活動を停止させる．Out でなく，どちらか一方が選ばれたとき，選ばれた buyer に新しいせり値（Bid）を送り，それが auctioneer に中継される．また，選ばれたオークションの現在の価格をメッセージベースから取り除く（rm(Price(?buyer, ?x))）．今回選ばれなかった buyer に対しては，なにもせず，それが担当する auctioneer への返答が引き伸ばされる．また，一方の auctioneer から，落札（Sell）できたら，その情報は buyer を中継して controller に送られ，controller の main サブプログラムの最後の if-then ルールで，もう一方のオークションの buyer に撤退メッセージが送られる．buyer1 の bidding サブプログラム，controller の main サブプログラムともに，最後の if-then ルールで自らを呼んで，ループ構造になっている．

auctioneer は，すべてのオークション参加のエージェントからせり値か Out が返ってくるのを待ち，それらがすべて集まったら次の値段を決めて入札者に送り，他のエージェントが撤退し，入札者が 1 人になったら，落札のメッセージ（Sell）をそのエージェントに送るようにプログラムされているものとする．したがって，この auctioneer のエージェントと接続するためには，buyer は，Price を受信したら，いつかは Bid か Out を返すようにプログラムされていなければならない．上記のエージェントプログラムはこの要求条件を満たしている．一方，買い手にとって，入札はたかだかどちらか一方のオークションからのみ得られることが要求される．すなわち，両方から同じものを 2 つ買いたくない．上記プログラムは，この要求に対して，単純だが見つかりにくいバグがある．これについては，3 章の検証の具体例で述べる．

3. エージェントプログラムの検証

いくつかのエージェントをプログラムしたとき，それらが全体としてプログラマが意図したとおりに動作するかを検証することが重要である．この検証のために Erdoes では，形式的検証の 1 つである時相論理 FCTL（Fair Computational Tree Logic）の symbolic モデルチェックアルゴリズム[6),8)] を利用する．このモデルチェックアルゴリズムを利用するには，システムの全体の動作（状態遷移）を論理的に表現した式

が必要になる．今，システムの状態を表すためのいくつかの命題変数を並べて $(x_1, \ldots, x_n)$（$\vec{x}$ とも書く）としたとき，システムの 1 ステップ実行後の次の状態を表す，$(x_1, \ldots, x_n)$ に対応した変数を $(x'_1, \ldots, x'_n)$ $(= \vec{x'})$ と書くことにする．このとき，上記の論理式は，システムの状態遷移関係（1 ステップ実行前と後のシステムの状態の関係）を，$\vec{x}$ と $\vec{x'}$ の関係で表したブール式 $R(\vec{x}, \vec{x'})$ である．Erdoes では，エージェントプログラムが与えられたとき，このブール式を，後述する外部メソッドに関する少しのマニュアルによる抽象化作業を除いて，自動で導出できる．

以下に，このブール式をどのように導くかを示す．

3.1 エージェントプログラムの状態遷移関係式への基本変換

エージェントプログラムの実行状態を決めるものは各エージェントのプログラムスタックの状態とメッセージベースの状態である．さらに，複数のエージェントが，interleaving モデルに基づいて，非同期に実行される場合，現在どのエージェントが，選択実行されているかを表すことも必要である．Erdoes のエージェントプログラムでは，if 部の論理式は変数を含み，if 部のテストでは，パターンマッチが行われる．しかし，そのパターンにマッチするインスタンスが有限個で，あらかじめすべて分かっている場合は，各々のインスタンスで，if-then ルールをインスタンシエートし，適当にプログラムを変形することで，同じ意味を持ち，変数を含まないプログラムを生成することが可能である．4 章の簡約化手続きでは，このようなインスタンスをすべて求めるので，その手続きの利用を前提として，ここではエージェントプログラムは変数を含まないものとする．また，手続きのポイントを明確にするため，エージェントプログラムにはエージェント生成の *create* アクションは含まないものとする．

[1] テスト式の論理式への変換

エージェント *agt* の if-then ルールのテスト式 ϕ はメッセージをアトム式とするブール式である．このテスト式に含まれる各メッセージ *msg* に対して，状態変数 $agt.[msg]$ を導入する．$agt.msg$ は，*agt* のメッセージベースにメッセージ *msg* が含まれることを表している．そして，ブール式の構造はそのまま保って，メッセージを上記の状態変数に置き換えて，テスト式を状態変数のブール式に変換する．たとえばテスト式 ϕ が $(msg1\ and\ msg2)\ or\ msg3$ の場合，$(agt.msg1 \wedge agt.msg2) \vee agt.msg3$ となる．このブール式を $bf(agt, \phi)$ と書くことにする．

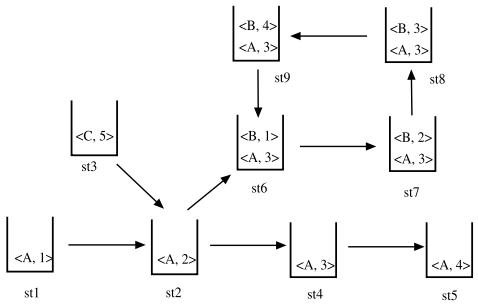

図 2 プログラムスタックの状態遷移
Fig. 2 State transition of program stack.

[2] プログラムスタックの状態遷移関係

エージェント *agt* のプログラムが与えられたとき，プログラムの中のサブプログラム呼び出しをたどることで，可能なプログラムスタックの状態をすべて求めることができる．ここで求められるプログラムスタックの状態数は有限になると仮定する．

一般には，このような仮定は成り立たないが，エージェントのアプリケーションは単純なループ構造を持つため，この制約内でもアプリケーションの記述には問題ないと考える．たとえばループを構成する *call* アクションに対し，それらはすべてサブプログラムの最後の行に出現するという条件を課せば，先の制約は満たされる．求められたプログラムスタックの各状態に対して，各々新しい状態変数を導入する．あるプログラムスタックの状態 *p_stack* に状態変数 *st* が対応したとすると，*st* が真であることは，エージェントの現在のスタックの状態が *p_stack* であることを表す（**図 2**）．また，$agt.[st]$ をエージェント *agt* の現在のプログラムスタックの状態が *st* であることを表す命題変数とする．

この変数を用いて，エージェント *agt* のプログラムの状態遷移関係を以下のように記述する．今，スタック状態変数 *st* に対応するスタックで実行中の if-then ルールのアクションが，サブプログラム呼び出し *call* アクション，あるいは *idle* アクションを含む場合，たとえば if ϕ then $call(\ldots)$ の形の場合，そのサブプログラムが呼ばれた場合の遷移後のプログラムスタックの状態に対応する状態変数が st_1，呼ばれなかった場合のプログラムスタックの状態に対応する状態変数が st_2 であったとき，新しく記号 $\Rightarrow$ を導入して，この状態遷移の関係を各々以下のように表すことにする．$bf(agt, \phi) \wedge agt.[st] \Rightarrow agt.[st_1]'$，$\neg bf(agt, \phi) \wedge agt.[st] \Rightarrow agt.[st_2]'$．if-then ルールが *call* アクションを含まない場合は，遷移後のスタックの状態を表す状態変数が st_1 であったとすると，この状態遷

移関係を $agt.[st] \Rightarrow agt.[st_1]'$ で表す．ここで定義した $A \Rightarrow B$ の形の式を以後，遷移関係式と呼び，B をその結果式と呼ぶ．

[3] メッセージベースの状態遷移関係

メッセージベースの状態の遷移は add と rm のアクションによって引き起こされる．

エージェント agt のプログラムスタックの状態 st には，現在実行中の if-then ルールが 1 つ対応する（スタックの最上部）．

st で実行中の if-then ルールのテスト式を ψ とし，その $then$ 以下に $add(agt_1{:}\ m)$ が現れていたとする．このとき，この add アクションによるメッセージベースの変化を，遷移関係式 $agt.[st] \wedge bf(agt,\ \psi) \Rightarrow agt_1.m'$ で表す．もし，add アクションが $else$ 以下に出現する場合，その変化を $agt.[st] \wedge \neg bf(agt,\ \psi) \Rightarrow agt_1.m'$ で表す．

もし，rm アクション $rm(m)$ が $then$ 以下に現れていたなら，このアクションによる変化を $agt.[st] \wedge br(agt,\ \psi) \Rightarrow \neg agt.m'$ で表す．$else$ 以下の場合も同様に表す．

[4] 非同期性

エージェントの interleaving モデルによる非同期性に関して，各エージェント agt_k に対し，命題変数 $agt_k.run$ を導入する．$agt_k.run$ は agt_k が現在 if-then ルールを実行中であることを表す．agt_k のプログラムから［2］, ［3］で導かれた遷移関係式 $A \Rightarrow B$ に対し，これを $agt_k.run \wedge A \Rightarrow B$ と書き換える．

[5] システム全体の遷移関係

これまでに得られた遷移関係式から，システム全体の遷移関係を表す論理式を作る．

まず，$A_1 \Rightarrow B$，...，$A_n \Rightarrow B$ を同じ結果式 B を持つ遷移関係式とする．このとき，これらを 1 つの遷移関係式 $A_1 \vee \ldots \vee A_n \Rightarrow B$ に変換する．そして，この変換をすべての結果式に対して行う．次に，メッセージ m のメッセージベースでの出現に関わる遷移関係式，すなわち $A1 \Rightarrow agt.m'$ の形のものと $A2 \Rightarrow \neg\ agt.m'$ の形のものから論理式 $agt.m' \rightarrow A1 \vee (\neg agt.run \wedge agt.msg) \vee (agt.run \wedge agt.msg \wedge \neg A2)$ を生成する．エージェント agt のプログラムスタックの状態変化に関わる遷移関係式，すなわち $A \Rightarrow agt.[st]'$ から論理式 $agt.[st]' \rightarrow A \vee (\neg agt.run \wedge agt.[st])$ を生成する．また，interleaving モデル l に基づく非同期性の表現のために，論理式 $\bigwedge_{i<j} \neg\ (agt_i.run \wedge agt_j.run)$ および $\bigvee_i agt_i.run$ を導入する．最後に，これまで導いたブール式すべての連言をとり，CTL のモデルチェックアルゴリズムに必要なシステムの状態遷移を表す論理式を得る．

［注］上記の手続きでは，議論を簡単にするために，プログラムは $create$ や，$kill$ などのプロセス制御のアクションは含まないものとした．不定個の，しかもどのようなプログラムが生成されるか分からない状況では，これらのアクションをここでの検証方法で取り扱うことはできない．しかし，途中で生成される可能性のあるプロセスの個数，名前，プログラムがあらかじめ分かっている場合は，上記の変換を，以下のように少し拡張するだけで，これらのアクションを扱うことができる．たとえばエージェント agt の $create$ を表すには，$agt.run$ と同様のアイデアで，$agt.alive$ という変数を導入する．この変数を使って，agt を $create$ するアクションが実行されたとき，次の状態で $agt.alive$ は true になり，agt を $kill$ するアクションが実行されるまで，true でありつづけるということを表す論理式を，上記の変換と同じ考え方で記述すればよい．さらに，run との関係で，$agt.run \rightarrow agt.alive$ という論理式も付け加える．$stop$ や $resume$ を扱う場合は，さらに同様の変数を導入して，そのアクションが実行される条件と run，$alive$ との関係に注意して同様の論理式を記述すればよい．

go アクションについては，Erdoes の計算モデルではエージェント間の通信に影響を与えないので検証にも関係しない．ただ，環境のモデル化（3.2 節）の仕方によっては，今後，このアクションが検証に影響を与えることも考えられる．

3.2 環境とのインタラクションと外部メソッドの抽象化

環境とのインタラクション：

3.1 節で，CTL のモデルチェックを適用するために，与えられたエージェントプログラムを，その状態遷移関係を表すブール式に変換する方法を示した．プログラマが，互いに通信・協調する一群のエージェントのプログラムを作成したとき，その中のいくつかのエージェントは，外部のエージェントと通信して動作することを前提にすることが多い．これらの外側のエージェントは，Erdoes の検証では環境の 1 つと考え，自分の作ったエージェントを検証するとき，その外部のエージェントは，外から見た振舞いのみを同じく Erdoes プログラムで記述し，自分の作ったエージェントとあわせて検証を行う．このように，検証を目的に，外部のエージェントの動作を記述するときは，以下の例に示すような非決定的なアクション選択のオペレータ（“|”）を使うことが許される．

たとえば，2.4 節のオークションの例では，auctioneer1，auctioneer2 は外部のエージェントで，その振

舞いは，下記のように記述される．

```
/*agt> auctioneer1 */
sub_p main
{
  if Break then else call(price);
 if Break then else call(main);
}

sub_p price
{
 if Bid then add(buyer1: Price), rm(Bid)
             | add(buyer1: Sell), add(:Break);
}
```

ここで記述されている，auctioneer1 の外から見た振舞いは，Sell を一度 buyer1 に送ったら，処理を終了し，それまでは，Bid メッセージが届くごとに，buyer1 へ，Price の送信か，Sell の送信を非決定的に選択する．

外部メソッドの抽象化：

CTL のモデルチェックアルゴリズムは基本的に，有限状態遷移関係にのみ適用される．

一方，エージェントプログラムでは，主に 2 つの事情から，システムは無限個の状態をとりうる．1 つは，実行状態が無限になる場合で，Erdoes の場合でいうと，プログラムスタックの深さが無限になりうる場合である．もう 1 つは，エージェントの扱うデータの大きさが無限になる場合であり，自然数や，集合，リストを扱っている場合，このようなことが起こる．

メッセージベースの大きさに関しては，扱うデータの種類が有限個になったら，メッセージのタイプも有限個なので，メッセージの種類も有限個になり，メッセージベースは queue でなく，普通の集合（多重集合ではない）なので，そのサイズも有限で抑えられる．

プログラムスタックの深さについて，Erdoes の検証では，3.1 節［2］のプログラムスタックの静的解析で，その遷移が有限にならないプログラムは扱わないことにしている．インターネットのエージェント通信では，そのプロトコルは周期的な単純なものがほとんどで，この条件はエージェントプログラムにとって大きな制約にはならないと考えている．

一方，インターネットエージェントでも，無限の状態をとりうるデータ構造を扱うことは普通にある．

Erdoes では，このようなデータの操作は *ex_call* のアクションを通して行う．そして，検証において *ex_call* の動作は black box と見なされる．すなわち，環境の場合と同様，*ex_call* の動きは，エージェントのプロトコルに影響を与える範囲でその入力と出力の関係が記述される．たとえば，先にあげたオークションの例では，外部メソッドの呼び出しの部分

```
if Price(buyer1, ?x) and Price(buyer2, ?y)
   then ex_call(choose Price-choose(?x, ?y));
```

は，

```
if Price(buyer1) and Price(buyer2)
   then add(:Choose(buyer1))
          | add(:Choose(buyer2)) | add(:Out);
```

と書き換えられる．ここで，?x や?y は，その値がプロトコルの検証に影響しないと判断されたため，定義域が一点化されメッセージから除かれた．この抽象化は，もちろんプログラムの他の部分にも反映される．すなわち，対応する?x や?y が同様にメッセージから除かれる．もし，たとえば，?x の値が 1000 以下とそれより多い場合でエージェントの振舞いが変わるとき，整数を想定している?x の定義域は，1000 を境に 2 つに分割され，それに対応して，2 種類のメッセージを使って動作を記述することになる．

また，この外部メソッドの可能な 3 つのメッセージ出力，Choose(buyer1)，Choose(buyer2)，Out に対しても，どれが出力されるかの判断は，ここでは非決定的に行われるものと単純化している．

現時点では，上記のような抽象化はプログラマの人手により行われ，自動化はなされていない．この課題については，5.1 節で再び議論する．

3.3 変換の例

2.4 節で導入した例を用いて，エージェントプログラムから状態遷移の関係を表すブール式への変換の具体例を示す．下記に，変換結果の断片を示す．&は「かつ」，| は「または」，～ は「否定」，− > は「ならば」を表す．

```
[buyer1]
     n_buyer1.Price -> ....
      :
(1) n_buyer1.Bid ->
     controller.run & c_controller.Choose(buyer1)
    & c_controller.[m2]
    | ~buyer1.run & c_buyer.Bid
    | buyer1.run & c_buyer.Bid & ~c_buyer.[m3-b3]
     :

     :
    n_buyer1.[m3] ->
    buyer1.run & c_buyer1.[m2]
    | ~buyer1.run & c_buyer1.[m3]

(2) n_buyer1.[m3-b1] ->
     buyer1.run & c_buyer1.[m3]
    | buyer1.run & c_buyer1.Out & c_buyer1.[m3-b4]
    |~buyer1.run & c_buyer1.[m3]
       :

[controller]
     n_controller.Price(buyer1) -> ....
       :
(3) n_controller.Choose(buyer1) ->
     controller.run & c_controller.Price(buyer1)
```

```
      & c_controller.Price(buyer2) & c_controler[m1]
    | ~controller.run & c_controller.Choose(buyer1)
    | controller.run & c_controller.Choose(buyer1)
       & ~c_controller.[m2]

(3)' c_controller.Choose(buyer1)
      & c_controller.Choose(buyer2)  -> false
    c_controller.Choose(buyer1)
      & c_controller.Out -> false
    c_controller.Choose(buyer2)
      & c_controller.Out -> false

     n_controller.[m1]-> ....
       :
```

4 章で述べる簡約化も利用して，必要な変数の数は，現在と次の状態の両方を数えて，69 個である．これは，1 ギガ程度のメモリを利用した symbolic モデルチェックのツールで扱える状態変数の数の上限に近い．導出された論理式は，先に説明したように，基本的には，メッセージベースの変化の部分と，スタックの変化の部分に分かれる．これらのコードで，n_x は，状態変数 x の次の状態，c_x は，x の現在の状態を表す変数である．

(1) は，buyer1 のメッセージベースの変化を表す式で，次の状態 (n_) で，buyer1 がメッセージ Bid をメッセージベースに持つための条件を示している．それは，今の状態で，controller が選択実行されていて（controller.run），スタックが，Bid を送るプログラムポイントにあり（c_controller.[m2]），その if-then ルールの then の部分が実行されるように Choose(buyer1) の形のメッセージが現在のメッセージベースにあるか，buyer1 が現在実行されていなくて（~buyer1.run），buyer1 が Choose(buyer1) をすでにメッセージベースに持っているか，現在実行されていて（buyer1.run），Choose(buyer1,*) のメッセージを保持していて，しかも現在の実行で，そのメッセージが if-then ルールの実行で除かれることがないことを表している．

(2) は，buyer1 のスタックが次の状態で [m3-b1]，すなわち main サブプログラムの 3 番目の if-then ルールで，bidding サブプログラムが呼ばれ，その最初の if-then ルールの実行を行う状態になるための，現在の状態の条件が書かれている．すなわち，現在，buyer1 に実行が与えられていて，main サブプログラムの 3 番目の if-then ルールを実行中か，あるいは bidding サブプログラムの 4 番目の if-then ルールが実行中で，この bidding サブプログラムをそのポイントで呼ぶ（call）ためのメッセージベースの条件が満足されているか，あるいは，現在，buyer1 プログラムは実行されていず，そのスタックが [m3-b1] の状態にある場合を表している．

(3) は，外部メソッドの非決定的な動作を表している．メッセージ Choose(buyer1,...) は，外部メソッド choose の可能な戻り値の 1 つである．(1) と同様，実行中で，新たにそのメッセージが加えられる場合，実行していなくて，すでにそのメッセージがある場合，実行中で，すでにそのメッセージがあり，そのメッセージを削除するアクションが起動されない場合に分かれる．そして，新たにメッセージが加えられる場合として，外部メソッド choose が起動される条件が書かれている．ただし，choose は，可能な 3 つの戻り値のうちどれを返すかが分からないので，その非決定的，排他的条件が (3)' で示されている．

3.4 検 証 例

2 章で触れたように，例題のオークションプログラムは，「2 つのオークションからともに落札を得ることはない」という要求仕様

AG(~(c_controller.Sell(buyer1)
& c_controller.Sell(buyer2)))

が満たされない．例外となるケースは，2 つのオークションに最初に入札したとき，たまたま他のどの buyer も入札を行わず，両方のオークションにおいて最初の段階で落札者になってしまう場合である．このようなバグは単純で，形式的検証を行うまでもないと感じられるかもしれないが，実際のエージェントプログラムは，複雑な分散アルゴリズムとは異なり，そのプロトコルは，単純で分かりやすく，バグの原因になるのは，この例のような単純なミスによるものが多い．しかし，その単純なミスもいろいろなパターンがあり，組織的につぶしていくのは難しい．また，上記の例は，実際の実行テストでは露見する可能性が低く，形式的検証の効果は大きい．

上記の問題を我々の実装した symbolic モデルチェックシステムで検証した．このシステムは論理式の表現として，否定枝つきの BDD を利用し，通常の BDD の実装どおり，hash と cache を利用して高速化を行っており，Java で実装されている．実行効率，使用メモリサイズは BDD の変数順序にもよるが，状態遷移の論理表現に，約 3 万の BDD ノードが生成され，検証は，BDD の and-exist 演算を 12 回行うことで完了した．実行時間は，512 M バイトのメモリを有する 1.10 GHz Pentium(R)M windows マシンで約 10 分だった．この実行で約 20 万ノードが，BDD ノードのガベージコレクション機能を入れた状態で生成された．この結果は，実際の利用に耐えうる範囲であると考える．

4. 検証の簡約化

エージェントシステムはメッセージパッシングに基づく非同期分散システムであるが，各エージェントは自律性を考慮してプログラムされるため，エージェント間の結びつきは比較的疎である．したがって要求仕様が与えられて，その検証を行うとき，その仕様の検証に影響を与えるエージェントプログラムの部分は全体の一部であり，そのプログラムの部分も要求仕様によって変わる．Erdoes では，その言語のデザインを生かして，このような，検証に影響を与えるプログラムの部分だけを自動的に抽出するプログラムの簡約化が効率的にできる．以下にその手続きを示す．

4.1 簡約化手続き

インターネットエージェントシステムに対する要求仕様は,「いつかはエージェント *agt* は，メッセージ *m* を受け取る」や「エージェント *agt* は，メッセージ *m*1 と *m*2 の両方を受け取ることはない」などメッセージ受信に関するものと,「エージェント *agt* は，いつかはサブプログラム *sp* を実行する」というプログラム実行に関するものがある．本論文の簡約化手続きでは，後者は，前者と本質的に同じように処理できるので，ここでは前者のメッセージ受信に関する要求仕様についてのみ説明する．

煩雑さをさけるために，こまかな手続きは省略し，簡約化手続きの本質的な部分のみを以下に説明する．簡約化手続きでは，プログラムの中の if-then ルールで，与えられた仕様に影響を与えるものにチェックマークがつけられる．

(1) メッセージの追跡

メッセージ受信に関する要求仕様に対し，その簡約化手続きでは，プログラムの中で，メッセージ送受信の関係を追う．すなわち，あるエージェントのあるメッセージの受信が問題になったとき，そのメッセージをそのエージェントに送信する可能性のあるエージェントをプログラム中，*add* アクションをもとに，メッセージおよび受信エージェント名のパターンマッチで探す．そして，マッチする if-then ルールが見つかったなら，それにチェックマークをつけ，そのルールの if 部のメッセージに対しても，同様の追跡を行う．if 部がメッセージの連言の場合，そのすべてのメッセージを追跡する．

(2) *call* の追跡

(1) で，受信メッセージにマッチする送信側の if-then ルールを見つけたが，その *add* アクションが実行されるには，if 部分が成功するだけでなく，その if-then ルール自身が実行されなければならない．そのためには，その if-then ルールを含むサブプログラムが main サブプログラムから直接または間接的に呼ばれなければならない．そして，その *call* アクションのつながりを実現するためには，その連鎖に含まれる *call* アクションを含む if-then ルール（これらにもチェックマークをつける）の if 部のメッセージが受理されなければならず，ここで再帰的に (1) の処理が起動される．

(3) ループからの復帰

(1) で見つかったマッチする if-then ルールが実行されるには，それを含むサブプログラムが呼ばれるだけではまだ不十分である．そのサブプログラムの先頭の if-then ルールから，対象としている if-then ルールまでの間に *call* アクションを含む if-then ルールがあり，その呼び出しを順方向にたどったとき，そこにループ構造があった場合，そのループから復帰しないと，対象としている if-then ルールへ実行が移らない．したがって，このループ構造を形成する *call* アクションを含む if-then ルール（これらにもチェックマークをつける）の if 部のメッセージに対しても，(1) の処理が起動される．

(4) 要求仕様に影響するプログラム部分の探索の流れ

最初にメッセージ受信に関する要求仕様が与えられたとき，その中に含まれる各メッセージに関して，(1) の処理を始める．その処理の中で，(1)，(2)，(3) の処理が再帰的に呼び出され，プログラム中の if-then ルールの中で，与えられた要求仕様に影響するものの探索が，メッセージの流れの逆向きの追跡により行われる．

このとき (1) や (2) ではマッチングを行うので，変数の unification 情報を保持して，各マッチングでは，これまでの unification の制約のもとでマッチするか（unifiable か）をチェックする．

マッチする if-then ルールが複数ある場合は，探索は分岐する．すなわち，別の探索スレッドを生成する．(2)，(3) の *call* アクションでも，その中で変数が使われていて，マッチする *call* アクションを行う if-then ルールが複数ある場合は分岐する．また if-then ルールのテスト式を選言標準形に直したとき，複数の選言子があった場合，その選言子ごとに探索はそこで分岐し，この場合も別のスレッドを生成する．

(1) で if-then ルールをたどったとき，その if 部がブール式 true であったら，そのメッセージの追跡はそこで終わる．すべての追跡が終わったら，その探索スレッドは成功して終了する．探索で，マッチする if-

then ルールが見つからない追跡が 1 つでもあったら，その探索スレッドはその時点で失敗する．

(5) 要求仕様に影響するプログラム部分の出力

すべての探索スレッドが終了したとき，成功した各探索スレッドに対し，そこに含まれる unification 情報で，エージェントプログラムをインスタンシエートし，その探索スレッドで，与えられえた要求仕様と関係があるとチェックされた if-then ルール以外はすべて取り除く．成功したすべての探索スレッドに対し，上記の処理結果を合併して最終的な結果とする．

この結果が，3 章の遷移関係論理式の自動合成手続きの入力となる．ここで，1 つのプログラムポイントの if-then ルールに複数のインスタンシエーションが存在しうることに注意する．

要求仕様が与えられたとき，システムが要求仕様を満たすことと，簡約化されたシステムが，3 章の検証手続きで，要求仕様を満たすことが同値になる．

4.2 簡約化の例とその効果

2.4 節で与えたオークションの例をもとに簡約化の具体的な動作を説明する．今，3.4 節で述べた要求仕様

AG(∼(c_controller.Sell(buyer1)
& c_controller.Sell(buyer2)))

を考えたとき，まず，controller.Sell(buyer1) からそのメッセージの送信アクションである buyer1 の bidding サブプログラムの 2 番目の if-then ルールが関連し，その発火条件となる Sell メッセージから，3.2 節の auctioneer1 の動作記述の price サブプログラムの if-then ルールが関連し，さらにその発火条件となる Bid メッセージから buyer1 の bidding サブプログラムの 3 番目の if-then ルールが関連する．そして，メッセージの追跡は Bid メッセージを buyer1 に送信する controller の 2 番目の if-then ルールに移る．一方，上記の関連する if-then ルールを含む buyer1 の bidding サブプログラムについて，その call に関係するのが main サブプログラムの 3 番目の if-then ルールと bidding サブプログラムの 4 番目の if-then ルールである．前者の発火条件は true なので，その追跡はここで終わる．後者の発火条件は Out なので，その追跡は controller の main サブプログラムの 3 番目と 4 番目の if-then ルールに追跡が分かれる．上記の操作を続けると，最終的に，例のオークションプログラムでは，buyer の main サブプログラムの最初の if-then ルールと，その registration サブプログラムの全体が検証の対象から除外される．また，controller の main サブプログラムの 4 番目の if-then ルールの?b は，buyer1 と buyer2 の 2 つのインスタンシエーションが生成される．

表 1 簡約化の効果
Table 1 Effect of program slicing.

	trial1	trial2	trial3	trial4	trial5
system1	32	30	30	34	35
system2	29	35	34	32	36
system3	27	31	31	23	29
system4	34	33	34	36	35
system5	30	30	33	31	31

簡約化の効果を一般的に確かめるため，下記の実験を行った．各々4 つの if-then ルールを持つサブプログラムを 5 つ持つ，5 つのエージェントを用意した．そして，そのサブプログラム呼び出しに対して，ループの有無や，呼び出しの深さなどの観点から，互いに違ったプログラム構造をエージェントに与えた．次に，エージェント間のメッセージの送受信の関係をランダムに作り，5 つのパターンのエージェントシステム system1, . . ., system5 を作った．この各々のプログラムに対して，任意にエージェントとそれが受信する可能性のあるメッセージを 5 パターン（trial1, . . ., trial5）選び，各受信メッセージ（1 個）に対して，それぞれのプログラムに簡約化手続きを実行した．**表 1** に，簡約化の結果，残った if-then ルールの数を示している．簡約化の前は，全部で 100 個の if-then ルールがあった．状態変数の数に応じて，指数オーダ的に状態数の増える形式的検証では，このプログラムの簡約の効果は大きい．

5. 議論と関連研究

5.1 議　　論

3 章で触れたように，実際の検証用の論理式の生成においては，すべてがプログラムから自動で行われるわけではなく，*ex_call* アクションで起動される外部メソッドの挙動に対しては，プログラマが，どのような計算結果がメッセージの形で出力されうるかを記述しなければならない．この外部メソッドの挙動の記述をなるべく自動化することが今後の大きな課題の 1 つである．

しかし一方，ここがこの言語のポイントでもある．すなわち，エージェントが内部で行う単独の処理（外部メソッド）と，メッセージ交換によるエージェント間のインタラクション部分を分離し，検証の対象をインタラクションの部分にしぼることが本論文のユニークなアイデアである．

そして，エージェントが互いに疎結合であるという

状況，すなわち，特定のパターンのメッセージをもとに互いの行動を制御し，またメッセージベースも queue でなく，集合で十分であることを利用して，インタラクション部分の遷移の有限化がなされ，論理式生成の自動化が実現されている．

上記，外部メソッドの抽象化に関して，Java などのプログラムを検証するための抽象化手法が多く研究されている．なかでも文献 5) では，さまざまな抽象化手法を統合して，実際に，一般のプログラマを対象に，プログラムの抽象化をサポートするツールが提案されている．そこでは，プログラムで用いられるデータと，そのデータ上のオペレーションの抽象化を記述する枠組みが提案され，複数の抽象化を用意したとき，それらを整合的に利用するためのナビゲーション方法も与えている．本論文では，外部メソッドの抽象化に対して，そのメソッドに対するメッセージの入力と出力を論理式の形で書くことを要求し，その論理式をどのように作成するかの指針・手法は現段階では提案できていない．そこで，これらの，ツールを外部メソッドの抽象化に利用することは，魅力的なアプローチになる．ただ，その手法を有効に利用するには，外部メソッドの抽象化にとどまらず，それとインタラクションするエージェントプログラムと一体で抽象化することが必要になる．そして，プログラム言語としてある種の制限を持ったエージェントプログラムの特性を，この抽象化にどのように反映させるかが今後の重要な検討課題である．一方，文献 5) では抽象化の一環として，プログラムの slicing 手法も提案している．これは，本論文の簡約化の手続きと共通するところが多い．そこでは，データの依存関係と，プログラムコントロールの依存関係をもとに，検証対象となる性質と関係するプログラムの部分を計算していく．本論文では，その考えを，提案したエージェントプログラムに特化して精密に実現している．つまり，データの依存関係はメッセージの送受信の関係をもとに，ユニフィケーション情報を含むパターンマッチで計算し，またプログラムのコントロールの依存関係は，call アクションの関係を通して解析する方法を提案した．本論文の方式では，この slicing と同時に，変数の代入によるシステム状態の有限化を行っている．ただし，一般にこの操作の停止性は保証されておらず，どのような有効なクラスでその停止性が保証されるかを調べることが今後の課題となる．

5.2 関 連 研 究

形式的検証のアプローチとしては，本論文で採用した時相論理のモデルチェック以外に，双模倣[9] やローカルモデルチェック[12] など CCS，π 計算などのプロセス代数に基づく手法もある．ただ，本文でも述べたように，これらの計算モデルはハンドシェイクをベースとしており，エージェントのオープン性，自律性を反映したモデルとは親和性が低く，エージェントの動作の記述が困難である．

本論文で対象としたインターネットエージェント開発・実行システムは Jade，Aglets などのモバイルエージェント系のシステムである．一方，BDI アーキテクチャと呼ばれる，エージェントの信念（Belief），目的（Desire），目的を達成するためのサブゴール・意図（Intention）をエージェントの内部状態の表現に用い，その動作をプラン生成の枠組みで記述するいわゆる知的なエージェントのシステム[11] があり，このようなシステムに対する形式的検証の研究もさかんに行われている[3),10)]．なかでも Wooldridge ら[4] は，本論文と同様に，実行可能な知的なエージェントシステムから直接形式的検証を行う研究を，我々の研究[1),2)] とは独立に進めている．彼らは，agentSpeak(L) という実行可能な BDI アーキテクチャのプログラム言語を対象に，そこから形式的検証ツール SPIN[7] の入力となるシステム動作記述を直接導く方法を提案している．本論文の研究との違いは，彼らのシステムでは，最初からプログラム言語で変数の利用を排除してシステムを有限状態化しているのに対し，我々の言語では，検証時に，マニュアルによる抽象化や，簡約化で，変数を消去する方法をとり，もともとのプログラム言語の表現力をなるべく強く保とうとしていることである．

また，彼らのプログラム変換の出力は，explicit モデルチェックツール SPIN のシステム記述言語 Promela によるエージェントの動作の記述であるが，Promela 自体は分散システムを記述する言語で，エージェントの動作の変換自体は，データ表現に多少の工夫が必要なものの，直接的に行える．むしろ，彼らの変換のポイントは，Belief，Intention，Desire のエージェントの心的状態表現の Promela による表現にある．一方，我々の検証では，explicit モデルチェックより効率が良いといわれている symbolic モデルチェクの利用を前提に，その入力となる論理式を直接導いている．symbolic モデルチェックでは，SMV のような検証ツールがあり，システム記述言語も提供されているが，ハードウェアの検証を目的に発展した言語であるため，分散システムの動作の柔軟な表現が難しい．そのような理由により，我々は直接論理式を導出するアプローチをとった．

Erdoes は，現在 Java 上のインタプリタの形で実

装され，その上で，エージェントによるグループスケジュールの自動調整や，知的なファイル共有システムなどのサンプルアプリケーションも開発している．また，エージェントのモバイルや，ネームサーバの機能も備えており，さらにエージェントの国際標準であるFIPA 仕様も満たし，Jade などのデファクトスタンダードなエージェント開発システムで作られたエージェントと相互接続が可能になっている．

謝辞 Erdoes のデザインなどについていろいろと議論していただき，また有益なコメントをいただいたATR 知能ロボティクス研究所桑原和宏氏に感謝します．また，論文の構成，理論展開などに対し，有益なコメントをいただいた京都大学情報学研究科佐藤雅彦教授に感謝いたします．

参 考 文 献

1) Araragi, T. and Kogure, K.: Dynamic Downloading of Communications Protocols Using a Logic Based Agent System, *Proc. CL-2000 Workshop on Computational Logic in Multi-Agent Systems*, pp.27–34 (2000).
2) Araragi, T., Attie, P., Keidar, I., Kogure, K., Luchangco, V., Lynch, N. and Mano, K.: On Formal Modeling of Agent Computations, *Proc. 1st Goddard Workshop on Formal Approaches to Agent-Based Systems*, pp.48–62 (2000).
3) Benerecetti, M., Giunchiglia, F. and Serafini, L.: A Model Checking Algorithm for Multi-agent Systems, Intelligent Agent Systems V, *LNAI*, Vol.1555, pp.163–176, Springer (1998).
4) Bordini, R.H., Fisher, M., Pardavila, C. and Wooldridge, M.: Model checking agentspeak, *Proc. AAMAS 2003*, pp.409–416 (2003).
5) Dwyer, M., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu Robby, C., Visser, W. and Zheng, H.: Tool-supported Program Abstraction for Finite-state Verification, *Proc. ICSE 2001*, pp.177–187 (2001).
6) Emerson, E.A.: Temporal and Modal Logic, *Handbook of Theoretical Computer Science*, pp.995–1072, The MIT Press/Elsevier (1990).
7) Holzmann, G.J.: The Model Checker SPIN, *IEEE Trans. Softw. Eng.*, Vol.23, No.5, pp.279–295 (1997).
8) McMillan, K.L.: *Symbolic Model Checking*, Kluwer Academic Publishers (1993).
9) Pistore, M. and Sangiorgi, D.: A Partition Refinement Algorithm for the pi-Calculus, *CAV*, LNCS, Vol.1102, pp.38–49, Springer (1996).
10) van der Meyden, R. and Shilov, N.: Model Checking Knowledge and Time in Systems with Perfect Recall, FSTTCS, *LNAI*, Vol.1738, pp.432–445, Springer (1999).
11) Shoham, Y.: Agent Oriented Programming, *Artificial Intelligence*, Vol.60, No.1, pp.51–92 (1993).
12) Stirling, C.: Modal and Temporal Logics for Processes, *Logics for Concurrency*, LNCS, Vol.1043, pp.149–237, Springer (1996).

(平成 17 年 8 月 23 日受付)
(平成 17 年 10 月 12 日再受付)
(平成 17 年 11 月 16 日採録)

櫟 肅之

1985 年東京大学理学部数学科卒業．1987 年同大学院修士課程修了．同年 NTT 情報通信処理研究所入社．1993〜1994 年ユトレヒト大学客員研究員．現在，NTT コミュニケーション科学基礎研究所勤務．様相論理，高階カテゴリー論理等の非標準論理，分散システムの形式的検証，モバイルエージェントに興味を持つ．人工知能学会会員．