契約による設計を用いたインタラクションの実装

土肥拓生[†] 本位田真一[†,††]

マルチエージェントシステムにおいて，インタラクションは重要な要素の 1 つである．しかし，従来のプログラミング言語では，エージェント間を横断する記述ができなかったため，インタラクションの開発は非常に困難であった．そこで，我々はすでにこの問題を解消するために，インタラクションの開発を容易にするインタラクション記述言語 IOM/T を提案した．一方，現在のマルチエージェントシステムの開発では，テスト/デバッグといった側面をうまく扱うことが難しかった．そこで，本論文では，マルチエージェントシステムのテスト/デバッグを容易にするために，IOM/T に対する「契約による設計」の概念を用いた拡張について述べる．

Implementation of Interaction using Design by Contract

Takuo Doi[†] and Shinichi Honiden[†,††]

Interaction is an important feature in multi-agent systems. However, It is hard to develop interactions since existing programing languages are not able to deal with the notation among several agents. We have proposed a new interaction description language IOM/T which enable us to implement interactions easily. On the other hand, It is hard to test and debug the system in the current multi-agent system development. In this paper, we show the extension of IOM/T with "Design by Contract" in order that we can easily test and debug the multi-agent systems.

1. はじめに

近年，ネットワークの高速化がすすみ分散システムが容易に利用できる環境が整いつつある．その一方で，多くの人々がコンピュータを利用するようになり，ソフトウェアに対して柔軟性も求められるようになってきている．そのようなシステムの構築に対して，マルチエージェントシステムは，有望な手法の 1 つである．しかしながら，マルチエージェントシステムにはそれ特有の難しさがあるのも事実である．それは，Java に代表されるオブジェクト指向プログラミング言語によってマルチエージェントシステムが実装されることに起因する．マルチエージェントシステムにおいて，エージェント間のインタラクションは重要な要素の 1 つである．オブジェクト指向プログラミング言語においては，エージェントはオブジェクトとして表現される．しかしながら，インタラクションはエージェントを横断する概念であるにもかかわらず，オブジェクト指向プログラミング言語では，オブジェクトを横断する記述ができないのである．そのため，概念上非常に関係の深い要素が分離して実装されることになり，インタラクションの開発は非常に困難であった．我々はこの問題を解消するために，インタラクションに着目したマルチエージェントシステムの開発を目指している．そして，エージェント間を横断するインタラクションの開発を容易にするインタラクション記述言語 IOM/T [4] を提案した．

ところで，マルチエージェントシステムは有効な手法でありながら，実用システムにおいて利用されていないのも現状である．その最も大きな理由の 1 つは，マルチエージェントシステム，さらには，エージェントが柔軟であるために，正確性，信頼性の確保といった点において，まだまだ未成熟であるからであると思われる．そのような課題を解決するためには，マルチエージェントシステムの開発において，テストやデバッグが容易に行えなければならない．

そこで，我々は IOM/T に対して，「契約による設計（Design by Contract）」の概念を導入した．IOM/T によるインタラクションの記述に対し，契約による設計を付加することにより，実装時に責任が明確になるとともに，テスト時にその設計をチェックすることにより，不具合の早期発見につながる．本論文では，こ

† 東京大学
The University of Tokyo
†† 国立情報学研究所
National Institute of Informatics

の拡張について述べるとともに，マルチエージェントシステムの動作時にいかにその契約をチェックするかを示す．

以降，2 章でインタラクション記述言語 IOM/T の概要について述べ，3 章で IOM/T に対する契約による設計の拡張について述べる．4 章では，この契約による設計の実現システムの実装について示す．その後，5 章で関連研究を交じえて本手法に対する考察を与え，6 章でまとめを述べる．

2. IOM/T

本章では，インタラクション記述言語 IOM/T [4] の概要について述べる．

従来のマルチエージェントシステムの開発においては，インタラクションの設計と実装の間に大きな隔たりが存在している．設計フェーズにおいては，インタラクションは固有の概念としてとらえられ，AUML [5] のシーケンス図[7] に代表されるようにインタラクションの記述が存在する．それに対し，実装フェーズにおいては，インタラクションは各ロールごとに分割され，それぞれのロールの機能として実装されていた．そのため，インタラクションの設計とインタラクションの実装とはまったく異なる構造となり，それらの対応関係を把握することが困難となっている．また，実装が複数のロールの機能として実装されるため，1 つのインタラクションにかかわる要素でありながらも，複数のロールの実装の中へと分散してしまう．その結果として，設計においては容易に把握できるインタラクションのプロトコルも，実装においては非常に煩雑な表現となってしまう．このように従来のインタラクションの実装言語には，設計との対応という点で表現力が低い．しかしながら，エージェント技術の分野において最も普及している言語は Java であるため，いかに表現力が十分にあろうとも独自の特殊な仕様の言語であっては，マルチエージェントシステムの開発者にとって魅力がない．

これらの現状をふまえて，IOM/T は以下の点を考慮して設計した．

- AUML のシーケンス図と対応が明確
- 1 つのインタラクションを 1 つのコードに記述可能
- 直感的に理解できる状態遷移表現が可能
- Java のような構文規則

この言語を用いることにより，従来のマルチエージェントシステムの開発において困難であったインタラクションの実装の複雑さを軽減する．

IOM/T において，インタラクションは *interaction* 構造として表現され，その内部には，参加するロールを表現する複数の *role* 構造と，実際のプロトコルを表現する *protocol* 構造を記述する．*role* 構造は，Java のクラス定義と同様な形式で記述し，ロールが持つ機能と変数とサブプロトコルを定義する．ロールとはエージェントを抽象化したものであり，インタラクションにおいては動作の主体としてとらえられるものである．*protocol* 構造は，*play* 構造と，*while* などの制御構造との組合せにより表現され，手続き型の言語のようにロールの動作とそれらの相互作用を記述したものである．*play* 構造は，エージェント間のコミュニケーション機能の利用を独自に与えているが，基本的には Java を用いてロールの動作を記述する．また，IOM/T の記述は特定のプラットフォームに依存しない記述である．しかしながら，一般的に，マルチエージェントシステムの実装は，動作プラットフォームに依存してしまう．IOM/T を用いたマルチエージェントシステム開発においても，インタラクションは IOM/T を用いてプラットフォーム非依存に実装するが，そのインタラクションに参加するエージェントは動作プラットフォーム特有の記述を用いる．ただし，エージェントの実装には，各エージェントが行うロールに対して，各ロールの機能から，その機能を実装しているメソッドへのマッピングを追加する．この記述の詳細は対象プラットフォームによって異なるが，**図 1** のように，エージェントを表現するクラスなどの中に，*playing* 構造によって表現する．エージェントの開発者は，利用するインタラクションとそれに参加する際のロールを決定し，そのロールの機能を実現するメソッドを実装するとともに，それに対するマッピングを追加する．

さらに，IOM/T はエージェント技術の国際標準化団体である FIPA [6] が規定する AUML のシーケンス図との対応関係が明確となっており，それらの間の等

```
1  public class BidderAgent extends Agent {
2    // ロール機能のマッピング
3    playing {
4      // createBid 機能は，createNextBid メソッドで実現する
5      createBid = createNextBid;
6    }
7    // createBid 機能の実現
8    public Bid createNextBid(Bid current) {
9      ...
10   }
11 }
```

図 1 ロールの機能のマッピング
Fig. 1 Example of role functionalities' mapping.

価性は π 計算[14] により証明可能である．

3. 契約による設計

本章では，IOM/T における「契約による設計（Design by Contract）[13]」の記述について述べる．契約による設計とは，プログラムコードに対し，その操作を実行するための条件や，その操作の効果，および，その操作によって変化してはならない条件などを記述し，プログラムの実行時にその条件をチェックする開発手法である．そして，明示的に条件を記述することにより，プログラムコードの責任を明確にし，開発を補助するとともに，不具合が存在した場合にはその原因を迅速に把握することが可能となる．契約による設計を用いた開発のためには，Eiffel[2] のように言語仕様にそれが組み込まれている言語を使用するか，または既存言語に対するツールを利用することが必要となる．これまでに，Java に対する契約による設計のためのツールとして，JML[12]，iContract[10] といったものが開発されており，また，AspectJ[8] を用いてアスペクト記述として契約を記述することによって契約による設計を実現することも可能である．

以降，3.1 節において，マルチエージェントシステムの開発における契約による設計の必要性と，その実現のための課題を示す．そして，3.2 節において契約による設計のための IOM/T の拡張について述べ，3.3 節においてその拡張を用いた記述例を示す．

3.1 マルチエージェントシステムの開発における契約による設計

マルチエージェントシステムにおける問題点の 1 つがテスト/デバッグ手法の不足である．一般の分散システムにおいてもいえることであるが，各エージェントは分散して実装され，並列に動作するため，システムにおいて異常が発生した際に，その原因を特定することが難しい．たとえば，オークションにおいて，落札していないにもかかわらず，落札者として認識するエージェントがいたとする．その場合，Auctioneer にバグがあり落札していない Bidder に影響を与えてしまった，Bidder にバグがあり Bidder が落札したと認識した，など様々な原因が考えられる．契約による設計を用いていなければ，それらの原因について 1 つ 1 つ確認をしなければならない．また，各エージェントは柔軟であるがゆえに，その責任が明確でなくなってしまいがちである．Bidder は現在の価格よりも高い価格のビッドを行わなければならない．しかし，その責任が，Bidder にあるのか，Auctioneer がそのようなビッドを拒否しなければならないのか，といった内容は曖昧になってしまいがちである．その点，契約による設計を用いることにより，システムの各状態に対して，異常な状態を明確に定義するため，異常を早期に検出できるとともに，その原因の特定も容易になる．また，各エージェントの責任も明確になるため，開発も容易になる．

このように契約による設計は，現在のマルチエージェントシステムの開発において不足しているテスト/デバッグを補助する手法であるが，既存の言語やツールでは十分にマルチエージェントシステムにおいてチェックすべき事項を表現することはできない．既存のオブジェクト指向開発における契約による設計による設計の言語やツールを用いることにより，オブジェクトやそのメソッドに対して契約を記述し，それを実行時にチェックすることは可能である．しかしながら，マルチエージェントシステムにおいては，各エージェントは自律的に並列に動作しており，各エージェントに対しての契約だけではなく，それらエージェント間の関係について規定することが重要となる．すなわち，エージェント間のインタラクションに対する契約が重要となる．

そこで，マルチエージェントシステムの開発においても，契約による設計を利用するため，IOM/T の記述を拡張し，インタラクション，インタラクションの状態，ロールの行動，ロールの 4 つに対する契約を可能にした．

この拡張により，インタラクションに関するエージェント，および，それらの関係についての責任が明確になり，マルチエージェントシステムの開発がより安全にすすめることができる．

3.2 IOM/T における契約による設計

IOM/T における契約による設計の記述は，iContract[10] と同等の記法を用いて，次のように，コードのコメント中に記述する．また，契約には，事前条件，事後条件，不変条件の 3 種類が存在し，それぞれ，@pre，@post，@invariant のキーワードを用いて表現する．

```
/** コメント
  * @pre 事前条件式
  * @post 事後条件式
  * @invariant 不変条件式
  */
```

3.2.1 IOM/T における契約の分類

IOM/T における契約を記述する対象は次の 4 つである．

- インタラクション

インタラクションに対する契約は，*interaction* 構造の直前のコメントに記述する．インタラクション間の関係に関する契約を記述する．たとえば，EnglishAuction に関して，開始前に Auctioneer と Bidder は，認証インタラクションを実行していなければならないといった契約を記述する．

- インタラクションの状態
  インタラクションの状態に対する契約は，*protocol* 構造と，*while* 構造や *if-else* 構造といった制御構造の直前のコメントに記述する．それぞれの状態におけるエージェント，および，エージェント間の関係を記述する．たとえば，EnglishAucion に関して，CFP とビッドのループにおいて，Bidder は 2 人以上いなければならず，オークション終了時には，Auctioneer の把握する落札価格と，落札した Bidder の落札価格は一致しなければならないといった契約を記述する．
- ロールの行動
  ロールの行動に対する契約は，*play* 構造の直前のコメントに記述する．各ロールの行動において，そのロールが満たさなければならない条件を記述する．
- ロール
  ロールに対する契約は，*role* 構造中のロールの機能，変数の直前のコメントに記述する．ロールの機能，変数が満たさなければならない条件を記述する．

3.2.2 契約の条件式の記述

iContract の契約による設計では，boolean に評価される Java の式と全量限量子，存在限量子，「ならば」の記述を組み合わせて記述する．オブジェクト指向プログラミングでは，これらの記述を用いることにより，プログラムの契約を記述する．マルチエージェントシステムのインタラクションについては，前節で述べた対象のうち，ロールの行動，および，ロールに対する契約は表現することが可能である．しかしながら，インタラクションに対する契約，インタラクションの状態に対する契約を記述するためには，これだけでは不足している．まず，インタラクションの動作の主体であるロールを扱うために，ロール全量限量子，ロール存在限量子を利用する．この記述を用いて，参加するエージェントがインタラクションの実行のたびに変化しても，ロールという概念で契約を扱うことが可能となり，インタラクションの状態の契約において，エージェント間の関係を表現することが可能となる．そして，インタラクションに対する契約を表現するために，*interact* 演算子を用いる．

- 全量限量子
  ⟨Iterator⟩ で指定された Iterator で列挙されるすべての ⟨Class⟩ 型の要素 ⟨var⟩ について，⟨Expr⟩ が成り立つことを示す．この記述はすべての契約において利用可能である．

```
forall <Class> <var>
          in <Iterator> | <Expr>
```

- 存在限量子
  ⟨Iterator⟩ で指定された Iterator で列挙される ⟨Class⟩ 型の要素 ⟨var⟩ について，⟨Expr⟩ が成り立つ要素が存在することを示す．この記述はすべての契約において利用可能である．

```
exists <Class> <var>
          in <Iterator> | <Expr>
```

- ならば
  ⟨Expr1⟩ が成り立つならば，⟨Expr2⟩ が成り立つことを示す．この記述はすべての契約において利用可能である．

```
<Expr1> implies <Expr2>
```

- ロール全量限量子
  ⟨Iterator⟩ で指定された Iterator で列挙される ⟨Role⟩ のすべてのエージェント ⟨agent⟩ について，⟨Expr⟩ が成り立つことを示す．この記述はインタラクション，および，インタラクションの状態に対する契約においてのみ利用可能とする．また，⟨Iterator⟩ は省略可能であり，省略された場合にはインタラクションに参加するすべての ⟨Role⟩ を行うエージェントに対するものとする．

```
forall <Role> <agent>
          [in <Iterator>] | <Expr>
```

- ロール存在限量子
  ⟨Iterator⟩ で指定された Iterator で列挙される ⟨Role⟩ のエージェント ⟨agent⟩ について，⟨Expr⟩ が成り立つ要素が存在することを示す．この記述はインタラクション，および，インタラクションの状態に対する契約においてのみ利用可能とする．また，⟨Iterator⟩ は省略可能であり，省略された場合にはインタラクションに参加するすべての ⟨Role⟩ を行うエージェントに対するものとする．

```
exists <Role> <agent>
          [in <Iterator>] | <Expr>
```

- *interact* 演算子
  ⟨Interaction⟩ で指定したインタラクションを，⟨Agent1⟩，⟨Agent2⟩，... が実行し，⟨Result⟩ で指定された状態で終了したことを示す．この記述は，インタラクションに対する契約にのみ利用可能とする．

```
interact< <Interaction> >
     (<Agent1>, <Agent2>, ...)
                       == <Result>
```

  インタラクションの終了状態の記述を *interaction* 構造の直前のコメントに@state キーワードを用いて記述する．⟨Expr⟩ が成立した状態で終了していれば，このインタラクションは ⟨State⟩ という状態で終了したことを示す．

```
/**
 * @state <State> <Expr>
 */
```

3.3 IOM/T における契約による設計の例

EnglishAuction を契約による設計を用いて IOM/T で記述すると，**図 2** のようになる．2–3 行目において，インタラクションに対する契約として，事前条件としてすべての Auctioneer と Bidder は Authorize インタラクションを success の状態で終了していなければならないことを記述している．そして，4–5 行目では，この EnglishAuction において，success な状態の終了とは，Auctioneer の winner 変数が null でない状態であるとしている．

次に，16–18 行目において，Bidder ロールの createBid 機能について，返り値の price フィールドの値は，引数である current の price フィールドの値よりも大きいものであることを記述している．

また，24–31 行目では，インタラクションの状態としてインタラクション全体の状態に対して契約を記述している．このインタラクションの終了時には，Auctioneer ロールのエージェントの winner 変数が null でなければ，isWin 変数が true の Bidder エージェントが存在し，Auctioneer エージェントの winner が null でなく，isWin 変数が true の Bidder b がいれば，Auctioneer の winner は b であり，それぞれの price 変数は同一でなければならないことを示している．

```
1  /**
2   * @pre forall Auctioneer a | forall Bidder b |
3   *             interact<Authorize>(a, b) == success
4   * @state success forall Auctioneer a |
5   *             a.winner != null
6   */
7  interaction EnglishAuction {
8    Role Auctioneer {
9      ...
10     int price;
11     AID winner;
12     ...
13   }
14   Role Bidder * {
15     ...
16     /**
17      * @post return.price > current.price
18      */
19     Bid createBid(Bid current);
20     boolean isWin;
21     int price;
22     ...
23   }
24   /**
25    * @post forall Auctioneer a
26    *           a.winner != null implies
27    *                       exists Bidder b | b.isWin
28    * @post forall Auctioneer a | forall Bidder b |
29    *           a.winner != null && b.isWin implies
30    *               a.winner == b && a.price == b.price
31    */
32   protocol {
33     ...
34   }
35 }
```

図 2 EnglishAuction に対する契約による設計の記述例
Fig. 2 An example description of design by contract for EnglishAuction.

4. 実　　装

本章では，IOM/T に対する契約による設計で記述された規約をいかにチェックするかを示す．

IOM/T を用いて実装したインタラクションは，特定のエージェントプラットフォームに依存した記述ではない．しかしながら，エージェントはシステムが動作するエージェントプラットフォームに特化したコードで記述しなければならない．そこで，我々は IOM/T コンパイラを開発した．IOM/T コンパイラは，インタラクションのコードと，特定のエージェントプラットフォームのエージェントのコードから，そのプラットフォームにおいて動作するマルチエージェントシステムのコードを生成する．4.1 節において，IOM/T コンパイラの実行コード生成について述べ，4.2 節において，この変換において，契約による設計の記述がどのように反映されるのかを示す．

4.1 IOM/T コンパイラ

本節では，Java ベースの代表的なエージェントプラットフォームである JADE エージェントプラットフォームを例に，IOM/T コンパイラの処理について述べる．2 章でも述べたように，エージェントプラットフォームにおいて動作するエージェントは，そのプラットフォー

ム特有の形式で記述しなければならない．JADE[3]においては，エージェントは jade.core.Agent クラスのサブクラスのインスタンスとして表現され，エージェントの動作は，jade.core.behaviours.Behaviour クラスのサブクラスのインスタンスとして表現される．インタラクションも各ロールの動作組合せであるため，各ロールごとの動作に分割され，複数の Behaviour クラスによって，1 つのインタラクションが表現される．また，この Behaviour クラスは，done メソッドが true を返すまで，action メソッドを呼び出されるという仕組みである．

IOM/T コンパイラは，このようなプラットフォーム特有の形式にマルチエージェントシステムの実装を変換する必要がある．IOM/T コンパイラは，IOM/T によるインタラクション実装と，エージェントの実装を入力として，対象とするエージェントプラットフォームで動作するコードを生成する．JADE プラットフォームで動作するコードを生成する際の処理の概要を，**図 3** に示す．まず，インタラクション中の *role* 構造から，各ロールのインタフェースの定義を生成する．次に，各エージェントについて，実行するロールのインタフェースの実装クラスを生成する．IOM/T コンパイラは，ロールの機能のマッピングを基に，各機能の処理をエージェントクラスに委譲するクラスを生成する．そして，各ロールに対して，IOM/T の *protocol* 構造からロールの動作を抽出して，それぞれの動作を実現

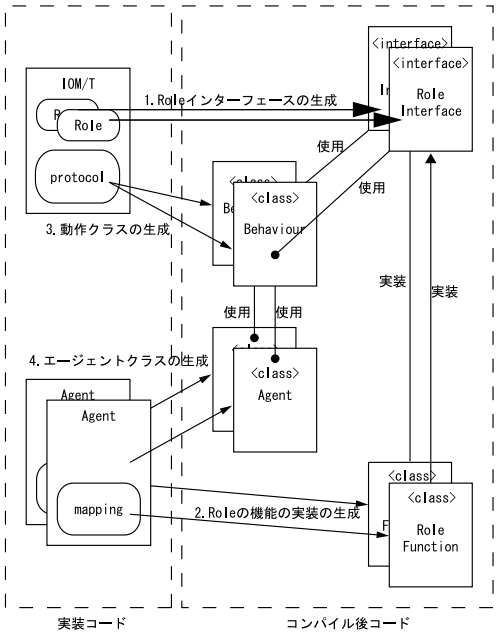

図 3 IOM/T コンパイラの処理
Fig. 3 An overview of the process of IOM/T compiler.

する Behaviour クラスのサブクラスを生成する．このときに，ロールの行動中のロールの機能の利用は，生成したロールのインタフェースを介しての呼び出しとする．最後に，エージェントの実装から，対象とするエージェントプラットフォームで動作するエージェントを生成する．

4.2 契約による設計のチェック

JML や iContract における契約による設計の記述は，Java のクラスやメソッドに対するものであったため，そのクラスに対するメソッド呼び出しなどの操作の前後に，契約が守られているかをチェックすることで対処できた．しかしながら，IOM/T で記述したインタラクションや，インタラクションの状態に対する契約は，複数エージェント，すなわち，複数クラスに関する契約であり，同様の手法ではチェックすることができない．したがって，エージェント間の契約をチェックするためには，チェック処理を追加するだけでは不足であり，エージェントの情報を集める機構も必要である．さらに，直接やりとりするエージェント間以外にまたがる契約もチェックするためには，インタラクションに参加するすべてのロールのすべての情報を把握する必要がある．その情報を集めるためには，やりとりメッセージに付加情報を与えるだけでは不可能であり，参加するエージェントの状態を 1 カ所に収集する機構が必要である．

そこで，各インタラクションに対し，マネージャエージェントを導入することで，エージェント間にまたがるチェックを行う．マネージャエージェントとは，インタラクションの Initiator となるエージェントが生成する．どのようなインタラクションにおいても，必ず 1 つの Initiator エージェントが存在し，1 つのインタラクションに対して，1 つのマネージャエージェントを生成する．そして，インタラクションに参加するエージェントは，インタラクションの各状態で行動し状態の変化の前後に内部状態の内容をマネージャエージェントに通知する．マネージャエージェントは，参加するすべてのエージェントから状態の通知を受け取ると，契約をチェックする．このように，マネージャエージェントを導入することにより，エージェント間にまたがる契約をチェックする．この機構を実現することにより，インタラクションが終了した時点で，マネージャエージェントはすべてのエージェントのすべての状態を把握しているため，各状態についての複数のエージェントにかかわる契約をチェックすることが可能となる．ただし，各エージェントは並列に動作しており，エージェントからの状態通知も非同期に行わ

れ，ある状態のチェック時にはエージェントはすでに次の動作をしていることが想定され，その時点でエージェントがその問題を回避するための処理を実行するようなことはできない．インタラクション実行後に，契約が遵守されていたかどうかを確認できるだけである．また，エージェントからのメッセージが届かなかった場合には，マネージャエージェントはすべての状態を把握できないため，完全なチェックを行うことができない．

IOM/T コンパイラは，このようなチェックを行うようなコードを自動的に生成する．その概要を，**図 4** に示す．まず，Initiator エージェントはインタラクションの開始前に，マネージャエージェントを生成する．そして，参加エージェントに対して，開始メッセージを送信し，インタラクションが行われる．Initiator エージェント，および，参加エージェントは様々なインタラクションを行うが，それぞれが行動をし，状態が変化する前後に現在の状態を通知する．マネージャエージェントは，Initiator エージェントと参加エージェントから通知を受け取り，インタラクションやインタラクションの状態に対する各契約についてのチェックを行う．各エージェントは通知の際にインタラクションにおける状態をあわせて送付する．そのため，マネージャエージェントは，通知を非同期に受信したとしても，チェックすべき状態の各エージェントの情報が集まった時点でチェックする．

また，インタラクション間の契約のチェックでは，終了したインタラクションにおいて，エージェント間でどのような結果に至ったかの情報が必要である．たとえば，認証を行うインタラクションについて，認証依頼者が正しく認証されたのか，そうでなかったのか，といったことを検証する．各エージェントは，このような情報を保持しているかもしれないが，保持している内容，さらには，保持の手法までも，エージェントによって異なる．また，何らかの形態で必要な情報を各エージェントが保持していたとしても，その内容がその後書き換えられてしまっていたり，過去のインタラクションに参加したエージェントによって解釈が異なっているかもしれない．そのため，終了したインタラクションの情報を利用するにあたって，各エージェントにその情報を保持させることは望ましいとはいえない．よって，インタラクション間の契約のチェックには，終了したインタラクションのマネージャエージェントが保持している情報を利用することとする．関係するインタラクションに対するマネージャエージェントを検索し，そのエージェントからそのインタラクションの結果を取得し，インタラクション間の契約が満たされているかをチェックする．

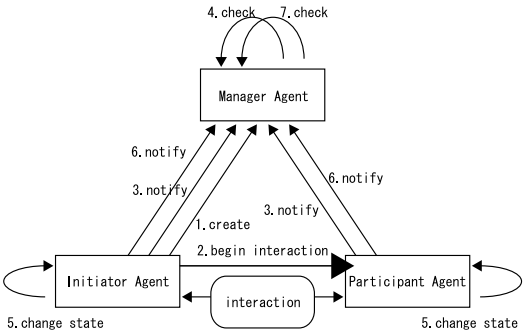

図 4 契約のチェック機構の概要
Fig. 4 An overview of the mechanism for checking contracts.

5. 考　　察

これまでに，IOM/T に対する契約による設計をどのように記述し，それがシステムにおいてどのように実行されるのかを示した．本章では，契約による設計を用いた IOM/T の拡張に関する考察を述べる．5.1 節では，契約による設計を導入するにあたって IOM/T を拡張した点について考察する．また，5.2 節において，関連研究と本研究との位置付けについて述べる．

5.1 IOM/T に対する契約による設計の評価

本節では，契約による設計を従来手法で実現した場合と比較して，IOM/T に契約による設計の概念を導入した利点を考察する．エージェントの分野において最も利用されている言語は Java であり，一般的にマルチエージェントシステムは Java を用いて実装される場合が多い．また，3 章で述べたように，Java に対する契約による設計を行うツールとして，JML や iContract などが存在する．そのため，これらのツールを利用することにより，IOM/T を使用しなくても，契約による設計を用いてマルチエージェントシステムを開発することは可能である．たとえば，図 2 で記述した EnglishAuction を実行するマルチエージェントシステムを JADE のエージェントを直接実装したとする．

4 章でも述べたように，JADE のエージェントの動作，および，インタラクションは，それぞれのエージェントの Behaviour クラスを拡張することにより実装される．Auctioneer ロールと Bidder ロールの Behavior はそれぞれ，**図 5** と**図 6** のように記述できる．ここでは，iContract の記法を用いて契約を記述している．

まず，図 2 の 2–3 行目で記述しているインタラク

```
1  public class Auctioneer {
2    ...
3    private int state;
4    private boolean isEnd = false;
5    ...
6    private Vector targetList;
7    private Vector authorizedList;
8    ...
9    private Bidder winneer = null;
10   private int price;
11   ...
12   public void action() {
13     switch (state) {
14       case 0:
15         doState0();
16         break;
17       ...
18       case N: // last state
19         doStateN();
20         isEnd = true;
21         break;
22     }
23   }
24   /**
25    * @pre for all Bidder b1
26    *          in targetList.elements() |
27    *              exists Bidder b2
28    *                  in authorizedList.elements() |
29    *                      b1 = b2
30    */
31   private void doState0() { ... }
32   ...
33   /**
34    * @post this.winner != null implies winner.isWin
35    * @post this.winner != null implies
36    *           this.price == winner.price;
37    */
38   private void doStateN() { ... }
39   ...
40 }
```

図 5 JADE における Auctioneer の実装例
Fig. 5 An example implementation of Auctioneer in JADE.

```
1  public class Bidder {
2    ...
3    private int state;
4    private boolean isEnd = false;
5    ...
6    private boolean isAuthorized;
7    private Auctioneer auctioneer;
8    ...
9    public void action() {
10     switch (state) {
11       case 0:
12         doState0();
13         break;
14       ...
15       case M:
16         doStateM();
17         break;
18       ...
19       case N: // last state
20         doStateN();
21         isEnd = true;
22         break;
23     }
24   }
25   /**
26    * @pre this.isAuthorized
27    */
28   private void doState0() {
29     ...
30     auctioneer = /* get auctioneer object */
31     ...
32   }
33   ...
34   private void doStateM() {
35     ...
36     Bid bid = ((BidderAgent)myAgent).createBid();
37     ...
38   }
39   ...
40   /**
41    * @post this.isWin implies
42    *          auctioneer.winner == this;
43    * @post this.isWin implies
44    *          auctioneer.price == this.price;
45    */
46   private void doStateN() { ... }
47   ...
48 }
```

図 6 JADE における Bidder の実装例
Fig. 6 An example implementation of Bidder in JADE.

ション間の契約は，図 5 の 25–29 行目と，図 6 の 26 行目として記述されている．これらの個所は，インタラクションにおける最初の状態の処理を 1 つのメソッドとして表現し，そのメソッドに対する契約として記述している．IOM/T で記述した契約の場合には，インタラクション間の契約であることを interact 演算子を用いて明示的に示しているのに対し，図 5 では，Authorize インタラクションを終了した Bidder は，authorizedList に追加されているという仮定のもとで契約を記述している．同様に，図 6 では，Authorize インタラクションが正しく終了していれば，isAuthorized が true になっているという仮定をもとに記述されている．これらの仮定を正しいものとするべき責任は Authorize インタラクションのコードであるべきである．しかしながら，異なるインタラクションは異なるコードに記述されるはずであり，Authorize インタラクションのインスタンスが EnglishAuction に関するインタスタンスに影響を与えることは不自然である．一般的に用いられるであろう構成としては，Behaviour クラスのインスタンスからアクセス可能なオブジェクトにその情報を保持することであるが，そのオブジェクトに正しい情報が格納されていることを保証することは難しい．

次に，図 2 の 25–30 行目で記述しているエージェント間の契約は，図 5 の 34–36 行目と，図 6 の 41–44 行目として記述されている．マルチエージェントシステムにおいては，各エージェントは識別子を用いて表現され，直接インスタンスを保持するということは一般的ではない．しかし，ここでは，Behaviour クラスのインスタンスが取得できるものとして契約を記述している．図 5 においては，Auctioneer クラスの winner フィールドを用いて契約を記述している．しかしながら，isWin フィールドが true である Bidder オブジェクトが複数存在してしまい，Auctioneer エージェントはそのうちの 1 つのみを winner としている場合には，落札に関して不具合が生じているにもかかわらず，契約を満たしていることになってしまう．一方，図 6

においては，このような問題は生じないものの，契約による設計を記述するための仕組の記述と，インタラクションの動作の記述が混同されてしまっている．

これら JADE のコードを直接記述した場合と比較して，IOM/T に対して記述した契約は，直感的に把握できる．また，契約による設計は，そもそも責任を明確にすることによって開発を容易にする手法であるにもかかわらず，契約の記述のための特殊なコードが必要であることや，そのコードに対する責任が不明確になってしまうこともなく，各インタラクションの責任を明確に指定できる．

まとめると，IOM/T を用いて契約による設計を行うことには以下のような利点がある．

- エージェント間，インタラクション間にまたがる契約を明示的に記述できる．
- 契約の記述のためのコードを，インタラクションの動作のコードに追加せずに記述できる．

これらの利点は，JADE の記述の表現力が不足しているからではなく，Java の表現力が不足しているからである．すなわち，非常に密接に結びついているインタラクションを，複数のクラスに分割して記述しなければならないからである．そして，iContract や JML はクラスという単位について契約を記述する言語であるからである．

マルチエージェントシステムを開発するうえでは，これまでのオブジェクト指向技術だけではなく，エージェント間のインタラクションというより大きな粒度でものを見る，インタラクション指向な開発が必要となるのである．

しかしながら，すべてのマルチエージェントシステムの開発に IOM/T が有効なわけではない．IOM/T はインタラクションを集中的に記述する言語であり，インタラクションが動的に追加，変更するようなシステムの開発には向かない．IOM/T を用いた開発プロセスを考慮してみると，次のようになる．

(1) AUML によるインタラクションの設計
(2) IOM/T によるインタラクションの実装
(3) IOM/T に基づくエージェントの実装
(4) マルチエージェントシステムのテスト/実行

このプロセスにおいて，最初の 2 つのフェーズはマルチエージェントシステムを管理するものによって行われる．すなわち，そのマルチエージェントシステムにおいて用いられるインタラクションは，その管理の下で決定されることとなる．一方，そのマルチエージェントシステムに参加するエージェントは，IOM/T で記述されたインタラクションに基づいていればよい．よって，目的，手法などはあらかじめ集中管理するが，参加するエージェントはその管理下で開発される必要がなく，様々な意図を持つシステムの開発が最も適している．そして，そのようなシステムにおいて，契約による設計を用いることにより，集中管理されるべき事項が守られているかをチェックすることができる．このようなシステムは，非常に特殊なシステムというわけではない．たとえば，オークションシステムのようなものが考えられる．システムが行うべきことには，オークションの出品物の募集，オークションによる落札者の決定などがある．これらの目的を達成するためのインタラクションや，最低ビット価格といったそのインタラクションにかかわるシステムの規則は集中管理で決定，実装される．しかしながら，出品物を提供するエージェント，Bidder として動作するエージェントなどは，その規則さえ守っていれば，独自の思考プロセスを持っていてよい t．このとき，IOM/T に契約による設計を適用することにより，独自の思考プロセスがシステムとしての規則に反しないかをチェックすることが可能になる．

5.2 関連研究

前節では，Java に対する契約による設計のツールである iContract，JML を対象として比較したが，本節では，その他の側面の関連研究との位置付けについて考察する．

エージェントシステムの安全性に関する研究としては，形式手法を用いたシステムの検証があげられる．これらに関しては，モビリティ，エージェント間の会話，行動など様々な研究が存在する．これらの研究は，AgentSpeak(L) [15]，CLAIM [1]，AF-APL [16] など論理に基づいたプログラミング言語に対しては非常に有効な手法である．しかしながら，現在，多くの実システムにおいて利用されている言語は Java であり，一般的なシステムを構築するためにこのような言語が使用されるようになるのは難しいと思われる．そのため，我々は Java の構文に似た言語仕様を与えている．

実践的な安全性のための研究として，マルチエージェントシステムに対するデバッグに関するものもいくつかある．エージェントは自律的に動作し，システムの動作も複雑であるため，マルチエージェントシステムのデバッグは簡単なものではなく，ある特化した部分に関してのデバッグを補助する枠組みとなっている．たとえば，エージェントの意思決定に特化したデ

バッグツールとして，意思決定についてのトレースを行い，その不具合を発見するための枠組み[11]などが提案されている．その観点では，本手法はインタラクションに特化したデバッグツールであるといえる．しかしながら，意思決定のデバッグツールでは不正な状態を検出することが自体が難しいのであるが，本手法では，明示的に契約を記述するため，実行時の不正な状態を自動検出することが可能である．

また，近年注目を集めているアスペクト指向プログラミング[9]とも関係がある．アスペクト指向プログラミングとは，複数のクラスなどに分散してしまう関心事の記述をそれらからは分離して，関心事ごとに記述する手法である．この意味で，IOM/T は，異なる関心事である知的動作をするエージェントと，規則に基づいた協調動作であるインタラクションを分離して記述する言語である．そのため，従来の契約による設計の手法であれば，エージェントに対する契約とインタラクションに対する契約とを分離することができなかったが，IOM/T を拡張して契約を記述することにより，インタラクションに関する契約をエージェントとは分離して記述することが可能となった．また，図5と図6 を例に示したように，既存の契約による設計の記述では，マルチエージェントシステムに対する契約の表現力が乏しいため，本来は必要としない記述を付加する必要があり，インタラクションの動作の記述と，契約による設計のための記述が混在してしまっていた．これに対し，IOM/T を用いて契約による設計を用いることにより，不要な記述を排除し，動作記述と契約記述を分離することが可能となった．

6. まとめ

本論文では，より安全なマルチエージェントシステムの開発のために，契約による設計を使用した拡張した IOM/T を提案した．また，この契約の記述がどのようにシステムの動作をチェックするのかに関する仕組みについても述べた．

契約による設計を用いたシステムの安全性の検証は，実行時の動作チェックのみである．今後，マルチエージェントシステムがいっそう普及していくためには，さらなる安全性の保証が必要である．より理論的な保証として，IOM/T，および，契約に対して，形式的な手法を用いた安全性の確保することを計画している．また，より実践的な保証として，インタラクションに対してユニットテストの適用し，安全な開発の手法も確立していく予定である．

参考文献

1) Suna, A. and El Fallah-Seghrouchni, A.: CLAIM: A computational language for autonomous, intelligent and mobile agents, *Programming Multiagent Systems*, *LNAI*, Vol.3067, Springer (2004).
2) Bertrand, M.: *Eiffel: the Language*, Prentice-Hall (1992).
3) CSELT: JADE. http://sharon.cselt.it/projects/jade/
4) Doi, T., Yoshioka, N., Tahara, Y. and Honiden, S., IOM/T: An interaction description language for multi-agent systems, *4th International Joint Conference on Autonomous Agents & Multi Agent Systems AAMAS2005* (2005).
5) FIPA: Agent UML. http://www.auml.org/
6) FIPA: The foundation for intelligent physical agents. http://www.fipa.org
7) FIPA: Interaction diagrams. http://www.auml.org/auml/documents/ID-03-07-02.pdf
8) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An overview of AspectJ, *Lecture Notes in Computer Science*, Vol.2072, pp.327–355 (2001).
9) Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. European Conference on Object-Oriented Programming*, Vol.1241, pp.220–242, Springer-Verlag, Berlin, Heidelberg, and New York (1997).
10) Kramer, R.: iContract — the Java design by contract tool, TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California, pp.295–307 (1998).
11) Lam, D.N. and Barber, K.S.: Debugging agent behavior in an implemented agent system, *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop* (2004).
12) Leavens, G. and Cheon, Y.: Design by contract with jml (2003).
13) Meyer, B.: *Object-Oriented Software Construction*, 2nd edition, International Series in Computer Science, Prentice-Hall (2000).
14) Milner, R.: *Communicating and mobile systems: The π-calculus*, Cambridge University Press (1999).
15) Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language, In

Rudy van Hoe, editor, *7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands (1996).
16) Ross, R.J., Collier, R. and O'Hare, G.M.P.: AF-APL bridging principles & practice in agent oriented languages, *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop* (2004).

(平成 17 年 10 月 3 日受付)
(平成 18 年 3 月 2 日採録)

土肥 拓生

1980 年生．2003 年東京大学理学部情報科学科卒業．同年東京大学大学院情報理工学系研究科コンピュータ科学専攻修士課程進学．2005 年同修了．同年同大学大学院情報理工学系研究科コンピュータ科学専攻博士課程進学．現在に至る．マルチエージェントシステム，ソフトウェア工学に興味を持つ．日本ソフトウェア科学会会員．

本位田真一（正会員）

1978 年早稲田大学大学院理工学研究科修士課程修了．(株）東芝を経て 2000 年より国立情報学研究所教授，2004 年より同研究所研究主幹を併任，現在に至る．2001 年より東京大学大学院情報理工学系研究科教授を併任，現在に至る．2002 年 5 月〜2003 年 1 月英国 UCL ならびに Imperial College 客員研究員（文部科学省在外研究員）．2005 年度パリ第 6 大学招聘教授．早稲田大学客員教授．工学博士（早稲田大学）．1986 年度情報処理学会論文賞受賞．ソフトウェア工学，エージェント技術，ユビキタスコンピューティングの研究に従事．IEEE，ACM，日本ソフトウェア科学会等各会員．本学会理事．