インタラクションに注目したマルチエージェントシステムの効率的な開発手法

土肥 拓生

電気通信大学 情報システム学研究科

博士 (工学) 学位申請論文

2016 年 9 月

インタラクションに注目したマルチエージェントシステムの効率的な開発手法

博士論文審査委員会

主査　大須賀昭彦　教授
委員　栗原聡　教授
委員　田原康之　准教授
委員　古賀久志　准教授

土肥 拓生

2016 年 9 月

Abstract

As a social infrastructure of IoT is achieving, more complex software systems are required. Multi-agent system (MAS), which is composed of “agents” (autonomous software components), is one of the attractive concepts for building such complex software systems. However, building MAS is not easy. A MAS has characteristics such as autonomy, cooperativeness. One of the problems of developing MAS is that there is a large gap between designs and implementations of cooperativeness. In other words, it is difficult to develop interactions among agents appropriately.

Thus, we propose a novel language for describing interactions named “IOM/T” (Interaction Oriented Model by Textual notation). IOM/T enables us to describe an interaction as a code unlike in the case of the existing object oriented language such as Java. We also establish a way to validate the equivalence of designs in sequence diagrams and implementations in IOM/T using pi-calculus. We also introduce the concepts of “Design by Contract” and “Unit Testing” to IOM/T. Because of these features of IOM/T, developers can implement interactions more easily. On the other hand, we show a design process for an iterative development focusing on interactions. A iterative development is one of the key idea of agile development. Recently, agile development receives much attention. We assume each requirement (user story) is satisfied an interaction in the software. We propose the way to design interaction on the basis of the user story.

We believe our solution which consider interactions in all phases of software development make MAS development more efficient.

概要

マルチエージェントシステム (MAS) の開発の課題の一つは，インタラクションを適切に開発することが難しいことである．そこで，本研究ではインタラクションを一つのソフトウェアモジュールとして表現する記述言語 IOM/T を提案する．IOM/T は，実装したインタラクションとシーケンス図による設計との等価性を π 計算を用いて可能にする．また．インタラクションに対する契約による設計の拡張や，インタラクションをソフトウェアモジュールと見做した単体テスト手法についても述べる．さらに，近年普及してきたアジャイル開発で MAS を開発する場合において，インタラクションに注目して，要求 (ユーザストーリ) から設計を導く手法について示す．IOM/T により，これまでの MAS 開発の課題が解決するのみならず，より堅牢性を高めたり，より容易に開発することが可能となる．

目 次

図 目 次

表 目 次

アルゴリズム目次

ソースコード目次

謝辞

この博士課程の研究は，多くの人たちに支えられて実施することができました．

まず，研究に対する考え方，姿勢に至る基礎的な部分から，将来のビジョンにまで渡りご指導を頂いた本位田真一教授に心より感謝申し上げます．本博士論文のみならず，今の自分があるのは，まさに本位田真一教授のご指導，ご支援のおかげであり，言葉では言い表せないほど感謝しております．

そして，主任指導教員として，様々な面でご指導，ご指摘を頂いた石川冬樹准教授に感謝を伝えたいです．なかなか成果を出せずにいた中でも，常に暖かく支援をしてくださいました．石川冬樹准教授と議論をさせて頂くことで，本論文の道筋を立てることができました．指導教員として，多大なる励ましを頂きました大須賀昭彦教授，田中健次教授にも感謝しております．指導教員の方々のお力添えがあって，この博士論文があると思っております．

審査会において，多くのご助言を頂きました，栗原聡教授，田原康之准教授，古賀久志准教授に感謝の意を表します．頂いたご助言により，本論文を一層，洗練することができました．

また，吉岡信和准教授からは，幅広い知見に基づいた様々なご助言を頂きました．本論文の研究内容の立ち上がりの時点で方向性を定められたのは，頂いたご助言のおかげであり，心より感謝の意を表します．

そして，研究活動に理解を示してくれた，寺地知帆社長をはじめとした株式会社レベルファイブのみなさまにも感謝の意を表します．末永俊一郎氏と，前澤悠太氏には，業務の傍らの研究活動に様々な面でご協力，ご配慮を頂き，本論文をまとめる環境を用意して頂きました．また，小牟田俊介氏，飯野匠氏，猪野李紗氏，大西美沙氏，田中健吾氏，小森弘彬氏には，本研究の実証実験の被験者としてご協力を頂き，ありがとうございます．

最後に，私の家族に深い感謝を表したいと思います．家庭で過す時間から，研究活動に時間を割いてしまっても，常に笑顔で応援し続けてくれた妻の黎と，そばでいつも笑顔でいてくれた娘の真歩，結真は，研究活動の原動力となっていました．ありがとう．

第1章　はじめに

近年，Internet of Things(IoT)[6] という言葉が代表するように，様々なものに計算資源が内包され，それらがインターネットに接続される社会基盤ができつつある．IoT の社会基盤においては，様々な場所で様々な計算資源を利用することが可能となり，それにより得られる情報を組み合わせることで，これまでは想像すら難しかった新しい情報システムが実現できる可能性がある．しかしながら，社会基盤として IoT が実現されたとしても，その可能性を最大限に利用したシステムを開発することは容易ではない．利用可能な計算資源は環境に応じて変化し，また，不安定なネットワークも存在するため，ソフトウェアはそれぞれの状況下に応じて，サービスや可用性を維持する複雑な振舞いが求められることとなる．特に，システム内に複数の意図が存在し，それぞれの意図の達成を目指すことが必要な場合は，システムの振る舞いは非常に複雑になる．

このような複数の意図が存在するシステムの構築には，マルチエーエジェントシステム(MAS)[89] の適用が有効的である．MAS とは，環境を知覚し自律性を持ったエージェントと呼ばれるソフトウェアコンポーネントから構成される，自律分散構成のソフトウェアシステムである．個々のエージェントがそれぞれの環境に応じて判断することで，状況に応じた計算資源の利用や，ネットワークの分断に対する対応など，中央集中型のシステムでは実現が難しい有用な性質を実現することができる．

しかしながら，MAS に基づくソフトウェアシステムの構築にも課題が存在する．MAS の主要な要素として，自律性，協調性を実現する必要がある．特に協調性を持ったエージェント間のインタラクションの実現に課題がある．各エージェントはインタラクションプロトコルと呼ばれる一連の協調手順に基づき振る舞うが，ソフトウェア開発において，このインタラクションプロトコルの開発には大きく 2 つの課題が存在する．

1 つ目は，インタラクションの設計と実装に大きな隔たりがあることである．システムの設計においては，各エージェントがどのように協調する必要があるかを検討し，UML のシーケンス図などを用いインタラクションプロトコルとして規定する．一方，システムの実装においては，インタラクションプロトコルのうち，各エージェントに担当する部分の振舞いのみを個別に実装することになる．結果として，設計においてはインタラクションプロトコルとして独立し，かつ，完結した形で表現されていたものが，実装においては

各エージェントに分散される．このため個々のエージェントの実装だけからはインタラクションプロトコル全体を把握することができなくなり，設計と実装に大きな隔たりが生じてしまう．

2つ目は，インタラクションにおける各エージェントの状態遷移の実装が困難であることである．各エージェントはインタラクションプロトコルにおいて現在の状態を管理し，それに応じた振舞いをする必要がある．しかし，近年主流となっているオブジェクト指向言語では，インタラクションプロトコルの状態遷移を可読性の高い記述で表現しインタラクションプロトコルの全体の構造を把握できるようにすることができず，インタラクションプロトコルの改変や，実装の修正に困難が伴う．

そこで，本研究では，システム開発において課題があるインタラクションを，システム開発における視点の中心として捉えるMASの開発について提案する．まず，MASにおけるインタラクションプロトコルの実装のためのインタラクション記述言語，IOM/T(Interaction Oriented Model by Textual notation) を開発した．IOM/Tは，Javaと類似した言語構造の記述言語であるが，設計において単一のシーケンス図で表現されるインタラクションプロトコルを，単一の実装コードとして記述することが可能な言語である．また，UMLのシーケンス図，IOM/Tのコードに対して，π計算により意味論を定義し，UMLのシーケンス図とIOM/Tの等価性をπ計算により判断する仕組みを確立した．

さらに，インタラクションの実装をより効率的に行なうために，IOM/Tの拡張についても構築した．まず，契約による設計の概念を適用し，IOM/Tの各コードブロックに対して，事前条件，事後条件，不変条件を付加的に指定することにより，それらの条件の検証を容易に実施する仕組みを提供することにより，より堅牢なシステムを実現するための拡張を行った．次に，従来の実装方式では，インタラクションプロトコルは，各エージェントの実装に分散してしまうため，MASとして期待する振舞いかどうかはテストすることができたが，インタラクションプロトコルそのものが期待通りに実装されているかを検証することは難しかった．これに対しIOM/Tにおいては，インタラクションプロトコルが単一のモジュールで実装されているため，このプロトコルに対しモックとなるエージェントを指定することで，インタラクションプロトコルが期待通りに実装されているかを確認するためのインタラクションの単体テストの実施手法について確立した．さらに近年では，アジャイル開発に期待が集ってきており，アジャイル開発，特にScrumを適用してMASを開発することを想定し，Scrumの成果物，セレモニーと，IOM/Tを用いた開発の進め方についても検討を行った．スプリント計画ミーティングにおいて，要求である各ユーザストーリをインタラクションプロトコルとして導出し，そのインタラクションプロトコルを設計する手法について確立した．これにより，ユーザストーリからIOM/Tによ

る実装までのインラクションの対応が明白になり，アジャイル開発に代表される反復開発をより効率的に行うことができる．

従来のオブジェクト指向言語では，インタラクションプロトコルの実装は煩雑になってしまうが，IOM/T を用いることにより，直感的に把握しやすい可読性の高い実装となる．結果として，MAS の主要要素である協調性の開発効率が向上し，MAS の開発効率も向上させることができる．

以降，第 2 章で，本研究が扱う MAS 開発の難しさについて考察し，第 3 章，第 4 章にて，その課題のソリューションとして提案するインタラクションに注目した開発手法について示す．その後，第 5 章にて，その評価と有効性について検討を行い，関連研究について第 6 章にて述べ，第 7 章にてまとめる．

第2章　マルチエージェントシステム (MAS)開発の課題

本章では，本研究の課題である，MAS 開発，特に，インタラクションプロトコルの開発の課題について述べる．まず，2.1 節にて，本研究において想定する MAS 開発の全体像について述べ，その前提における，MAS のインタラクションの開発には，どのような課題があるのかを，2.2 節にて明確にする．

2.1　MAS の開発

MAS は分散システムの実現手法の一つである．特に，MPI[37] を利用して実装されたシステムや，Apache Hadoop[87]，Apache Spark[91] に代表される MapReduce[28] によるシステムのように分散する各コンポーネントが統一した意思決定に基づいて動作するのではなく，個々のコンポーネントがそれぞれの意図を持つような分散システムに適している．

例えば，スマートハウス向けの制御プロトコルおよびセンサーネットプロトコルであるエコーネット [25] という標準規格がある．この規格に基づく Home Energy Management System(HEMS)，スマートメータ，エアコンなどの家電機器は，それぞれのベンダーにより開発される．個々の機器は，規格に従って互いに通信・制御のやりとりを互いに実施するが，各機器ごとに実施すべき処理がある．また，電力の需要と供給の事情に応じて調整して節電を目指すデマンド・レスポンス (DR) が注目されてきている．この DR の自動化を目指した Open ADR[4] という規格がある．この場合も，通信・制御は共通規格に基づきながらも，電力会社，アグリゲータ，需要家はそれぞれの意図を反映した意思決定により，システム全体が調整されることとなる．このように，MAS の適用が有効と想定されるシステムの需要は高まりつつある．

一方，MAS も様々な実現方法が考案されている．MAS を構成するエージェントの明確な定義は確立されていないが，自律性，協調性，移動性，学習性，適応性，永続性などにより特徴付けられる．特に，自律性，協調性は，マルチエージェントを構成するエージェントには必須の要素である．

本研究では，各エージェントには役割が与えられ，エージェント間のコミュニケーションプロトコルは規定されているものを対象とする．そして，自律性とは，その役割に対して，実施の可否，時期，方法などについて，外部からの指示に従うだけでなく，自身で判断して決定する能力を持っていることとし，協調性とは，規定されたプロトコルに従ったコミュニケーションを実施するが，プロトコルに従う限り，どのような相手とも協調することができることとする．つまり，エコーネットや Open ADR のような共通のプロトコルが規定され，様々な機器が個々の意図を持って判断するようなシステムを，各機器がエージェントにより制御され，それらが協調することにより MAS として機能するシステムを想定している．

なお，MAS の開発も，一般的なソフトウェアシステムと同様に，その MAS により実現したい要求を明確にし，その要求を実現するためのアーキテクチャを設計し，その設計に基づき MAS を実装することを前提としている．

2.2 インタラクション開発の課題

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捉えられ，UML[69] のシーケンス図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る．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全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．

例えば，図 2.1 のシーケンス図に示す単純なインタラクションを実現する実装を検討する．なお，ここでは，マルチエージェントシステムのフレームワークとして，10 年以上に渡って開発が継続され，代表的なフレームワークの一つである JADE (Java Agent DEvelopment Framework)[13] を用いた実装を考える．

JADE は Java ベースのマルチエージェントフレームワークであり，エージェントは Agent クラスのサブクラスとして実装される．また，エージェントの振舞いは，Behaviour クラスのサブクラスとして実装され，図 2.2 に示すような構成となる．

そして，このクラス図において，図 2.1 で示された 1 つのインタラクションは，*AliceBehaviour* と *BobBehaviour* として実装されるのである．つまり，もともとはインタラクションとして独立して，個別に管理していた要素が，複数のクラスに分散して実装されるのである．

更に，*AliceBehaviour* は，リスト 2.1 のように実装される．JADE のエージェントの振舞

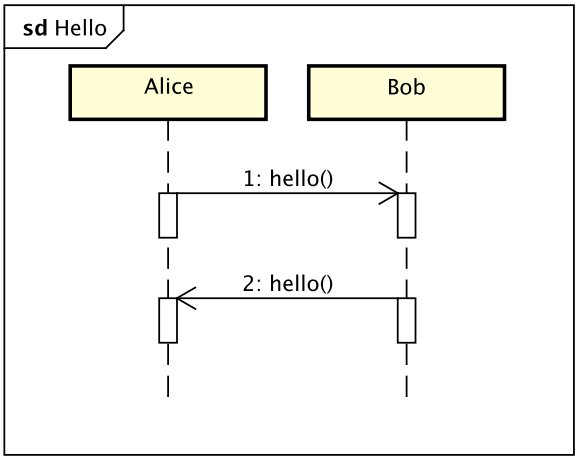


図 2.1: 単純なインタラクションのシーケンス図

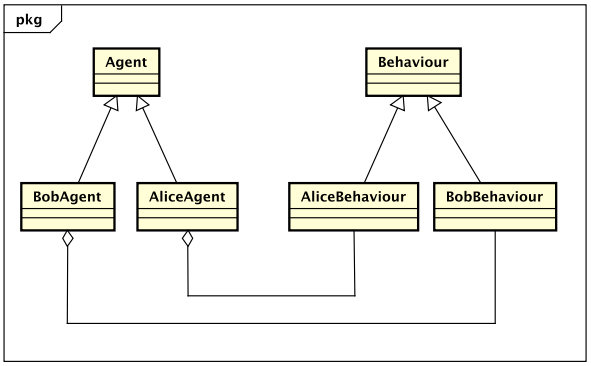


図 2.2: 単純なインタラクションを実現するクラス図

いは，そのエージェントが保持する Behaviour クラスの *action*() メソッドを繰り返し呼びだされることにより実現される．そして，インタラクションの進行の状況を *AliceBehaviour* クラスのインスタンス変数 *status* に記憶し，その値に応じた処理を *action*() メソッドで実行している．ここで重要なのは，状態遷移をインスタンス変数の値が変更することで実現しているが，*action*() 内の処理で依存関係を示すことができず，インタラクションの状態遷移をソースコードから把握することは非常に難しいということである．今回の例においては，非常にシンプルなインタラクションであるとともに，自律性などを処理を記述していないため，コードから全体像を把握することはできなくはない．しかし，インタラクションがより複雑となり，自律的な動作の処理が加わった場合は，ソースコードから正しくインタラクションを抽出することは現実的ではない．

リスト 2.1: AliceBehaviour の実装

```
1  class AliceBehaviour extends Behaviour {
2    private enum STATUS = { INIT, WAITING, DONE }
3    private STATUS status = INIT;
4    public void action() {
5      switch (status) {
6        case INIT:
7        myAgent.sendMessage(new ACLMessage(ACLMessage.INFORM));
8        status = WAITING;
9        break;
10       case WAITING:
11       ACLMessage response = myAgent.receive();
12       if (response != null) {
13         status = DONE;
14       }
15       break;
16     }
17   }
18   public boolean done() {
19     return status == DONE;
20   }
21 }
```

ところで，MAS のエージェントには，大きく分けると，自律性と協調性が備わっている．つまり，その時々の状況に応じて判断して行動が変化する機能と，エージェント同士がコミュニケーションを取りシステムとして稼動するための機能が必要となる．後者の協調性に注目するが，人間の会話コミュニケーションのように無秩序なコミュニケーション

は対象としていない．我々が対象とする MAS によるソフトウェアシステムは，意思決定は各エージェントが自律的に行なうが，その意思決定を，規定されたプロトコルに従ってエージェント間で共有することにより，全体として，複雑な意思決定をする MAS である．

そのため，我々が対象とする MAS においては，プロトコルは柔軟なものではない．そのため，複数のエージェントが同一のインタラクションを行う際には，インタラクション部分に関しては，同様の処理が実行されることを期待する．しかし，それぞれのエージェントの実装は別であるため，単純に実装をしてしまうと，同様の処理が記述される箇所がそれぞれのエージェントに内在してしまう．特定のインタラクションに参加可能なエージェントクラスを用意し，そのクラスを継承することによって，重複箇所を避けることは可能であるが，複数のインタラクションに参加することを想定すると，多重継承 [20][85], あるいは，Mix-in[19] が必要となる．しかしながら，ダイアモンド継承など多重継承にも問題があり，Java を始めとする多くの言語は Mix-in をサポートしていない．

そのため，コードの重複を容認するか，重複を避けるためには，インタラクションの一部を実装したコンポーネントを作成し，そのコンポーネントに移譲するかしかない．しかしながら，インタラクションの一部を委譲することは，設計が不必要に複雑になる可能性がある．

すなわち，インタラクションプロトコルの実装は，DRY(Dont't Repeat Yourself) 原則 [44] を守ることが難しいのである．

昨今，DevOps[9]，ソフトウェア進化 [36] という言葉が表すように，ソフトウェアシステムは一通りの機能が実装されて完了となるのではなく，運用を通して，ソフトウェアシステム自体を改善し続けることによって，その価値が発揮されると言われるようになっている．その状況下においては，MAS も修正し続けられる必要があり，MAS の中核をなすインタラクションプロトコルの実装コードの可読性は非常に重要である．

以上のことより，MAS の開発には，インタラクションプロトコルにまつわる次の課題がある．

- インタラクションプロトコルの設計と実装の隔りが大きい
- インタラクションプロトコルの実装コードの可読性が低い
- インタラクションプロトコルの実装において DRY 原則を守りにくい

第3章　IOM/T: Interaction Oriented Model by Textual Notation

本研究では，第2章で示した，MAS開発における課題を，インタラクションに焦点を当てたソフトウェア開発を実施することにより解決を目指す．本章では，その中心となるインタラクション記述言語IOM/T[31]について述べる．

まず，3.1節でIOM/Tの言語設計について，3.2節にてIOM/Tの言語仕様について述べる．そして，3.3節では，IOM/Tで記述したインタラクションと，UMLのシーケンス図の関係について示す．その後，IOM/Tで記述したインタラクションを利用したMASがどのように実行されるかについて，3.4節で示す．

3.1　IOM/Tの設計思想

第2章で示した課題を解決するためには，まず第一に，一つのインタラクションプロトコルは，一つのコードに記載できなければならない．このことは，インタラクションプロトコルは個々のエージェントに依存するものではなく，独立した概念として実装されなければならないことを意味する．そのためには，従来のオブジェクト指向言語のエージェントプラットフォームのように，MASは複数のエージェントから構成されると捉えるのではなく，インタラクションプロトコルもMASの構成要素でなければならない．つまり，MASはインタラクションプロトコルとそのインタラクションに参加するエージェントから構成されると捉える必要がある．

また，一つのインタラクションプロトコルを一つのコードして表現できるだけでなく，シーケンス図との対応が明確であることが望ましい．シーケンス図はオブジェクトやエージェント間のインタラクションを表現する道具として広く普及しており，設計ドキュメントとしても一般的なものである．そのため，MASに求められる要件を定義し，その要件を実現するMASのインタラクションプロトコルは，シーケンス図で表現されることになるため，シーケンス図とインタラクションプロトコルの実装コードの対応が明確であれば，実装そのものが容易であるだけでなく，改修や要件変更に伴なう設計変更に対する対

応が容易となるのである．

さらに，実装したインタラクションから，各エージェントの状態遷移も容易に理解できるべきである．あるエージェントに注目した際のインタラクションにおける状態遷移は，シーケンス図で表現される状態遷移のみ表現できれば十分である．リスト 2.1 で示したような状態遷移の実装は，複雑な状態遷移も表現が可能であるが，その分，実装コードからその状態遷移を読み取ることは難しい．そのような複雑な表現ができることよりも，インタラクションにおける状態遷移が表現でき，かつ，その状態遷移が実装コードから容易に読み取れることが重要なのである．

そして，現在，最も利用されているプログラミング言語の一つは，Java である [21]．これは，MAS の開発においても例外ではない．そのため，構文規則が Java と同等であることは，多くのエンジニアにとって理解しやすく言語仕様であり，言語の学習コストを低減させるために重要である．

これらのことを踏まえ，IOM/T は以下の性質を持つ，インタラクション記述言語として設計した．

- 一つのインタラクションを一つのコードに記述
- UML のシーケンス図と対応が明確
- インタラクションにおけるエージェントの状態遷移の可読性が高い
- 構文規則が Java と類似

3.2　IOM/T の言語仕様

本節では，3.1 節で述べた設計思想に基づいて開発したインタラクション記述言語 IOM/T の言語仕様について述べる．

3.2.1　インタラクションの定義

IOM/T では，キーワード *interaction* を指定したコードブロックで，インタラクションプロトコルを定義する．インタラクションプロトコルは，そのインタラクションに参加する 2 つ以上のロールと，プロトコルによって構成される．

3.2.2 ロールの定義

ロールとは，そのインタラクションにおける役割を定義したものであり，エージェントを抽象化した概念である．ロールは，キーワード *role* を指定したコードブロックで定義され，そのコードブロックでは，そのロールとして振る舞うために必要な要素を宣言する．Java のインターフェースの定義と同様に，そのロールとして振る舞うためにエージェントが備えている必要がある機能を，メソッドの宣言として記述する．また，インタラクションの遂行に必要な情報を保持する変数も宣言することができる．

3.2.3 プロトコルの定義

プロトコルとは，そのインタラクションにおいて，各ロールがどのような振る舞いをし，どのようなメッセージのやりとりを行うのかを規定するものである．プロトコルの定義は，制御構造と，ロールの振る舞いにより構成される．制御構造としては，Java の構文規則と類似した記法で，条件分岐，繰り返し，並列実行を組み合せることが可能である．

ロールの振る舞い

キーワード *play* を利用したコードブロックにより，特定のロールのエージェントがインタラクションのその時点で，どのような振る舞いをするかを記述する．このコードブロックの中には，Java コードとして，そのロールの振る舞いを記述する．なお，このコードブロックでは，そのロールで定義された機能と変数を利用することができる．さらに，メッセージの送受信を記述する下記の特殊な記述が利用可能である．

```
// メッセージmとして，msgを単一のエージェントへ送信
void send<m>(Agent agent, Message message);

// メッセージmとして，msgを複数エージェントへ送信
void send<m>(Collection<Agent> agents, Message message);

// メッセージmを受信
Message recv<m>();

// メッセージmを複数のエージェントから受信
void recv<m>(List<Message> messages);
```

ただし，一つの play 構造中には，メッセージ受信は一回までであり，メッセージ送信後にメッセージ受信を記述することは許容しない．

図 2.1 のインタラクションを記述すると，リスト 3.1 のようになる．

リスト 3.1: IOM/T の記述例

```
1  interaction Hello {
2    role Alice {
3    }
4    role Bob {
5    }
6    protocol {
7      play Alice {
8        Message msg;
9        send<hello1>(bob, msg);
10     }
11     play Bob {
12       Message msg = recv<hello1>();
13
14       Message res;
15       sendAsync<hello2>(res);
16     }
17     play Alice {
18       Message msg = recv<hello2>();
19     }
20   }
21 }
```

条件分岐

キーワード *if*，*else* を使った，Java の条件分岐と同様な記述で，インタラクションプロトコルの条件分岐を表現する．ただし，条件分岐の条件は，単一のロールの機能と，ロールの変数による条件のみを指定することができるものとする．つまり，条件分岐は単一のロールが判断してプロトコルが分岐するもののみ表現できるものを対象とし，各分岐の最初のメッセージは，判断を行なうロールからの送信でなければならない．

図 3.1 のインタラクションを記述すると，リスト 3.2 のようになる．

リスト 3.2: 条件分岐を含む IOM/T の記述

```
1  interaction Loop {
2    role Role1 {
3    }
4    role Role2 {
```

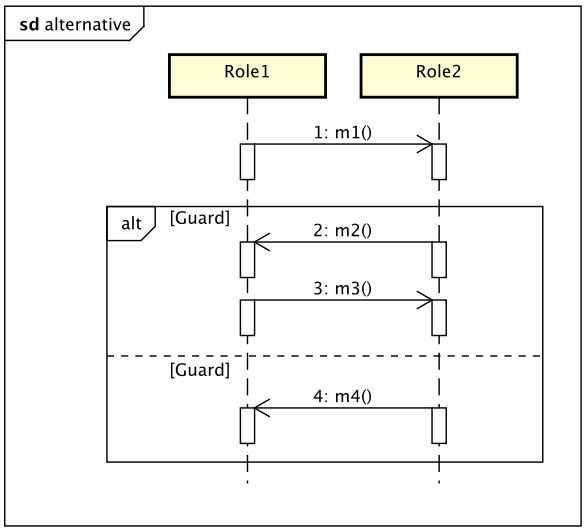


図 3.1: 条件分岐を含むシーケンス図

```
5     boolean acceptable(Message msg)
6     Message offer;
7   }
8   protocol {
9     play Role1 {
10      Message msg;
11      send<m1>(role2, msg);
12    }
13    play Role2 {
14      offer = recv<m1>();
15    }
16    if (Role2.acceptable(offer)) {
17      play Role2 {
18        Message msg;
19        send<m2>(role2, msg);
20      }
21      play Role1 {
22        Message msg = recv<m2>();
23
24        Message res;
25        send<m3>(role2, res);
26      }
27      play Role2 {
28        Message res = recv<m3>();
29      }
```

```
30      } else {
31        play Role2 {
32          Message msg;
33          send<m4>(role2, msg);
34        }
35        play Role1 {
36          Message msg = recv<m4>();
37        }
38      }
39    }
40  }
```

繰り返し

キーワード *while* を使用して，Java の繰り返し処理と同様な記述でインタラクションプロトコルの繰り返しを表現する．なお，繰り返しの条件は，単一のロールの機能と，ロールの変数による条件のみを指定することができるものとする．また，繰り返し処理の終了時には，その単一のロールから，関連するロールに対して，終了を通知するメッセージ送信があるものを対象とする．つまり，条件分岐は単一のロールが判断して，繰り返しを実施するものを対象とする．

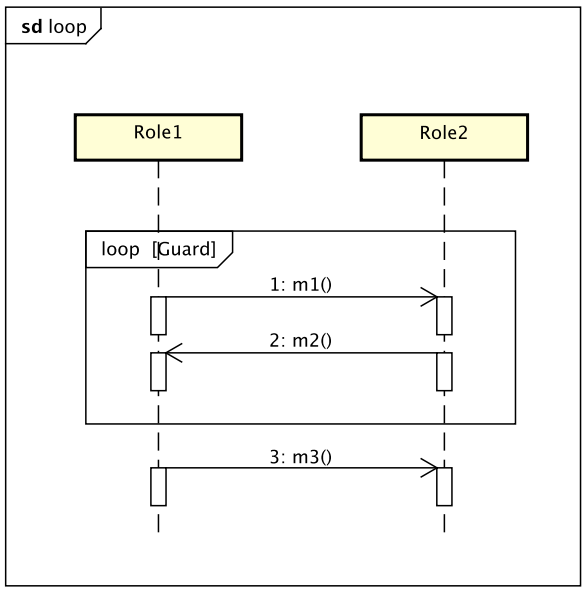


図 3.2: 条件分岐を含むシーケンス図

図 3.2 のインタラクションを記述すると，リスト 3.3 のようになる．

リスト 3.3: 繰り返しを含む IOM/T の記述

```
1  interaction Loop {
2    role Role1 {
3      boolean isContinue();
4    }
5    role Role2 {
6    }
7    protocol {
8      while (Role1.isContinue()) {
9        play Role1 {
10         Message msg;
11         send<m1>(role2, msg);
12       }
13       play Role2 {
14         Message msg = recv<m1>();
15
16         Message res;
17         send<m2>(role1, res);
18       }
19       play Role1 {
20         Message res = recv<m2>();
21       }
22     }
23     play Role1 {
24       Message msg;
25       send<m3>(role2, msg);
26     }
27     play Role2 {
28       Message msg = recv<m3>();
29     }
30   }
31 }
```

並列実行

キーワード *parallel* を利用したコードブロックにより表現する．このコードブロックの中には，複数のコードブロックを記載し，それらのコードブロックが並列実行するものとする．

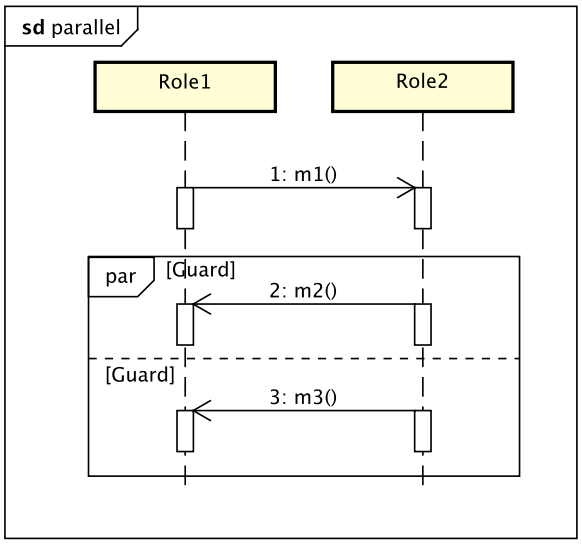


図 3.3: 並列実行を含むシーケンス図

図 3.3 のインタラクションを記述すると，リスト 3.4 のようになる．

リスト 3.4: 並列実行を含む IOM/T の記述

```
1  interaction Parallel {
2    role Role1 {
3    }
4    role Role2 {
5    }
6    protocol {
7      play Role1 {
8        Message msg;
9        send<m1>(role2, msg);
10     }
11     play Role2 {
12       Message msg = recv<hello1>();
13     }
14     parallel {
15       {
16         play Role2 {
17           Message msg;
18           send<m2>(role1, msg);
19         }
20         play Role1 {
21           Message msg = recv<m2>();
22         }
```

```
23        }
24        {
25          play Role2 {
26            Message msg;
27            send<m3>(role1 , msg);
28          }
29          play Role1 {
30            Message msg = recv<m3>();
31          }
32        }
33      }
34    }
35  }
```

また，付録 A に，IOM/T の構文規則を示す.

3.2.4 IOM/T の実装例

図 3.4 に示すやや複雑なインタラクションプロトコルは，IOM/T で記述するとリスト 3.5 のように記述することができる.

リスト 3.5: やや複雑なインタラクションの IOM/T による記述

```
1  interaction Sample {
2    role Role1 {
3      boolean isFirstLoopContinue();
4      boolean isSecondLoopContinue();
5      Agent role2;
6    }
7    role Role2 {
8      boolean isFirstCase();
9      Agent role1;
10   }
11   protocol {
12     while (Role1.isFirstLoopContinue()) {
13       play Role1 {
14         Message msg;
15         send<m1>(role2 , msg);
16       }
17       play Role2 {
18         Message msg = recv<m1>();
```

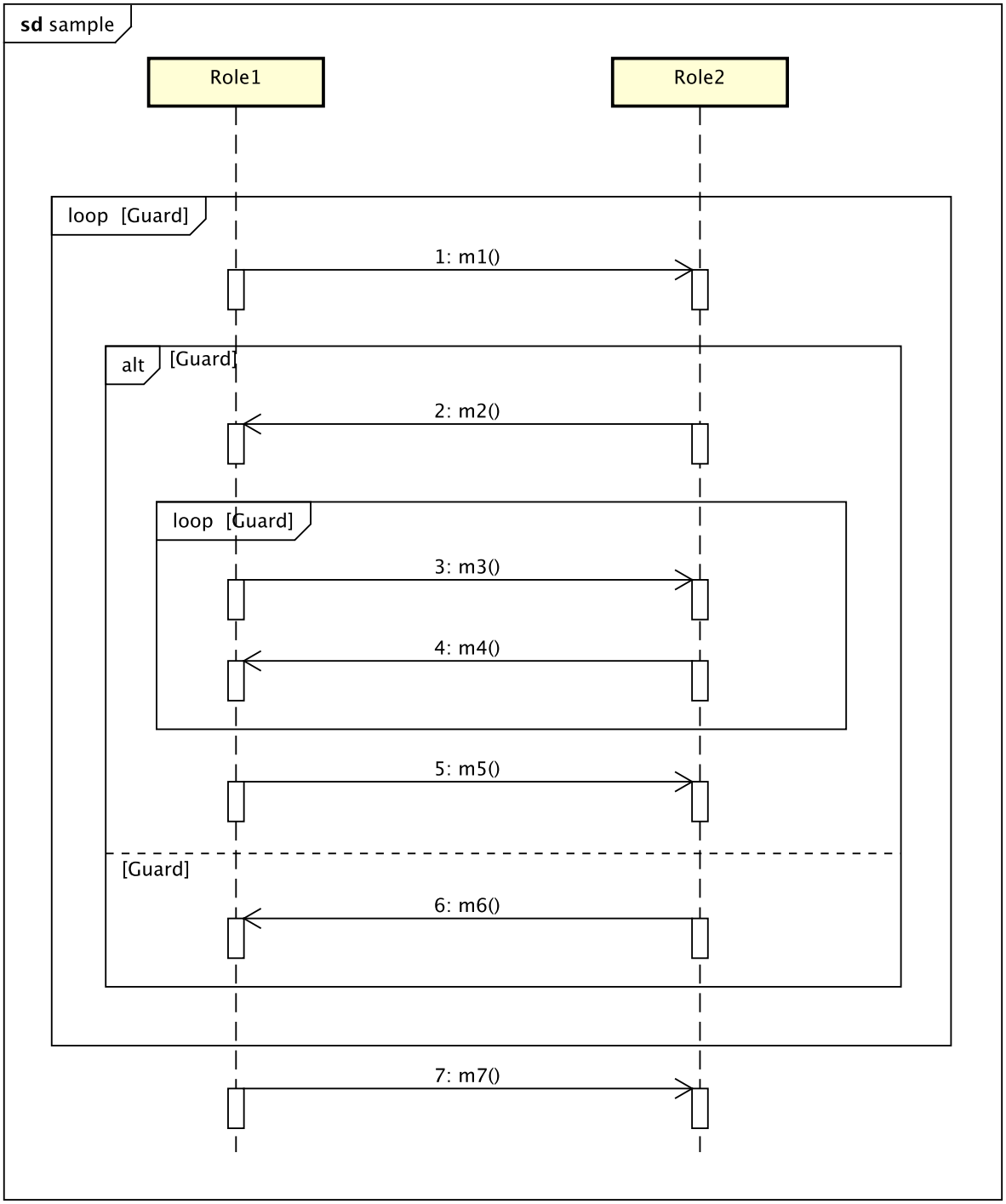


図 3.4: やや複雑なインタラクションのシーケンス図

```
19        }
20        if (Role2.isFirstCase()) {
21          play Role2 {
22            Message msg;
23            send<m2>(role1, msg);
24          }
25          play Role1 {
26            Message msg = recv<m2>();
27          }
28          while (Role1.isSecondLoopContinue()) {
29            play Role1 {
30              Message msg;
31              send<m3>(role2, msg);
32            }
33            play Role2 {
34              Message msg1 = recv<m3>();
35              Message msg2 = ...;
36              send<m4>(msg1.getSender(), msg2);
37            }
38            play Role1 {
39              Message msg = recv<m4>();
40            }
41          }
42          play Role1 {
43            Message msg;
44            send<m5>(role2, msg);
45          }
46          play Role2 {
47            Message msg = recv<m5>();
48          }
49        } else {
50          play Role2 {
51            Message msg;
52            send<m6>(msg);
53          }
54          play Role1 {
55            Message msg = recv<m6>();
56          }
57        }
58      }
```

```
59      play Role1 {
60        Message msg;
61        send<m7>(msg);
62      }
63      play Role2 {
64        Message msg = recv<m7>();
65      }
66    }
67  }
```

3.3　IOM/T とシーケンス図

IOM/T は，3.2 節で示した言語仕様を持つため，一つのインタラクションを一つのコードとして記述できる．また，その構造上，シーケンス図との対応も読み取りやすい言語となっている．しかしながら，開発者が目視で対応関係を把握しやすいだけでは，安全に開発を進めることができないため，本節では，IOM/T とシーケンス図の π 計算 [65] を用いた形式的な等価性を検証する方法について述べる．形式的に IOM/T とシーケンス図の対応が扱えるようになることで，設計されたインタラクションプロトコルが，IOM/T の記述として正しく実装されているかを検証することが可能となる．

我々は，まず，シーケンス図の各要素に対して，インタラクション，すなわち，メッセージの送受信という観点で，各ライフラインごとに π 計算により形式的に意味論を定義した．まず，ライフライン R を次のように定義する．

$$R \;=\; new\ done([[P]](done) \mid done())$$

ここで，$[[P]](done)$ とは，そのライフライン上のプロトコルに対応したプロセル P を実施後，*done* チャネルでメッセージを送出するプロセスである．

そして，例えば，「m というメッセージが送出されている」のであれば，その部分プロセス $A(x)$ は，π 計算において，以下のように定義する．

$$A(x) \;=\; new\ next(\overline{m}<>.\overline{next}<> \mid next().[[N]](x)))$$

なお，$[[N]](x)$ とは，そのライフライン上で続く処理を実施後，x チャネルでメッセージを送出するプロセスである．

このように，ラインフラインの事象に対して定義すると，シーケンス図の各ライフラインを π 計算のプロセスとして意味を定義することができる．結果として，シーケンス図で表現されたインタラクションプロトコルは，次のように形式的な意味を与えることができる．

$$InteractionProtocol \ = \ R_1 \mid \ldots \mid R_n$$

IOM/T の記述に対しても，各ロールごとに，コードの断片に対して形式的意味論を与えることで，各ロールの意味論を *pi* 計算のプロセスとして，定義することができる．また，IOM/T で記述されたインタラクションプロトコルも同様に以下のように定義される．

$$InteractionProtocol \ = \ R_1 \mid \ldots \mid R_n$$

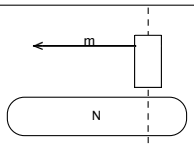
シーケンス図，IOM/T，それぞれの部分構造に対する π 計算による意味論の定義を，表 3.1，表 3.2 に示す．

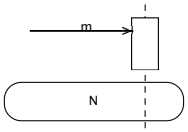
IOM/T のコード，及び，シーケンス図に対して，*pi* 計算による意味論を与えることができれば，*pi* 計算の双模倣等価を検証 [57] することが可能となる．つまり，シーケンス図をもとに，IOM/T のコードを記述した際に，IOM/T のコードがシーケンス図と等価であるか，詳細な実装情報を IOM/T に追加するためにコードを変更した際に，意図せずシーケンス図と異なる修正を行なっていないかを検証することができるのである．

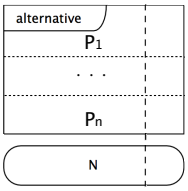
3.4 IOM/T を用いた開発と MAS の実行

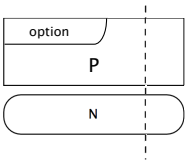
MAS は，一般的にエージェントプラットフォームと呼ばれる基盤フレームワーク上で動作する．IOM/T はインタラクションプロトコルを記述する言語であり，エージェントプラットフォームを提供するものではない．本節では，第 2 章でも扱った，エージェントプラットフォームである JADE の場合を例として，IOM/T により記述したインタラクションプロトコルを用いる場合，開発者は何を作成し，そして，その成果物がエージェントプラットフォーム上で実行される仕組みについて述べる．まず，3.4.1 節において，JADE において MAS を実装した際のクラス構成について述べ，3.4.1 節では，IOM/T を利用した場合は，そのクラス構成において何を開発者が実装するのかを示し，3.4.3 節で，それを実現するための IOM/T のコード変換について述べる．

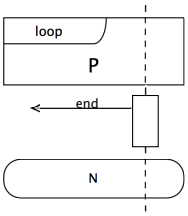
シーケンス図の部分構造	IOM/T の部分構造	形式化

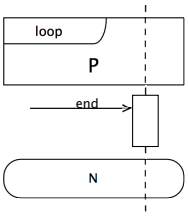
	play R {
	ACLMessage msg;
	sendAsync<m>(msg);
	}
	N
		$A(x) = new\ next(\overline{m}<>.\overline{next}<> \mid next().[[N]](x)))$

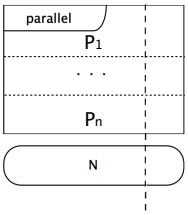
	play R {
	ACLMessage msg = recv<m>();
	}
	N
		$A(x) = new\ next(m().\overline{next}<> \mid next().[[N]](x)))$

	if (...) {
	P1
	} else if (...) {
	...
	} else {
	Pn
	}
	N
		$A(x) = new\ done([[P_1]](done) + \cdots + [[P_n]](done) \mid done().[[N]](x))$

	if (...) {
	P
	}
	N
		$A(x) = new\ done(([[P]](done) + \overline{done}<>) \mid done().[[N]](x))$

	while (S.functionallity()) {
	P
	}
	play S {
	ACLMessage msg;
	sendAsync<end>(msg);
	}
	N
		$A(x) = new\ done, next([[P]](done) \mid (done().A(x) + (done().\overline{end}<>.\overline{next}<> \mid next().[[N]](x))))$

	while (S.functionallity()) {
	P
	}
	N
		$A(x) = new\ done, next([[P]](done) \mid (done().A(x) + (done().end().\overline{next}<> \mid next().[[N]](x))))$

	parallel {
	{ P1 }
	...
	{ Pn }
	}
	N
		$A(x) = new\ done([[P_1]](done) \mid \cdots \mid [[P_n]](done) \mid done().\cdots.done().[[N]](x))$

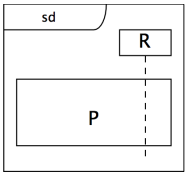
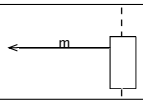
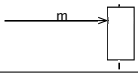
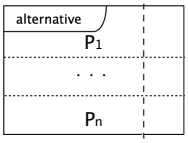
	interaction InteractionName {
	role R { ... }
	...
	protocol {
	P
	}
	}
		$R = new\ done([[P]](done) \mid done())$

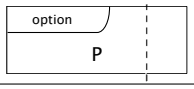
表 3.1: π 計算による形式的意味論の定義 (1/2)

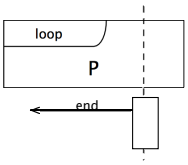
シーケンス図の部分構造	IOM/T の部分構造	形式化

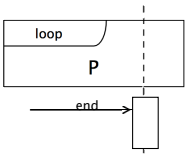
	play R {
	ACLMessage msg;
	sendAsync<m>(msg);
	}
		$A(x) = \overline{m}<>.\overline{x}<>$

	play R {
	ACLMessage msg = recv<m>();
	}
		$A(x) = m().\overline{x}<>$

	if (...) {
	P1
	} else if (...) {
	...
	} else {
	Pn
	}
		$A(x) = [[P_1]](x) + \cdots + [[P_n]](x)$

	if (...) {
	P
	}
		$A(x) = [[P]](x) + \overline{x}<>$

	while (S.functionallity()) {
	P
	}
	play S {
	ACLMessage msg;
	sendAsync<end>(msg);
	}
		$A(x) = new\ done, next([[P]](done) \mid (done().A(x) + (done().\overline{end}<>.\overline{next}<> \mid next().\overline{x}<>)))$

	while (S.functionallity()) {
	P
	}
		$A(x) = new\ done, next([[P]](done) \mid (done().A(x) + (done().end().\overline{next}<> \mid next().\overline{x}<>)))$

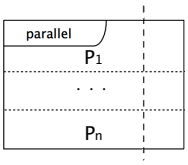
	parallel {
	{ P1 }
	...
	{ Pn }
	}
		$A(x) = new\ done([[P_1]](done) \mid \cdots \mid [[P_n]](done) \mid done().\cdots.done().\overline{x}<>)$

表 3.2: π 計算による形式的意味論の定義 (2/2)

3.4.1　JADE における MAS のクラス構成

JADE においては，図 3.5 に示すように，MAS は複数の Agent クラスのインスタンスにより構成される．また，メッセージの送信・受信をはじめとしたエージェントの動作は，Behaviour クラスのインスタンスとして表現され，Agent クラスのインスタンスには，1 つ以上の Behaviour クラスのインスタンスが関連付けられることによりエージェント，および，MAS が実装される．開発者はこれらのクラスを実装する．

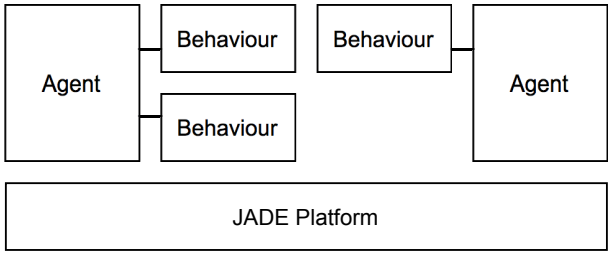


図 3.5: JADE における MAS の構成

3.4.2　IOM/T を利用した開発の概要

IOM/T を利用した開発内容の概要を図 3.6 に示す．まず，開発者は，インタラクションプロトコルを IOM/T で記述する．そして，IOM/T の処理系により，Role と対応したインターフェース，インタラクションプロトコルのための振る舞いが，ロールごとに実装された Behaviour のサブクラスが生成される．次に，開発者はそれに基づいて，Agent クラスのサブクラス，及び，インタラクション以外の振る舞いの Behaviour を実装する．つまり，図中で白塗りで示した，IOM/T によるインタラクションプロトコル，Agent クラス，インタラクション以外の振る舞いのための Behaviour クラスを実装することで，MAS を構築することができる．

3.4.3　IOM/T の変換

我々は，特定のエージェントプラットフォームに依存することを避けたが，エージェントが処理を実行するためには，その基盤となるエージェントプラットフォーム上で動作しなければならない．そこで，IOM/T の記述がそのまま実行されるのではなく，IOM/T の記述を元に，各プラットフォームで動作可能なクラスを生成する．ここでは，JADE の場合を例とするが，Java ベースのエージェントプラットフォームであれば，同様の変換により実現できる．

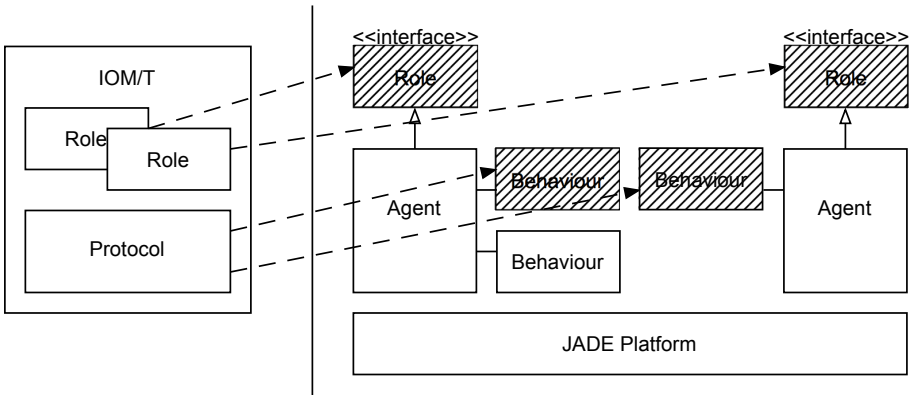


図 3.6: IOM/T を利用した場合の MAS の構成

まず，IOM/T で記述された各ロールに対し，状態遷移モデルを構築する．一つの play 構造を一つの状態として抽出し，ループや条件分岐を考慮して，各状態間に状態遷移を設定する．IOM/T では，一つの play 構造では，メッセージ受信は最大一回であり，メッセージ送信後にメッセージ受信することがないため，一つの play 構造を一つの状態として扱うことが可能である．

例えば，リスト 3.6 のインラクションの実装の場合は，Role1，Role2 の状態遷移モデルは，それぞれ図 3.7，図 3.8 のようになる．

リスト 3.6: IOM/T の変換例

```
1  interaction Sample {
2    role Role1 {
3      boolean isLoopContinue();
4      Agent role2;
5    }
6    role Role2 {
7      Agent role1;
8    }
9    protocol {
10     play Role1 {
11       Message msg;
12       send<m1>(role2, msg);
13     }
14     play Role2 {
15       Message msg = recv<m1>();
16     }
17     while (Role1.isLoopContinue()) {
18       play Role1 {
19         Message msg;
```

```
20          send<m2>(role2 , msg);
21        }
22        play Role2 {
23          Message msg = recv<m2>();
24
25          Message response;
26          send<m3>(role1 , response)
27        }
28        play Role1 {
29          Message msg = recv<m3>();
30        }
31      }
32      play Role1 {
33        Message msg;
34        send<m4>(msg);
35      }
36      play Role2 {
37        Message msg = recv<m4>();
38      }
39    }
40  }
```

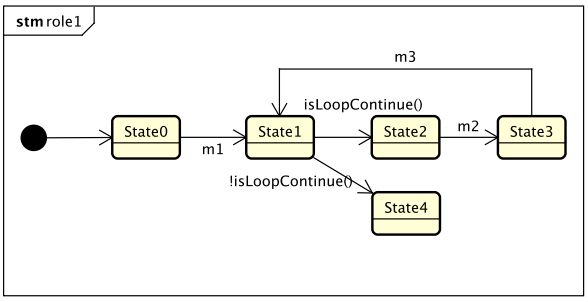


図 3.7: Role1 の状態遷移モデル

次に，各状態の処理のために，private メソッドを作成する．この private メソッドの返り値は，メッセージを送信する状態の場合は void，メッセージを受信する状態の場合は，正しいメッセージを受信したかどうかを示す boolean とする．なお，ここでは簡単のために，ACLMessage クラスの Content として，どのメッセージなのかを指定するようにしている．メソッドの中身は，IOM/T に記述されたものそのままであるが，JADE のメッセージ機構に合わせて書き変えている．その上で，JADE の Behaviour クラスの仕様通りイベントループを action() メソッドで実現し，適切な private メソッドを呼び出すようにする．

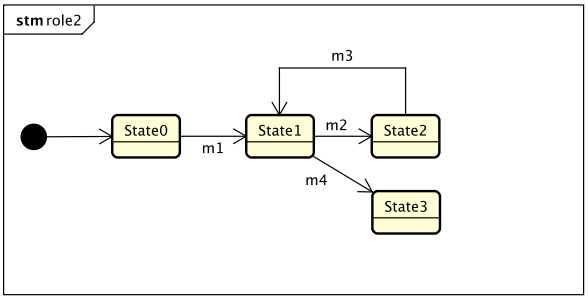


図 3.8: Role2 の状態遷移モデル

ただし，Role2 の state1 の状態のように，自身の Role が判断しないループや条件分岐の先頭の状態は，受け取ったメッセージによって状態遷移が変わるため，特別に扱わなければならない．この場合は，action メソッド中でメッセージを受信し，そのメッセージの内容を判断し，適切な private メソッドを呼ぶようにする．

先ほどの，リスト 3.6 のインラクションの実装の場合は，リスト 3.7，リスト 3.8 のような Behaviour を生成する．

リスト 3.7: 変換された Role1 の Behaviour

```
1  class Role1Behaviour extends Behaviour {
2    private int state_ = 0;
3    private boolean done_ = false;
4    public void action() {
5      switch (state_) {
6        case 0:
7          state0();
8          state_ = 1;
9          break;
10       case 1:
11         if (myAgent.isLoopContinue()) {
12           state_ = 2;
13         } else {
14           state_ = 4;
15         }
16         break;
17       case 2:
18         state2();
19         state_ = 3;
20       case 3:
21         if (state3()) {
22           state = 1;
```

```
23          }
24          break;
25        case 4:
26          state4();
27          done_ = true;
28      }
29    }
30
31    private void state0() {
32      ACLMessage msg = new ACLMessage(UNKNOWN);
33      msg.addSender(myAgent.getAID());
34      msg.addReceiver(role2.getAID());
35      msg.setContent("m1");
36      myAgent.send(msg);
37    }
38    private void state2() {
39      ACLMessage msg = new ACLMessage(UNKNOWN);
40      msg.addSender(myAgent.getAID());
41      msg.addReceiver(role2.getAID());
42      msg.setContent("m2");
43      myAgent.send(msg);
44    }
45    private boolean state3() {
46      ACLMessage msg = myAgent.receive();
47      return msg != null && "m3".equals(msg.getContent());
48    }
49    private void state4() {
50      ACLMessage msg = new ACLMessage(UNKNOWN);
51      msg.addSender(myAgent.getAID());
52      msg.addReceiver(role2.getAID());
53      msg.setContent("m4");
54      myAgent.send(msg);
55    }
56  }
```

リスト 3.8: 変換された Role2 の Behaviour

```
1  class Role1Behaviour extends Behaviour {
2    private int state_ = 0;
3    private boolean done_ = false;
4    public void action() {
```

```
5       switch (state_) {
6         case 0:
7           if (state0()) {
8             state_ = 1;
9           }
10          break;
11        case 1:
12          {
13            ACLMessage msg = myAgent.receive();
14            if (msg == null) {
15              break;
16            }
17            if ("m2".equals(msg.getContent()) {
18              state2(msg);
19              state_ = 1;
20            } else if ("m4".equals(msg.getContent()) {
21              state3(msg);
22              done_ = true;
23            }
24          }
25          break;
26      }
27    }
28
29    private void state0() {
30      ACLMessage msg = myAgent.receive();
31      return msg != null && "m1".equals(msg.getContent());
32    }
33    private boolean state2(ACLMessage msg) {
34      ACLMessage msg = new ACLMessage(UNKNOWN);
35      msg.addSender(myAgent.getAID());
36      msg.addReceiver(role1.getAID());
37      msg.setContent("m3");
38      myAgent.send(msg);
39    }
40    private void state3(ACLMessage msg) {
41      // ...
42    }
43  }
```

以上の手順で，各 Role ごとの Behaviour クラスのサブクラスを生成する．

また，変換処理に注目するため省略したが，来るはずのメッセージが一定時間来なかった場合にはエラーとして処理する，複数エージェントからのメッセージを受信する場合は一定時間待ちその時点までに集ったメッセージを受信したとするなどのコードを生成する．

例えば，受信のタイムアウトを考慮すると全ての受信処理は，リスト 3.9 のようになる．

リスト 3.9: 受信のタイムアウトの制御

```
1  public class Role1Behaviour {
2    ZonedDateTime beginToRecv = null;
3    ...
4    public void action() {
5      switch (state_) {
6      ..
7      case 3:
8        if (beginToRecv = null) {
9          beginToRecv = ZonedDateTime.now();
10       }
11       if (state3()) {
12         state_ = 1;
13       } else {
14         Duration duration = Duration.between(beginToRecv, ZonedDateTime
               .now());
15         if (duration.getSeconds() < 30) {
16           // エラー
17         }
18       }
19       break;
20     ...
21     }
22   }
23   ...
24   private boolean state3() {
25     ACLMessage msg = myAgent.receive();
26     return msg != null && "m3".equals(msg.getContent());
27   }
28   ...
29 }
```

また，複数のメッセージ受信をする状態においては，リスト 3.10 のように，受信開始時刻と受信したメッセージのリストを保持するフィールドを追加し，一定時間メッセージ

を待ち，それまでに受けとったリストを処理するコードを生成する．

リスト 3.10: 複数エージェントからの受信の制御

```
1  public class RecvMultiBehaviour {
2    ZonedDateTime beginToRecv = null;
3    List<ACLMessages> messages = null;
4    ...
5    public void action() {
6      switch (state_) {
7      ...
8      case 3:
9        if (state3()) {
10         state_ = 4;
11       }
12       break;
13     ...
14     }
15   }
16   ...
17   private boolean state3() {
18     if (beginToRecv = null) {
19       beginToRecv = ZonedDateTime.now();
20       messages = new LinkedList<ACLMessage>();
21     }
22     ACLMessage msg = myAgent.recv();
23     messages.add(msg);
24     Duration duration = Duration.between(beginToRecv, ZonedDateTime.now
           ());
25     if (duration.getSeconds() < 30) {
26       return false;
27     }
28
29     // Handle messages...
30
31     return true;
32   }
33 }
```

第4章　IOM/Tの拡張

第3章では，IOM/Tの基本的な機能について述べた．本章では，より効率的にMASを開発するためのIOM/Tの拡張機能について述べる．

4.1節では，契約による設計(Design by Contract)[63]の概念を導入し，複雑になりがちなMASのインタラクションプロトコルにおける妥当性の検証を実施する手法を示す．また，IOM/Tによりインタラクションプロトコルを単一のモジュールとして扱うことが可能となった．それに伴ない，IOM/Tで実装したインタラクションプロトコルが正しく実装されているかを検証する必要がある．そのための，インタラクションプロトコルに対する単体テストの手法を，4.2節で示す．そして，4.3節では，近年のアジャイル開発に代表されるインクリメンタル開発において，IOM/Tを想定した場合の上流工程を含む開発プロセスについて述べる．

4.1　IOM/Tにおける契約による設計

一般的に，エージェントシステムには柔軟さが求められるため，機能の実現に対して，関連するモジュール間の責任が曖昧になりがちである．例えば，オークションを実現するMASにおいて，ビッダーが定時する入札は，現在の最高入札価格以上でなければいけないが，ビッダーが最高入札価格以上の入札をする責任を負っているのか，あるいは，オークションを取り仕切るオークショニアが，最高入札価格以下の入札を拒否する責任を負っているのかなど，明確でない場合が存在する．このような曖昧さを排除するため，IOM/Tに契約による設計の概念に基く拡張を実施した．契約による設計とは，メソッドなどに対して，事前条件，事後条件，不変条件を指定することにより，その対象を利用する際の検証の責任を明確にする手法である．Eiffel[64]には，言語仕様として契約による設計が含まれているが，Javaには言語使用上にはその概念は存在しない．代わりに，JML[56], iContract[51]などのライブラリを利用することで，契約による設計を適用することが可能である．

4.1.1 契約による設計の対象

我々は，インタラクションプロトコルを記述するという IOM/T の特性を考慮し，インタラクション，インタラクションの状態，ロールの振る舞い，ロールの 4 種類の対象に対して契約による設計を可能とした．

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，あるインタラクションを実行する前には，それに参加するロール間では，別の認証インタラクションを実行していなければなら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，while，あるいは，if-else といった制御構造の直前に記述する．それぞれの状態におけるロール，および，ロール間の関係を記述する．

ロールの振る舞い

ロールの振る舞いに対する契約は，play 構造の直前に記述する．各ロールの振る舞いにおいて, 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の直前に記述し，各機能が満たすべき条件を記述する．

4.1.2 契約による設計の記述

IOM/T における契約による設計の記述は，iContract と同様な記法を使い Java のアノテーションの記法により，事前条件，事後条件，不変条件を次のように記述する．

```
/**
 * @pre          事前条件
 * @post         事後条件
```

```
 * @invariant 不変条件
 */
```

また，この条件式には，ロールの機能や変数を使った Java の条件式を記述できる．更に，以下の特殊な記法を利用することができる．

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，⟨*Expl*⟩ 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

⟨*Expr*1⟩ が成り立つならば，⟨*Expr*2⟩ 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．

$$< Expr1 > implies < Expr2 >$$

interact 演算子

⟨$Role_1$⟩，...，⟨$Role_n$⟩ が，⟨*Interaction*⟩ で指定したインタラクションをが実行し,⟨*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．

$$interact < \langle Interaction \rangle > (\langle Role_1 \rangle, \ldots, \langle Role_n \rangle) == \langle Result \rangle$$

また，この際の状態を指定するために，interaction 構造の直前のコメントに@state アノテーションとして記述する．$\langle Expr \rangle$ 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状態 $\langle State \rangle$ で終了したことを示す．

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

4.1.3 契約による設計の検証の実行

Eiffel をはじめとしたオブジェクト指向言語における，契約による設計においては，メソッドの呼び出し元と，呼び出し先との責任を明確にすることが重要な要素であり，この責任を明確にすることで，事前条件，事後条件，不変条件が充足されなかった場合を例外として扱うことが可能であった．また，メソッドの前後で検証することで，例外の判定も可能である．一方，IOM/T における契約による設計の対象は，複数のロールの状態に基づく検証が必要となるため，特定のロールの特定の時点に注目しても，条件の充足の検証を行なうことはできない．

そこで，契約による設計の検証を実施する際には，検証を担当するロールを自動で追加し，3.4 節で述べた MAS の実行の仕組みにおいて，各ロールの状態が遷移する play ブロックの前後において内部状態を検証ロールに通知する処理をはさむことで，各ロールの各時点における状態を集約する．このような仕組みを導入することで，複雑な非同期の状態変化が発生した場合でも，契約による設計の検証を実現することが可能である．

4.2 インタラクションの単体テスト

単体テストとは，ソースコードの個々のモジュールが適切に実装されているかを確認するテストであり，オブジェクト指向言語の場合は，クラス，あるいは，メソッドが正しく実装されているかを検証するテストである．これらのオブジェクト指向言語における単体テストにおいては，オブジェクト間の依存関係，すなわち，メソッド呼び出しに関わる部分の単体テストでは，あるクラスのオブジェクトは特定の条件下で，特定のメッセージを

送信すると，その後，内部の状態や，他のオブジェクトへのメッセージの送信などの外部への振る舞いが，期待した通りに実施されることを確認するものであった．

同様の考えに基づくと，MAS における単体テストは，エージェントがある状態において，特定のメッセージを送信すると，ある振る舞いをすることを確認することで，単体テストを実施することは可能である．しかしながら，エージェントという粒度は，オブジェクトと比較して大きい．また，IOM/T を使うことでインタラクションを一つのモジュールと見なすことが可能となり，各エージェントの内部状態や，各エージェントの自律性に基づく機能とは区別することができるため，我々は IOM/T で実装したインタラクションに対する単体テストを考案した．

4.2.1　インタラクションの単体テストの構造

一般的な単体テストは，AAA(Arrange, Act, Assert) により構成される．JUnit を代表とする xUnit フレームワークにおいては，テスト対象のオブジェクトの状態を設定 (Arrange) し，テスト対象のメソッドを呼び出し (Act)，呼び出し結果，および，呼び出し後のオブジェクトの状態を検証 (Assert) する．

IOM/T で実装したインタラクションプロトコルの単体テストにおいては，それぞれ，次のことを意味する．

IOM/T の単体テストの Arrange

対象とするインタラクションに参加するロールを担うエージェントを指定する．MAS においては，自律性を持ったエージェントがインタラクションすることにおいて，より柔軟なシステムを実現するが，ロールの振る舞いを想定しなければ，インタラクションが意図した通りに動作したのかを検証することはできない．そこで，自律性を極力排除したエージェントを用意することが，Arrange に相当する．

IOM/T の単体テストの Act

Act は，Arrange で用意したエージェントをそれぞれのロールとして実装したインタラクションを実行することである．

IOM/T の単体テストの Assert

Assert では，インタラクションを実施した結果として，各ロールの変数の状態を検証することである．

4.2.2 インタラクションの単体テストの例

ここでは，開発する MAS には，リスト 4.1 に示すようなオークションにより意思決定をする機能が含まれており，そのオークションのインタラクションに対する単体テストを考える．

リスト 4.1: オークションプロトコルの記述例

```
1  interaction Auction {
2    Role Auctioneer {
3      List<Agent> getBidders();
4      Message selectBid(Map<Agent, Bid> bids);
5      ...
6      Agent winner;
7    }
8    Role Bidder {
9      Message createBid(Message cfp);
10     boolean win;
11   }
12   protocol {
13     ...
14   }
15 }
```

このインタラクションの単体テストをするためには，それぞれのロールの集合がテストケースとなる．例えば，最低落札価格が 1000 円の auctioneer と，1100 円までビッドするロール bidder1，1200 円までビッドするロール bidder2 を用意し，これらをテストケースとして見なすのであれば，インタラクションの終了後には，次のことが実現されているはずである．

```
auctioneer.winner = bidder2
bidder1.win == false
bidder2.win == true
```

また，同様の Bidder に対して，最低落札価格が 2000 円の auctioneer をテウトとケースとすれば，この場合は両ビッダーともに最低落札価格を下回るビッドしかしないため，落

札者なしという状態になり，次のことが実現されるはずである．

```
auctioneer.winner = null
bidder1.win == false
bidder2.win == false
```

このような単体テストを用意することにより，将来的な変更の際にも安全に修正を行うことが可能である．

4.3　アジャイル開発における利用

アジャイル開発とは，4 つの価値基準と 12 の原則によって提唱されるアジャイルマニフェスト [11] に従った開発であり，ウォーターフォール型の開発を代表とする従来の開発とは大きく思想が異なる．その一つとして，従来の開発では，初期の段階で全ての要件を確定させ，その要件を適切に実現するための設計を作成する．そして，この計画の達成具合いに応じて，ソフトウェア開発のプロジェクトの成功の可否が決定すると考えられている．この方法は，ソフトウェアを取り巻く環境が変化せず，様々なものが予測可能な場合においては，無駄を排除することが可能であり，非常に有効である．

しかしながら，近年のソフトウェアでは，ビジネスの環境の変化が激しく，ソフトウェアにおいて実現すべき機能が時間と伴に変化していく．そのような場合には，将来的に必要な機能は未知であり，将来を予測して過分に実装するよりも，現時点で必要な機能を素早く実装し，将来的に必要な機能はその時点で実装する方が効率的な場合が多い．

アジャイル開発のプラクティス集である XP[10] では，YAGNI 原則としてこのことが提唱されている．YAGNI 原則とは，"You ain’t gonna need it"の略称であり，ソフトウェアの機能は実際に必要となるまでは，実装しない方がよいとするものである．

ところで，このような YAGNI 原則に従って実装すると，初期の段階で全てを計画する手法と比較して，機能の追加にオーバーヘッドが生じる可能性がある．このオーバーヘッドを最小限に維持することが重要であり，そのためには，各時点での機能の実装の設計は非常に重要な要素である．

しかしながら，適切な設計を実施するためには，高度なスキルと経験が必要となることが多く，経験の浅いエンジニアにとっての大きな障壁の一つである．そこで，我々はインタラクションを中心に考慮することにより，一定の制約の中で適度な自由度を持って設計を実施することが可能となり，経験の浅い開発者であっても，シンプルで将来的な変更にも対応可能な適切な設計 (Simple Design[83]) ができることを目指す．

また，要求から設計，そして，実装まで，インタラクションという概念を一貫して導

入することで，要求から実装までのトレーサビリティの維持を目指す．ユーザストーリ(フィーチャ)と，それを実現するインタラクションが明確な対応があるため，インクリメンタルに開発を実施するアジャイル開発においても，既存のインタラクションを修正するのか，新規のインタラクションの追加をするのかから考えることが可能となる．

4.3.1 アジャイル開発における適用の前提

次に示す設計プロセスのもとで，アジャイル開発における設計にインタラクションを導入するインタラクション駆動設計という手法を確立した．まず，我々が想定する設計プロセスについて定義をするために，設計プロセスについて，SEMAT[47] を用いて記述した．SEMAT では，ソフトウェア開発を実施する際に扱うべき要素をアルファとして定義し，アルファ間の関連性を記述する．また，アルファには取り得る状態が定義されている．我々が定義したアジャイルにおいて各機能を実装するための設計業務は，図 4.1 のアルファから構成され，各アルファはそれぞれ図 4.2 のような状態を持つと定義する．

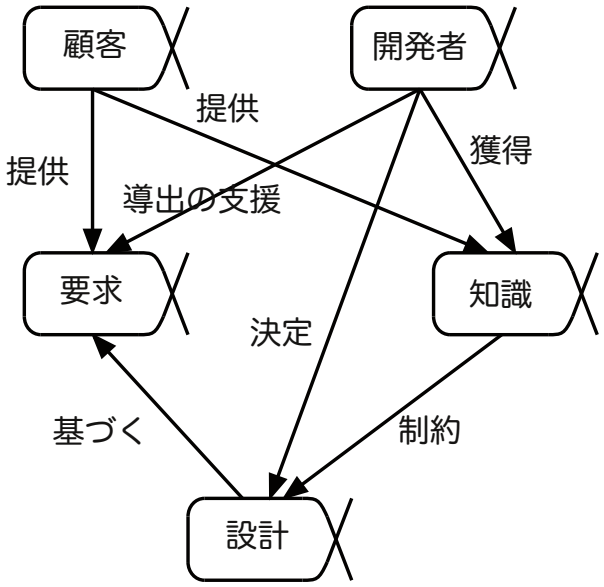


図 4.1: 設計プロセスのアルファ

設計プロセスをこのように定義した上で，本稿の手法の対象は次のように限定する．

- 要件アルファは「計画されている」状態である．
- 知識アルファは「必要な知識の獲得方法が判明している」または「必要な知識を獲得している」状態である．
- 設計アルファは「決定されていない」状態であり，設計プロセス終了後は,「要求を満たすことが検証されている」状態となる．

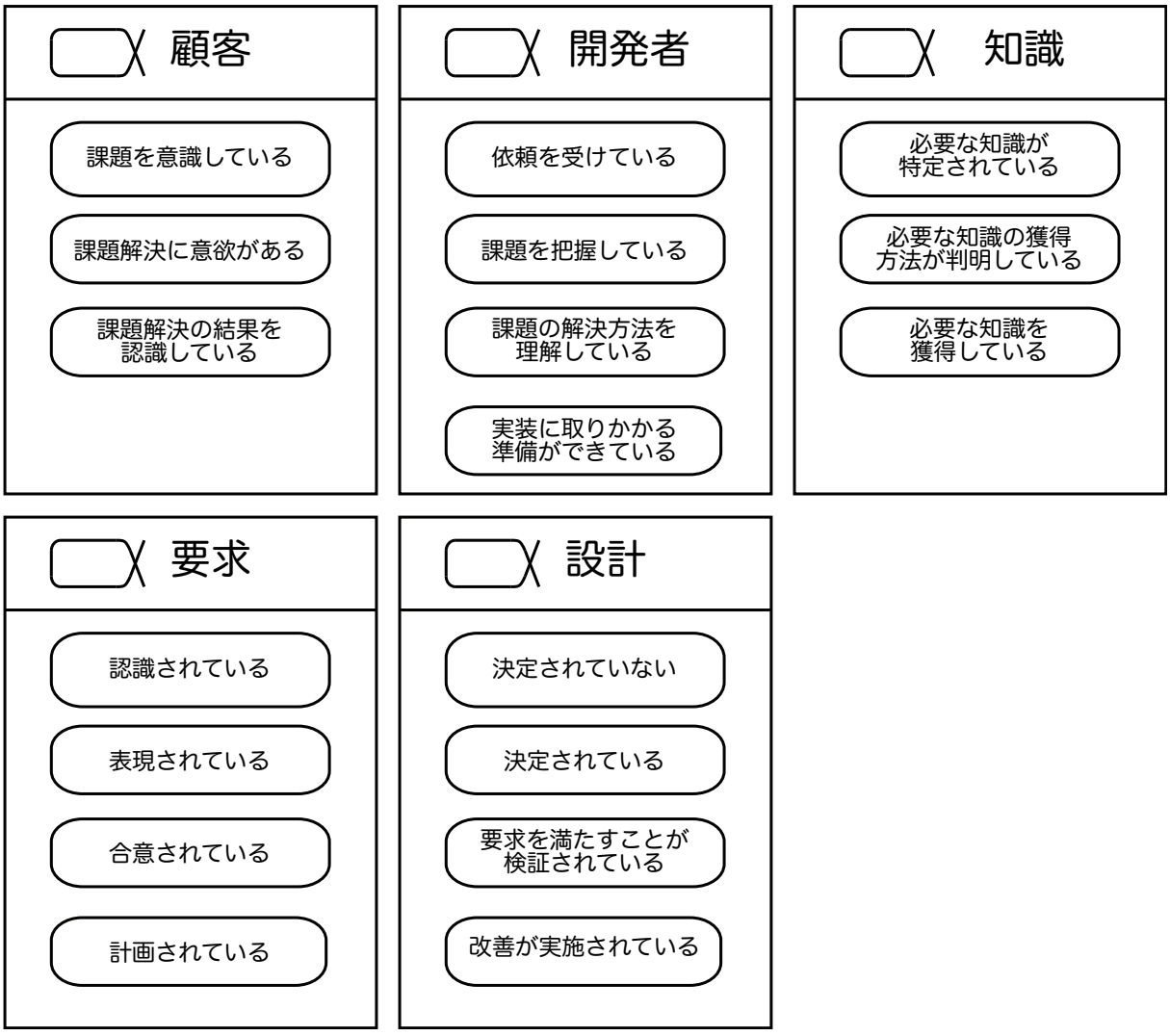


図 4.2: 設計プロセスのアルファの状態

このことは，アジャイル開発を実施する上での標準的なフレームワークである Scrum[81] で言えば，該当機能はプロダクトバックログの上位にあり，現スプリントで実装を予定しており，その機能の実現に必要な業務知識については，既にヒアリング済み，もしくは，必要な際に顧客から情報を取得できる体制がとれており，実現のためのアルゴリズムやライブラリといった技術的な知見はチーム内に存在する状態と言える．

そのため，本研究では，顧客からの要求が曖昧な場合や，非常に大きな技術的課題がある開発は想定していない．

4.3.2 インタラクション駆動設計

4.3.1 節で示した前提において，各要求は，インタラクションにより充足されるとみなし，要求の内容をシーケンス図を詳細化することにより設計していくインタラクション駆動設計 (Interaction Driven Design, IDD) を確立した．以下に，IDD の手順を示す．

1. システムを表わすライフラインを追加し，入力と出力のメッセージを追加する．
2. 各ライフラインについての責務を検討する．
   (a) メッセージに対する責務を明確化し，同粒度の責務に分割できないかを検討する．分割できる場合はそれを列挙する．
   (b) 複数の責務に分割できる場合には，分割した責務ごとにその責務を負うライフラインを作成し，そのライフラインとの間にメッセージを追加する．
3. 各メッセージに対して，メッセージのコンテンツを検討する．
   (a) メッセージにコンテンツを明記する
   (b) 送信者がメッセージのコンテンツを保持しているか確認する．保持していない場合は，必要があればライフラインを追加した上で，コンテンツを取得するためのメッセージのやりとりを追加する．
   (c) 責務を果たす上で情報が足りているか検討する．不足している場合は，受信するメッセージのコンテツを増やすか，その情報を取得する責務を追加するかのいずれかを実施する．
   (d) 責務を果たす上で，コンテンツの形式に変更の必要があるかを検討する．形式の変更が必要な場合には，必要があればライフラインを追加した上で，形式を変更するメッセージのやりとりを追加する．
4. 複数のライフラインの統合を検討する．
5. 各メッセージに対してコンテンツが複数となっているものに対して，他のメッセージでも同一の組み合わせが存在した場合には，ドメインオブジェクトの導入を検討する．

6. 2 から 5 の手順を繰り返す.

手順 2 においては，ライフラインと責務を対応付け，責務の詳細化にのみ集中させる．その上で，手順 3 において，責務遂行のための，データのフローを考えることにより，実現可能性について検証する．この作業を繰り返すことにより，その機能を実現するために必要なロールとそれらの間のメッセージシーケンスを導出する．しかし，これらの手順だけでは，過剰に責務が小さくなったロールが発生してしまう．そのため，手順 4 を実施することにより，小さな関連する責務をともに扱うロールに統合することを検討する．また，主として機能について分析を進めるため，データモデルの構築を実施することができない．そのため，手順 5 で，機能の実現のために，頻繁に発生するデータの組み合わせから新しいデータモデルを抽出することによりドメインモデルを導出する.

4.3.3　インタラクション駆動設計の例

本節では,「文字列にて数式が与えられた際にその数式を計算する」機能をインタラクション駆動設計で実施した例を示す．ここでは知識として，この処理は逆ポーランド記法に変換した上で，順にスタックに積んでいくことで実現できることが，分かっているものとする．なお，紙面の都合上，逆ポーランド記法への変換処理については，責務をこれ以上分割しない.

まず，手順 1 に従って，システムの全体を System ライフラインとして表現すると図 4.3 のようになる.

この System のライフラインは,「入力された数式を計算する」という責務があり，知識として得られた情報から，以下の 4 つに責務が分割することができる.

- 数式を逆ポーランド記法に変換する
- 逆ポーランド記法の文字列をトークンに分割する
- 逆ポーランド記法の各トークンをスタックに積む
- 計算結果を取得する

この責務を踏まえ，手順 3 までを実施し，シーケンス図を洗練すると，図 4.4 となる.

更に，手順 4 で複数のライフラインの統合を検討し，逆ポーランド記法への変換結果は，トークンへの分割の入力以外に利用されていないため，この 2 つのライフラインは統合すべきであると判断し，1 巡目は図 4.5 のようなシーケンス図で終了となる.

2 巡目においては，計算スタックのライフラインの責務を中心に詳細化されて行き，手順 4 まで実施すると，図 4.6 のようになる.

次に手順 5 にて，トークンデータを受け渡すメッセージが多いことを踏まえ，トークン

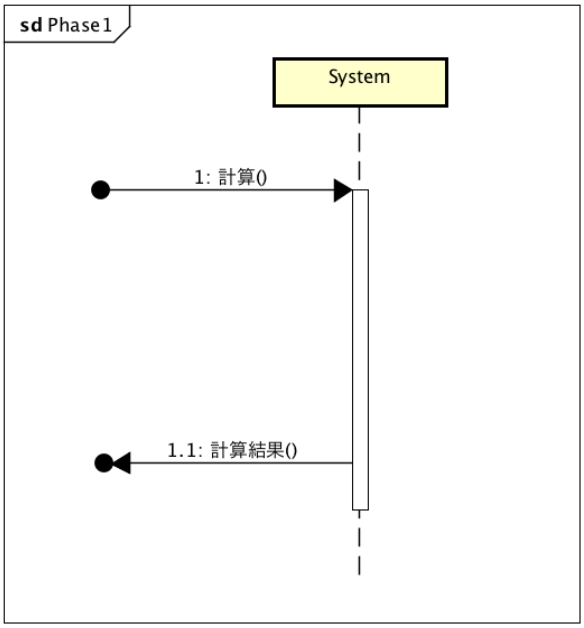


図 4.3: IDD の例 (1/6)

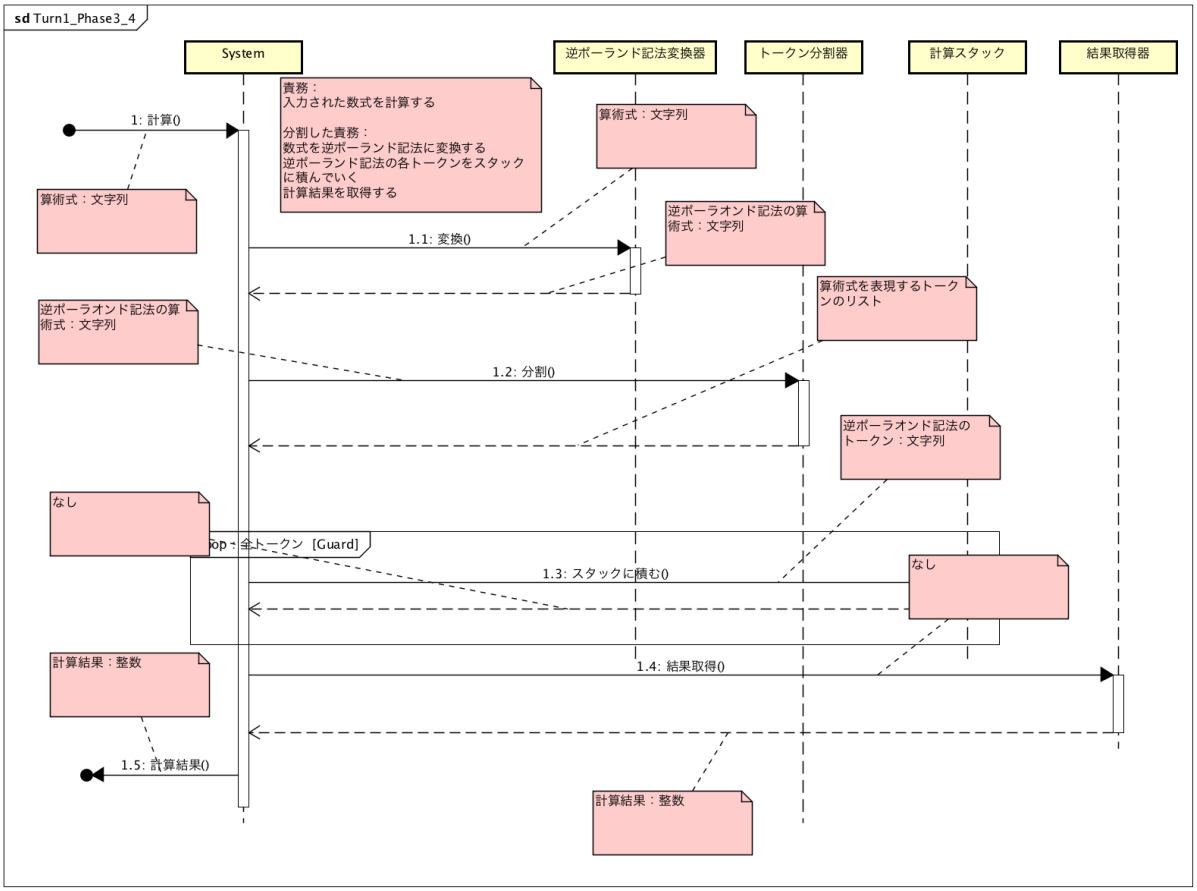


図 4.4: IDD の例 (2/6)

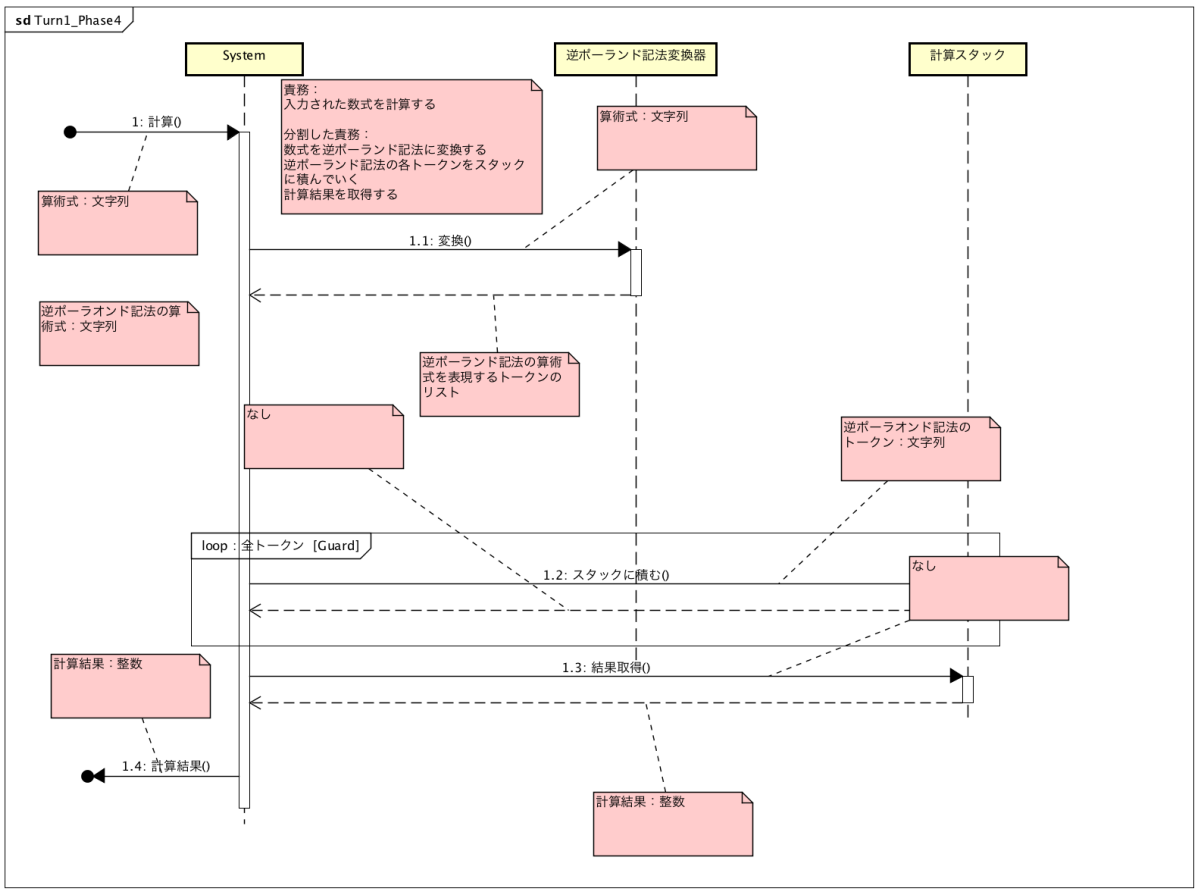

sd Turn1_Phase4
System
逆ポーランド記法変換器
計算スタック
責務：
入力された数式を計算する
分割した責務：
数式を逆ポーランド記法に変換する
逆ポーランド記法の各トークンをスタック
に積んでいく
計算結果を取得する
1: 計算()
算術式：文字列
算術式：文字列
1.1: 変換()
逆ポーラオンド記法の算
術式：文字列
逆ポーランド記法の算術
式を表現するトークンの
リスト
なし
逆ポーラオンド記法の
トークン：文字列
loop：全トークン [Guard]
1.2: スタックに積む()
なし
計算結果：整数
1.3: 結果取得()
1.4: 計算結果()
計算結果：整数

図 4.5: IDD の例 (3/6)

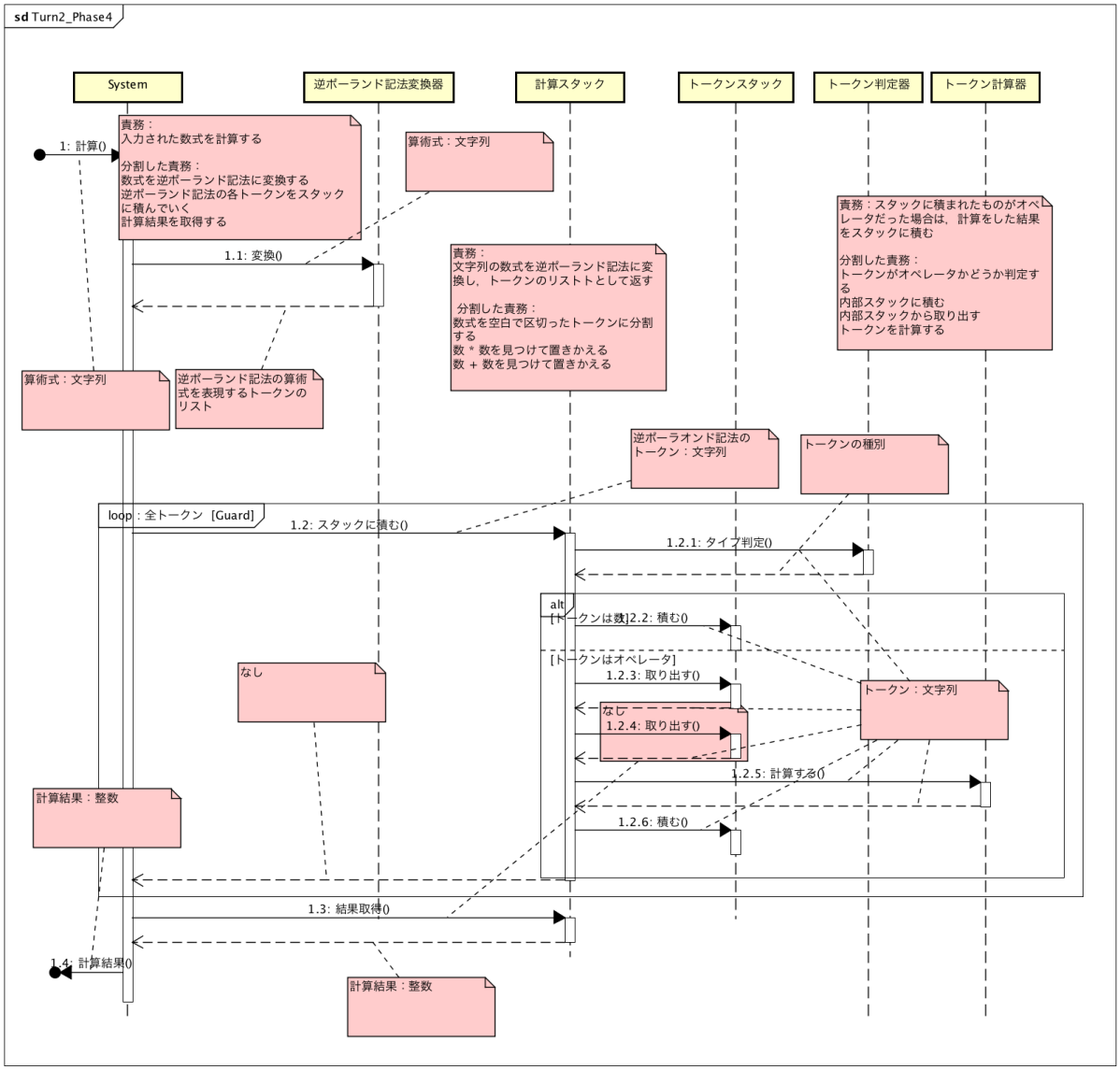

sd Turn2_Phase4
System
逆ポーランド記法変換器
計算スタック
トークンスタック
トークン判定器
トークン計算器
責務：
入力された数式を計算する
分割した責務：
数式を逆ポーランド記法に変換する
逆ポーランド記法の各トークンをスタック
に積んでいく
計算結果を取得する
1: 計算()
算術式：文字列
1.1: 変換()
責務：
文字列の数式を逆ポーランド記法に変
換し，トークンのリストトとして返す
分割した責務：
数式を空白で区切ったトークンに分割
する
数 * 数を見つけて置きかえる
数 + 数を見つけて置きかえる
責務：スタックに積まれたものがオペ
レータだった場合は，計算をした結果
をスタックに積む
分割した責務：
トークンがオペレータかどうか判定す
る
内部スタックに積む
内部スタックから取り出す
トークンを計算する
算術式：文字列
逆ポーランド記法の算術
式を表現するトークンの
リスト
逆ポーラオンド記法の
トークン：文字列
トークンの種別
loop：全トークン [Guard]
1.2: スタックに積む()
1.2.1: タイプ判定()
alt
[トークンは数]2.2: 積む()
[トークンはオペレータ]
1.2.3: 取り出す()
なし
トークン：文字列
なし
1.2.4: 取り出す()
1.2.5: 計算する()
計算結果：整数
1.2.6: 積む()
1.3: 結果取得()
1.4: 計算結果()
計算結果：整数

図 4.6: IDD の例 (4/6)

をドメインオブジェクトとして導入すると，図 4.7 のようになり，2 巡目は終了となる．

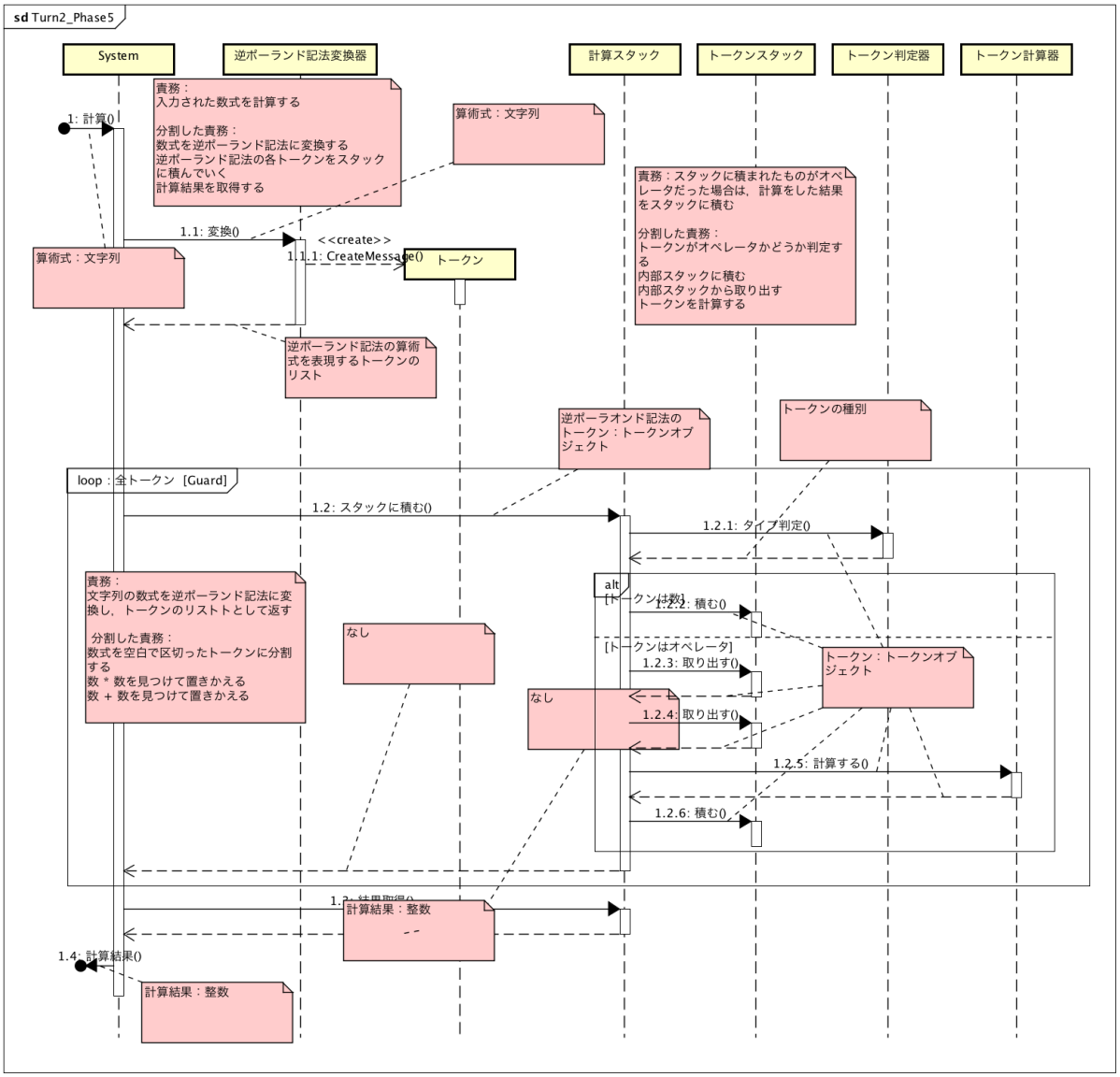


図 4.7: IDD の例 (5/6)

同様の手順を繰り返すことにより，最終的に図 4.8 として設計が完了する．

本章では，IOM/T を中心として MAS を開発する際に，堅牢性を高めるための契約による設計による拡張について述べ，効率よく MAS を開発するためのインタラクションに対する単体テストについて提案し，インタラクションを要求から実装まで導入することにより，インクリメンタル開発の一貫性を保つための IDD について提示した．これらの拡張を踏まえて IOM/T を利用したインタラクションを中心とした MAS 開発により，効率化が期待できる．

今回の実験は被験者の数は多くないため，統計的に価値があると結論ずけることは早計ではあるが，外的要因として特定の思考をさせることは，経験が浅い開発者であっても，

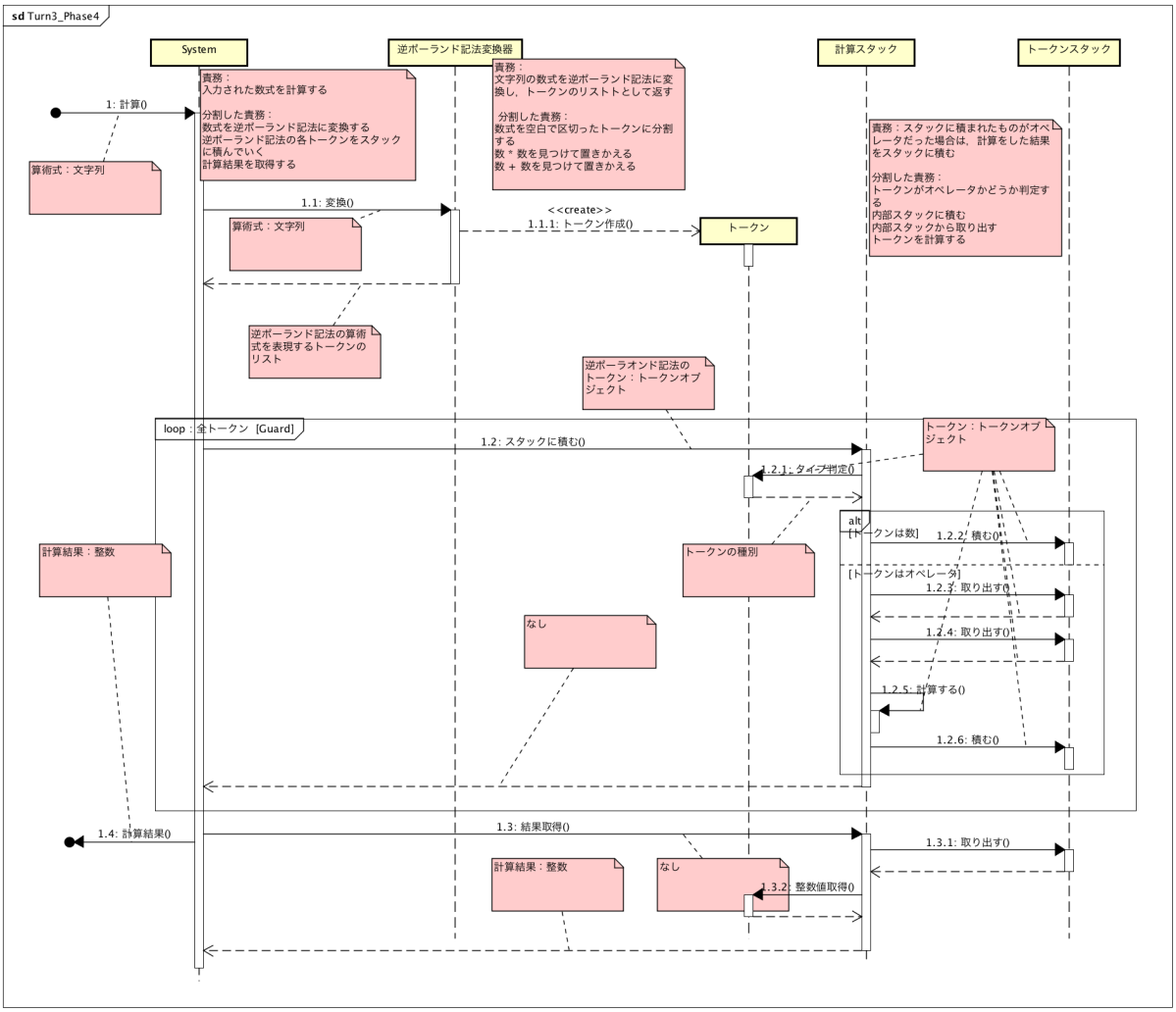

sd Turn3_Phase4
System
逆ポーランド記法変換器
計算スタック
トークンスタック
責務：
入力された数式を計算する
分割した責務：
数式を逆ポーランド記法に変換する
逆ポーランド記法の各トークンをスタックに積んでいく
計算結果を取得する
責務：
文字列の数式を逆ポーランド記法に変換し，トークンのリストとして返す
分割した責務：
数式を空白で区切ったトークンに分割する
数 * 数を見つけて置きかえる
数 + 数を見つけて置きかえる
責務：スタックに積まれたものがオペレータだった場合は，計算をした結果をスタックに積む
分割した責務：
トークンがオペレータかどうか判定する
内部スタックに積む
内部スタックから取り出す
トークンを計算する
1: 計算()
算術式：文字列
1.1: 変換()
算術式：文字列
<<create>>
1.1.1: トークン作成()
トークン
逆ポーランド記法の算術式を表現するトークンのリスト
逆ポーラオンド記法のトークン：トークンオブジェクト
loop：全トークン [Guard]
1.2: スタックに積む()
トークン：トークンオブジェクト
1.2.1: タイプ判定()
alt
[トークンは数]
1.2.2: 積む()
[トークンはオペレータ]
1.2.3: 取り出す()
1.2.4: 取り出す()
1.2.5: 計算する()
1.2.6: 積む()
計算結果：整数
トークンの種別
なし
1.4: 計算結果()
1.3: 結果取得()
1.3.1: 取り出す()
計算結果：整数
なし
1.3.2: 整数値取得()

図 4.8: IDD の例 (6/6)

熟練者の思考の一端を捉えることが可能になるのではないか.

第5章　評価と考察

本章では，第3章，第4章で示した，インタラクション記述言語IOM/Tを中心とした開発手法について，その有効性について議論する．

5.1　IOM/Tによる実装の効果

本節では，IOM/Tによる実装の効果について考察する．まず，5.1.1節で，IOM/Tでサポートしていない構造について示し，その影響度について議論する．その上で，5.1.2節にて，従来のエージェント記述言語のインタラクションの記述と具体的に比較することにより，どのような相違が生まれるかを示し，IOM/Tにより第2章で示した課題が解決できるのかを検証する．

5.1.1　IOM/Tで表現できるインタラクションプロトコルの制約

3.2節で述べたように，IOM/Tは，いかなるインタラクションをも記述できるものではない．例えば，図5.1のようなシーケンス図を想定していない．

このシーケンス図は，シーケンス図として正しいが，システムとして考えた場合に，複合フラグメントのどちらが実行されるかは，複数のロールが同一の認識をする必要がある．この判断がどちらかのロールの内部状態に基づくものであれば，他方のロールにはメッセージのやり取りなしには伝達できず，それぞれのロールが別な複合フラグメントの処理を実施してしまう可能性がある．また，どちらのロールにも属さない外的要因に基づく判断だったとしても，それぞれのロールが判断するタイミングを保証することはできず，やはり，MAS内の処理に不整合が発生する可能性がある．IOM/Tは，3.1節で述べた性質のMASを対象とするため，このようなインタラクションプロトコルは対象としていない．

そのため，while構造の条件は単一のロールにより判断でき，ループの最初のメッセージ及び，ループ終了後，最後のメッセージは同一のロール間のメッセージの送受信でなければならない．if構造の条件も単一のロールにより判断でき，各分岐において，最初の

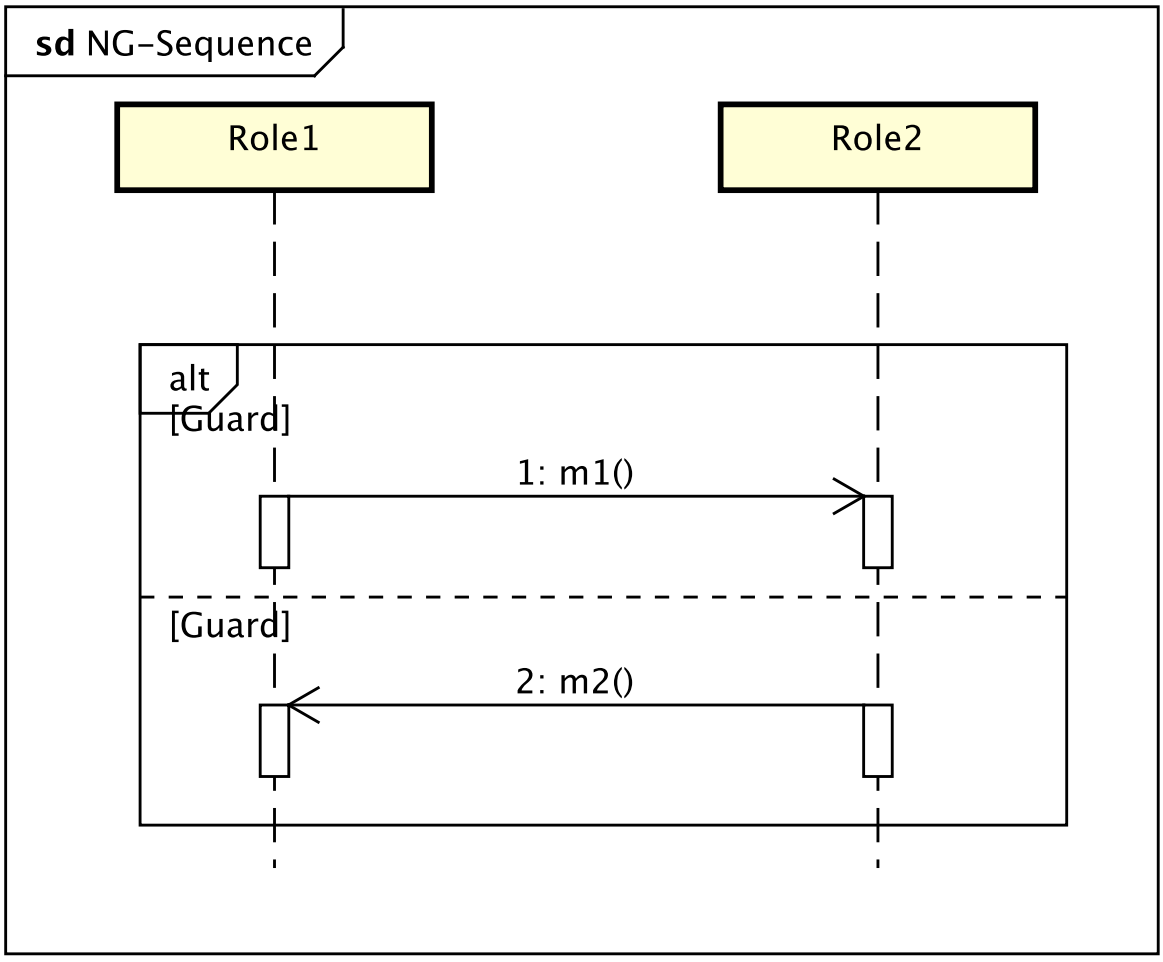


図 5.1: IOM/T が想定しないシーケンス図

メッセージは，条件を判断したロールからのメッセージ送信でなければならない．といった制約は，妥当である．

また，シーケンス図の複合フラグメントには，IOM/T と対応がある alt，loop，opt，par 以外にも，break，critical，assert，neg，ignore，consider などが規定されている．これらの複合フラグメントについても，同様の判断から一般的なインタラクションプロトコルの定義においては，対応している，alt，loop，opt，par により表現できるとの観点から除外している．

5.1.2 従来のエージェント記述言語との比較

図 3.4 のインララクションを JADE で記述した場合を考える．実装は，Role1，Role2 の 2 つに分割され，それぞれ，リスト 5.1，リスト 5.2 のような記述となる．

リスト 5.1: やや複雑なインタラクションの JADE における Role1 の記述例

```
1  public class Role1Behaviour extends jade.core.behaviours.Behaviour {
2    int state_ = 0;
3    boolean isTerminate_ = true;
4    public void action() {
5      switch (state_) {
6      case 0:
7        if (/* is first loop continue? */) {
8          state_ = 1;
9        } else {
10         state_ = 7;
11       }
12       break;
13     case 1:
14       /* send message m1 */
15       state_ = 2;
16       break;
17     case 2:
18       /* recv message m2 or m6 */
19       if (/* is first case? */) {
20         state_ = 3;
21       } else {
22         state_ = 6;
23       }
24       break;
```

```
25      case 3:
26        if (/* is second loop continue? */) {
27          state_ = 4;
28        } else {
29          state_ = 5;
30        }
31        break;
32      case 4:
33        /* send message m3 */
34        // recv message m4
35        ACLMessage msg = myAgent.receive();
36        Object content = msg.getContent();
37        /* handle the content */
38        state_ = 3;
39        break;
40      case 5:
41        /* send message m5 */
42        break;
43      case 6:
44        state_ = 0;
45        break;
46      case 7:
47        /* send message m7 */
48        isTerminate_ = true;
49        break;
50      }
51    }
52    public boolean done() {
53      return isTerminate_;
54    }
55  }
```

リスト 5.2: やや複雑なインタラクションの JADE における Role2 の記述例

```
1  public class Role2Behaviour extends jade.core.behaviours.Behaviour {
2    int state_ = 0;
3    boolean isTerminate_ = true;
4    public void action() {
5      switch (state_) {
6      case 0:
7        /* recv message m1 or m7 */
```

```
8       if (/* is first loop continue? */) {
9         state_ = 1;
10      } else {
11        state_ = 7;
12      }
13      break;
14    case 1:
15      if (/* is first case? */) {
16        state_ = 2;
17      } else {
18        state_ = 6;
19      }
20      break;
21    case 2:
22      /* send message m2 */
23      state_ = 3;
24      break;
25    case 3:
26      /* recv message m3 of m5 */
27      if (/* is second loop continue? */) {
28        state_ = 4;
29      } else {
30        state_ = 5;
31      }
32      break;
33    case 4:
34      // send message m4
35      ACLMessage msg = new ACLMessage();
36      /* create the content of message */
37      msg.setContent(...);
38      myAgent.send(msg);
39      state_ = 3;
40      break;
41    case 5:
42      state_ = 0;
43      break;
44    case 6:
45      /* send message m6 */
46      state_ = 0;
47      break;
```

```
48      case 7:
49        isTerminate_ = true;
50        break;
51      }
52    }
53    public boolean done() {
54      return isTerminate_;
55    }
56  }
```

例えば，メッセージ m2 に注目すると，このメッセージの送受信は，リスト 5.2 の 18 行目付近と，リスト 5.1 の 22 行目付近に記述される．これらの箇所は非常に密接な関連を持っているにも関わらず，別なコードに実装されていることとなる．また，m2 のコンテンツを変更した場合には，送信側も受信側も修正する必要があるが，これらの対応を明確に示すものは何もない．一方，IOM ／ T においては，インタラクションは一つのコードとして記述され，AUML のシーケンス図と構造的に対応が存在する．そのため，メッセージの送受信に関しての対応が把握しやすい．例えば，15 行目のメッセージ送信との 18 行目のメッセージ受信，26 行目のメッセージ送信との 31 行目のメッセージ送信とが対応している．メッセージ送受信がエージェントごとに分散し，明示的な状態遷移の記述であった既存言語の記述と比較して，容易に対応が把握できる．そのため，メッセージに対する修正が発生した際に，送信側と受信側の対応が把握でき，修正の影響範囲も特定できるため，保守性が向上する．43 行目のメッセージの内容の作成する部分に変更が生じたとすると，それのメッセージを受信し処理する個所にも影響があるが，その個所が 51 行目であることは，容易に特定できる．

また，この例のインタラクションプロトコルには，2 重のループが存在するが，リスト 5.1，リスト 5.2 のコードだけを見て，そのループを認識することは容易ではない．Role1 の場合であれば，state_変数の値が，0 から始まり，内部の分岐や内側のループによって，1 から 6 までの値に順次変わっていき，また，0 に戻ることから，ループが表現されているが，switch 文の各ブロックに意味が定義されている訳ではなく，結果としてループ構造が実装されているだけであり，保守性が高いとは言えない．IOM/T で記述したリスト 3.5 では，12 行目からの while ブロックで明示的なループ構造が示されている．そのため，ソースコードからもインタラクション全体の構造が把握できる．また，制御構造が大きく複雑になってくると，コードが分散している JADE そのものの実装よりは可読性が高いかもしれないが，IOM/T のコードでも把握が困難になってくることが想像される．しかしながら，IOM/T によるインタラクションの実装は，π 計算によりシーケンス図と等価性を検証

できるため，IOM/T のコードをもとに，シーケンス図を生成することも可能である．よって，IOM/T をシーケンス図として可視化することにより，インタラクションの全体像を把握し，その判断に基づき，インタラクションの特定の部分に注目する際には，ソースコードとしてより詳細な実装情報まで把握するということが可能となる．

5.2 IOM/T における契約による設計の効果

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 MAS は Java を用いて実装される場合が多い．また，4.1 節で述べたように，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 MAS を開発することは可能である．ここでは，English Auction を例に比較する．IOM/T により English Auction を実装した例をリスト 5.3 に示す．

リスト 5.3: English Auction における契約による設計の記述例

```
1  /**
2   * @pre forall Auctioneer a | forall Bidder b | interact<Authorize>(a,
        b) == success
3   * @state success forall Auctioneer a | a.winner != null
4   */
5  interaction EnglishAuction {
6    Role Auctioneer {
7      ...
8      int price;
9      AID winner;
10     ...
11   }
12   Role Bidder {
13     ...
14     /**
15      * @post return.price > current.price
16      */
17     Bid createBid(Bid current);
18     boolean isWin;
19     int price;
20     ...
21   }
```

```
22    /**
23     * @post forall Auctioneer a a.winner != null implies exists Bidder b
          | b.isWin
24     * @post forall Auctioneer a | forall Bidder b | a.winner != null &&
          b.isWin implies a.winner == b && a.price == b.price
25     */
26    protocol {
27      ...
28    }
29  }
```

同様のインタラクションをJADEのエージェントを直接実装したとすると，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，Auctioneerロールと Bidderロールの Behaviorはそれぞれ，リスト5.4，リスト5.5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．

リスト 5.4: JADEにおけるAuctioneerに対する契約による設計の記述例

```
1  public class Auctioneer {
2    ...
3    private int state;
4    private boolean isEnd = false; ...
5    private Vector targetList; private Vector authorizedList; ...
6    private Bidder winneer = null; private int price;
7    ...
8    public void action() {
9      switch (state) {
10       case 0:
11       doState0();
12       break;
13       ...
14       case N: // last state doStateN();
15       isEnd = true;
16       break;
17     }
18   }
19   /**
20    * @pre for all Bidder b1 in targetList.elements() | exists Bidder b2
         in authorizedList.elements() | b1 = b2
21    */
```

```
22    private void doState0() { ... }
23    ...
24
25    /**
26     * @post this.winner != null implies winner.isWin
27     * @post this.winner != null implies this.price = winner.price
28     */
29    private void doStateN() { ... }
30  }
```

リスト 5.5: JADE における Bidder の契約による設計の記述例

```
1   public class Bidder { ...
2     private int state;
3     private boolean isEnd = false; ...
4     private boolean isAuthorized; private Auctioneer auctioneer; ...
5     public void action() {
6       switch (state) {
7         case 0:
8           doState0();
9           break; ...
10        case M:
11          doStateM();
12          break;
13          ...
14        case N: // last state
15          doStateN();
16          isEnd = true;
17          break;
18       }
19    }
20    /**
21     * @pre this.isAuthorized
22     */
23    private void doState0() {
24      ...
25      auctioneer = /* get auctioneer object */
26      ...
27    }
28    ...
29    private void doStateM() {
```

```
30      ...
31      Bid bid = ((BidderAgent)myAgent).createBid();
32      ...
33    }
34    /**
35     * @post this.isWin implies auctioneer.winner == this;
36     * @post this.isWin implies auctioneer.price == this.price
37     */
38    private void doStateN() { ... }
39  }
```

まず，リスト 5.3 の 2-3 行目で記述しているインタラクション間の契約は，リスト 5.4 の 25-29 行目と，リスト 5.5 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，リスト 5.4 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，リスト 5.5 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，リスト 5.3 の 25-30 行目で記述しているエージェント間の契約は，リスト 5.4 の 34-36 行目と，リスト 5.5 の 41-44 行目として記述されている．MAS 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．リスト 5.4 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，リスト 5.5 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．これは，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ず，また，iContract や JML はクラスという単位において契約を記述するためである．

以上のことより，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と言える．

- 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．

5.3 インタラクションの単体テスト

これまでの MAS の記述言語と比較して，IOM/T の最も大きな違いは，複数のエージェントにまたがるインタラクションを一つのソフトウェアコンポーネントとして扱うことを可能したことである．

そのため，従来であれば，インタラクションに関わる単体テストを実施する場合は，あるエージェントがある状況下に応じて，ある対象に対して，あるメッセージを送信する．というテストしか記述することができなかった．しかし，インタラクションという観点から言えば，送信者と受信者のデータフォーマットが共通であることなどもテストしたい項目でありながら，送信者が適切に送信する，受信者が受信したものに対して適切に処理するという 2 つのテストにより構成されてしまい，この 2 つが一貫性がとれているかどうかはテストすることができず，これらのテストは結合テストで検証されていた．

IOM/T により，インタラクションプロトコルをソフトウェアコンポーネントとしてとらえ，テストケースとして，そのインタラクションに参加するロールを設定すると，インタラクションにとっての意図が達成されるかどうかを検証することが可能となった．

テスト可能となるタイミング，つまり，不具合が不具合として検出されるタイミングは

データ a，b，c が格納された csv ファイルを読み込み，a と c および，b と c のデータにおいて線形回帰を実施し，a と b どちらが c のデータを予測するのに適しているかを判定する．なお，今回の線形回帰の計算には Apache Commons Mathematics Library の SimpleRegression を利用することとする．また，今後，データの種類や，回帰モデルの変更などが想定される．

図 5.2: 実験で利用した課題 (要約)

遅れれば遅れるほど，その修正に要するコストは増大するため，インタラクションプロトコルの実装を単体テストとして実施可能とすることには大きな価値があると考える．

ただし，オブジェクトの単体テストと比べると，関連するロールの機能を簡易実装したエージェントが必要であり，その準備に要するコストは大きくなってしまう．しかしながら，契約による設計の記述と併用することにより，このコストをかなり低減できる可能性がある．つまり，ロールの振る舞いに対する契約による設計の記述を解釈することにより，テストケースとなるエージェントを半自動生成することが可能となる．実際，形式的な手法と単体テストの融合を狙った研究 [46] もなされている．

5.4　IDD の適用可能性

IDD は比較的経験の浅い開発者が，ユーザストーリーごとにインクリメンタル開発を行なう際の設計を支援するものである．我々は，IDD がこの目的を達成しているかを判断するために，以下の 2 点を検証するための実験を実施した．

- 経験の浅い開発者でも IDD を利用できるか？
- IDD を利用することで，アジャイル開発に適した Simple Design に近づくか？

実験では，図 5.2 に示すプログラムの作成課題を提示し，インタラクション駆動設計を利用した場合 (グループ A) と，しなかった場合 (グループ B) の差異を確認した．また，実験は，新入社員を中心としたソフトウェア会社の若手社員 6 名 (実験 1) に対してと，クラウドソーシングサービス Lancers に登録している Java の開発経験が 7 年〜10 年の開発者 6 名 (実験 2) に対して実施した．なお，いづれの実験においても，無作為に 3 名ずつグループ A とグループ B に分けて実施した．

5.4.1 実験 1: 経験の浅い開発者

経験の浅い開発者を対象とした実験 1 では，時間の制約上，作業時間の上限を 8 時間として依頼した．また，本実験においては，実装を行う上でのスキルが十分でない被験者も含まれるため，開発環境のトラブルなど，設計や実装に関わらない部分については，支援を行ないながら実験を行なった．

結果として，被験者のうち課題を完了できたのは，6 人のうちグループ B の 2 人のみであった．また，グループ A の 3 人はプログラムの実装は間に合わなかったが，インタラクション駆動設計によるシーケンス図による設計を完了することはできた．

完了済みの部分だけではあるが，その成果物を見てみると，グループ A の被験者は，3 名ともそれぞれ異なるシーケンス図を作成していたが，全員が線形回帰の結果とその誤差を計算するクラスを導入するなどの共通点が見られた．グループ B では，単一クラスにの単一の長いメソッドとして実装している被験者，将来的な拡張を考慮してデータオブジェクトとそのファクトリクラスを導入するといった冗長な構成の実装をする被験者がいた．

プログラム完成の達成率は，実装にかけることができた時間の差であると思われる．両グループともに十分な Java の実装力がなく，また，設計については経験がない状態であったため，グループ A の開発者は IDD の適用に時間を多く消費してしまい，実装に充てる時間が足りなくなったと考えられる．

グループ A の開発者が作成した設計に注目してみると，IDD を通して，明確に責務の分離を意識されていると感じられた．全開発者が実装を完了できていないため，同一条件でのメトリクスの測定はできなかったが，グループ B の実装と比較して，小さなクラスから全体が構成されており，保守性の高い設計になっていた．特に責務の分割が，知識として実験課題で提示されたものに関しては，3 人とも同様に責務を分割し，同様の構造となっていた．一方で，知識として明確に分割方法が与えられなかった部分については，個々人の判断により，結果が異なり，場合により保守性の低い部分も発生していた．つまり，責務を分割するための知識があれば，比較的経験の浅い開発者であっても，IDD に従うことにより，妥当な設計に至る可能性が高いと推定される．

次に，責務を分割するための知識があるという仮定について考えてみる．従来のウォーターフォール型の開発であれば，これらの情報は顧客との接点のある上位の SE や，ドメイン知識や経験の豊富なアーキテクトに集中しがちであり，開発チームの全員が保持していることは稀であると考える．しかしながら，アジャイル開発においては，妥当な仮定である．なぜなら，例えば，Scrum においては，プロダクトバックログ・リファインメントでは，顧客やプロダクトオーナーが持つ知識を共有することができ，また，スプリント計画ミーティング，デイリースクラムなどを通して，開発者間での技術的な判断も共有する

表 5.1: 計測メトリクス一覧

LOC	コード行数 (Lines Of Code). 古典的なメトリクスの一つであり，プログラムの規模の推定に利用される.
# Classes	プログラムを構成するクラス数．これもプログラムの規模を示唆する指標となる.
LCOM4	Lack of Cohesion Of Methods version 4．凝集度を表す指標．1 であることが最適であり，2 以上であった場合は，クラスが複数の責務を負っていることを意味する.
CC	循環的複雑度 (Cyclomatic Complexity)．このメソッド (クラス) の複雑さを表す指標であり，この指標が大きくなるほど複雑さが増していることを意味する.

表 5.2: グループ A のメトリクス

	LOC	# Classes	LCOM4	CC
A-1	155	4	1	6.625
A-2	238	5	1	3.167
A-3	296	6	1.167	4.452
平均	230.0	5.0	1.056	4.748

ことができる．更に言えば，コミュニケーションを重視するため，経験が浅い開発者が判断のための知識を保有していなかった場合でも，アジャイル開発を進める上で，その知識は獲得できるものであると信じる.

以上のことから，IDD はアジャイル開発においては，経験の浅い開発者でも利用可能であり，IDD を利用することで，設計の品質を向上させることが期待できる.

5.4.2　実験 2: 経験のある開発者

ある程度の経験のある開発者を対象とした実験 2 では，クラウドソーシングで依頼しているという条件のため，実際の作業時間は計測できていないが，依頼後，7 日以内に開発が完了した.

開発者の実装したプログラムに対して，凝集度を中心とした表 5.1 に示すメトリクスを計測した．グループ A に対する結果を，表 5.2 に，グループ B に対する結果を，表 5.3 に示す.

こちらの実験においては，グループ A，B の結果のメトリクスに特徴的な違いが見られた．まず，構成されるクラス数は，グループ A が平均 5 クラスであるのに対し，グループ

表 5.3: グループ B のメトリクス

	LOC	# Classes	LCOM4	CC
B-1	53	1	1	4.333
B-2	77	1	2	5.25
B-3	77	1	1	6.75
平均	69.0	1.0	1.333	5.444

B は，3 名とも 1 クラスで実装している．また，コードの行数はグループ A の方が 4 倍ほど増えているものの，循環的複雑度はグループ A の方が小さくなっている．また，LCOM4 もグループ A の方が，1 に近い．このことが意味することは，実験 1 と同様にクラスの責務とその分割を明確に意識することにより，クラスを分けることができ，また，責務という観点からクラスを分けることにより，凝集度の高い小さなクラスの組み合わせによって実現される構造になったと言える．また，各クラスの責務を小さくすることにより，それぞれのクラスのメソッドの複雑さも小さくなり，循環的複雑度も小さくなっていると考えられる．

今回のプログラムはそれほど大きなものではないため，将来的な変更に対しての大きな問題はないが，より大きなシステムで同様の状態を考えると，将来的の修正のコストは異なってくる．例えば，回帰モデルのアルゴリズムを変えたことを想定した場合，グループ B の実装では，単一クラスではあるもののプログラム全体への影響を与える可能性のある修正が必要となる．一方，グループ A の形式の実装であれば，回帰モデルを計算するクラス部分の修正で対応が可能となり，その影響範囲は大きく異なる．

以上のことから，IDD を適用することで，凝集度の高い，将来的な変更のコストを抑えられる設計になっていると言える．別の言い方をすれば，IDD を適用することで，アジャイル開発における Simple Design に近づくと言える．

第6章　関連研究

本研究は，インラクション記述言語 IOM/T の提案第 3 章を中心として，設計と実装の隔りを埋め，MAS の開発の効率化を目指すものである．また，4.1 節で示した IOM/T における契約による設計，4.2 節で示した IOM/T で実装したインタラクションの単体テストは，MAS の検証，テスティングについて扱ったものである．そして，4.3 節では，IOM/T を使い MAS を開発する前提の上で，アジャイル開発を適用する際の取り組みである．

以降，6.1 節では，設計と実装を埋めるという観点から，6.2 節では，検証，テスティングの観点から，6.3 節では，類似の観点を持つ Web サービスの観点から，最後に，6.4 節では，アジャイル開発の設計という観点から，関連研究について触れると共に，本研究と比較する．

6.1　設計と実装の隔りに関する関連研究

6.1.1　MAS における設計と実装の隔り

本研究は，エージェント指向ソフトウェア工学 (AOSE) と呼ばれる領域の研究である．エージェント，MAS によるソフトウェア開発の手法に対する研究であり，エージェント，MAS のための，プログラミング言語，モデリング手法，フレームワーク，アーキテクチャなどが提案されている．

特に設計と実装の隔りを課題とした研究はこれまでにも実施されており，設計記述を改良することにより解決を目指すアプローチと，実装記述を改良することにより解決を目指すものが存在する．

設計記述の改良を目指すものとしては，設計モデルに情報を付加し，設計をそのまま実行することを目指したもの [33] [26] や，UML で記述した設計モデルから，MAS のスケルトンコードを生成することにより，設計と実装の隔りを減らすことを目指したものなどがある．スケルトンコードを生成する研究としては，Agent UML と有限オートマトンを用いて，IBM の ABLE[14] で利用される cpXML(Conversation Policy XML) の記述を JADE のソースコードへ変換するグラフィカルツール [30]，グラフィカルなツールにより，

Bee-gent[49] のソースコードを生成する IPEditor[86] などがある．本研究と最も関係が深いのは，同様にインタラクションに注目し，Agent UML のシーケンス図からスケルコードを生成する研究 [43] である．これらのスケルトンコードを生成する手法は，設計から実装に向けての隔たりを埋めるために有効な手法ではあるが，ソースコードから設計に還元することが難しい．その点，IOM/T はシーケンス図と等価な記述力があり，双方向に変換が可能である．

実装記述の改良によるアプローチとしては，BDI エージェント [75] に代表される手続き的ではない実装記述を可能にすることにより，実装を設計に近づけるもの [18] [72] が存在する．本研究も後者のアプローチを取るものであるが，インタラクションに注目し，インタラクションを一つのモジュールとして扱う点，また，シーケンス図と等価な記述を保証することが実現できる点が本研究の特徴である．

一方，エージェントのモデルとしては，BDI モデル [38] などの人間の思考を表現しようとするものが有望しされるようになってきていた．そのような中で，2007 年以降は意思決定やシミュレーション，自己適用などをいかに実現するかが，AOSE の中心的な関心事となり，それ以外の領域に対する研究は少なくなっていた．

ところが，IoT という言葉が代表する環境の変化や，マルチプロセッサ，マルチコアといった並列計算基盤の普及により，2011 年頃より，エージェントという人間を代行するような概念から離れ，より一般的に分散オブジェクトによる並列計算のための研究として，2007 年頃までの研究領域が再び，注目を集めるようになってきている [78]．そして，本研究と関連が深い研究としては，複数のネットワーク基盤を通じてインタラクションを実施するモバイルアプリケーションのためのライブラリ [15] が提案されるなど，インタラクションの実装を容易にするための研究は注目を集めている．なお，インタラクションを一つのモジュールとして実装するというコンセプトは，現時点でも類を見ないものであり，本研究の特に斬新な点である．

また，エージェントに関するプログラミング言語は多数提案されている．3APL[27] は，ゴール指向の概念に基づくプログラミング言語である．シナリオ記述言語 Q[45] は，エージェントの内部の記述ではなく，エージェント間のインタラクションを設計するための言語である．同様に，インタラクションに注目した言語として，AgenTalk[52]，COOL[8]，April[62] などが提案されている．MAS の動作の確認，検証は，本研究の大きな対象であり，モビリティ，エージェント間の会話や協調などを対象とし，AgentSpeak(L)[74]，CLAIM[5]，AF-APL[79] また，エージェント間のインタクションを検証する Java-Prolog コンポーネント [3] や，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解析し，不具合の発生を検知する [53] などの手法も開発されている．近年

は，新しいパラダイムを表現するためにプログラミング言語やフレームワークを提案することよりも，それらの成果を使い，現実社会の問題に適用するための課題に対する研究が主流である [77]，[42].

設計と実装の隔りという意味では，開発方法論に関する研究は非常に関連がある．エージェント，MAS の開発方法論としては，MaSE[29]，Gaia[88]，Tropos[39] などが提案されている．SODA[70]，TuCSoN[71]，エージェント間コミュニケーション基盤としては，KAoS[17] や，エージェント間のコミュニケーションのオブジェクト指向における考察 [7] も興味深い研究である．本研究も，インタラクションという概念を中心に捉えた，一つの開発方法論を目指すものでもある．

6.1.2　アスペクト指向

エージェント，MAS という観点以外では，アスペクト指向プログラミング [50] も設計と実装の隔りに取り組む研究である．IOM/T は，インタラクションというエージェントにまたがる横断的関心事を，自律性としてのエージェントと，協調性としてのインラクションとを区別し，関心事の分離 (Separation of Concerns) を達成するものである．近年では，イベント駆動に注目した研究が多く，モジュール分割をした際の等価性をイベント計算を用いて保証するもの [12] や，動的にイベントハンドラが設定される際の並列処理における安全性を型の概念により確保するもの [58] などが提案されている．

6.2　検証・テスティングに関する関連研究

オブジェクト指向言語を中心としたソフトウェア開発においては，様々な手法が体系的に考案され，その手法も確立されている [16, 84]．MAS の検証・テスティングにおいて，これらの技術をそのまま適用することは難しい．エージェントとオブジェクトには，エージェントには非決定的な動作が期待されるなどモデルにおける概念的相違と，エージェントはメソッド呼び出しにより協調動作を行なうわけではないため，オブジェクト指向言語のアプローチをそのまま適用することが難しいという実装上の相違があるためである．

しかしながら，MAS においてもこれらの検証・テスティングは非常に大きな関心事の一つである．例えば，モバイルエージェントが正しいタスクを実行しているかを，オートマトンを利用して，パラメタライズド検証を行う手法 [80]，テストやデバッグの支援が提供されたエージェントプラットフォームである INGENIAS[41] の提案などを始めとして，多くの研究がある．エージェントは，BDI モデルに代表されるように手続きよりもルー

ルで表現されることが主流であるため，テストや検証も形式手法を利用した研究が多い [92, 93, 34, 54, 68, 23, 66, 67].

本研究の契約による設計や，単体テストの導入は，手続き的に表現したインタラクションに対して，想定したことが想定した通りに実行されることを保証する手法である．これは，IOM/T を利用して実装することを前提として，オブジェクト指向において確立されたテスト技術を MAS においても適用可能な形として拡張したものである．

6.3 Web サービスに関する関連研究

個別に提供されるサービスを組み合わせて，より大きなことを実施する仕組みは，Web サービスにおいても期待されていることであり，Web サービスの連携について記述する仕様が，WS-BPEL[2]，WS-CDL[48] として規定されている．そして，Web サービスの連携においても，その協調に対する検証方法についての研究 [90, 76] や，ビジネスモデルからこれらの記述の導出方法についての研究 [32] が実施されている．

WS-BPEL や WS-CDL と，IOM/T は表現する内容は類似している．しかしながら，XML 形式による定義の記述であるため，粒度の細かな情報については表現できない，あるいは，表現できたとしても冗長な表現となってしまう．一方，IOM/T は実装言語であるため，変数の扱いなど，直感的に理解可能な形で表現することが可能である．

6.4 アジャイル開発の設計に関する関連研究

アジャイル開発は，アジャイル開発宣言 [11] を以降広まってきているソフトウェア開発の考え方である．より具体的な手法としては，開発フレームワークとしての Scrum[81]，エンジニアリングのプラクティス集として eXtream Programming(XP)[10]，リーンソフトウェア開発 [73]，Crystal[22] などが代表例である．残念ながら，アジャイル開発自体が学術的に注目を集めるようになったのは最近であり，アジャイル開発に関する研究は多くない．アジャイル開発の適切なイテレーション期間の推定手法 [82] やケーススタディとして教育に適用した事例 [61]，[59] が報告されているにとどまり，これまでのソフトウェア工学に関わる研究と比較すると多くない．本研究は，アジャイル開発の反復的開発を想定し，ユーザストーリごとに，インラクションを追加，あるいは，拡張していく．IDD によりユーザストーリとインラクション，そのインタラクションに対して，シーケンス図による設計と IOM/T による実装，それぞれが明確に対応し，また，インタラクション単独での単体テストや，契約による設計による検証など，テストハーネスを用意している．ま

た，IDD は，MAS において，シーケンス図によりドメインモデルを構築しながら設計する手法であり，ドメイン駆動開発 (DDD)[35] との関連も深い．

ところで，設計はソフトウェア工学における主要な関心事の一つであり，これまでに，多くの設計手法が提案されている．Jackson 法 [55] や構造化分析 [60] をはじめ，これまでのソフトウェア工学の成果として，その前提や用途に応じて，様々な設計法 [40] などである．特に，ICONIX[1] と IDD の考え方は近い．また，ICONIX をアジャイル開発に適用した事例 [24] も報告されている．既に多くの設計に関する研究が存在しているが，ほとんどの場合，ウォーターフォールモデル型の開発を想定しており，アジャイル開発においては，そのまま適用することが難しいものが多い．

第7章　まとめと今後の研究

7.1　まとめ

オブジェクト指向言語による実装を想定した従来のMASの開発には課題があり，特に，エージェント間に横断して関連するインタラクションプロトコルの実装には，インタラクションが複数のエージェントに分散されて実装されるため，設計と実装の隔りが大きく，結果として可読性が低く，DRY原則を守りにくいという課題があることを示した(第2章).

その上で，我々は．インタラクション記述言語IOM/Tを提案した．IOM/Tはシーケンス図との等価性を検証することができる言語仕様となっており，また，インタラクションを一つのソフトウェアモジュールとして扱うことができる(第3章).

更に，IOM/Tを中心とした拡張として，契約による設計，インタラクションに対する単体テスト，アジャイル開発への適用を考慮したインタラクション駆動設計を提案した(第4章).

そして，これらの提案がMAS開発の課題を軽減し，一定の期待した効果があることを示した(第5章).

7.2　今後の研究

本研究の発展として，以下のことを検討している．

7.2.1　IOM/Tの実証実験

実際のIoT環境を想定したアプリケーションの開発において，IOM/Tを利用した開発による効率化が期待できるのかを，実際にアプリケーション開発を実施することにより検証したい．

7.2.2　メトリクス

本研究において注目している課題は定性的に定義をしている．これらの課題に対して，定量的に評価するためのメトリクスの開発を検討している．

7.2.3　上流工程への更なる拡張

本研究では，IDD を提案し，限定的ではあるが実証実験によりその効果を検証した．今後は，より精度の高い実証実験を計画するとともに，より上流，つまり，要求の時点からインタラクションを検討することで，要求獲得を支援する手法を構築していく．

付 録A　BNF - IOM/Tの言語仕様

Interaction ::= *interaction Name Roles Protocol*

Roles ::= *Role* | *Role Roles*

Role ::= *role Name Variables Functionalities*

Variables ::= *Variable* | *Variable Variables*

Variable ::= *Type Name*;

Functionalities ::= *Functionality* | *Functionality Functionalities*

Functionality ::= *Type Name* (*Args*);

Args ::= *Arg* | *Arg*, *Args*

Arg ::= *Type Name*

Protocol ::= *protocol Activities*

Activities ::= *Activity* | *Actitivy Activities*

Activity ::= *Control* | *Play*

Control ::= *Conditional* | *Loop* | *Parallel*

Conditional ::= *If* | *If Else* | *If ElseIfs Else*

If ::= *if* (*Condition*) *Activities*

ElseIfs ::= *ElseIf* | *ElseIf ElseIfs*

ElseIf ::= *else if* (*Condition*) *Activities*

Else ::= *else Activities*

Loop ::= *while* (*Condition*) *Activities*

Parallel ::= *parallel ParallelActivities*

ParallelActivities ::= *ParallelActivity* | *ParallelActivity ParallelActivities*

ParallelActivity ::= *Activities*

Play ::= *play Role JavaStatements*

Condition ::= *Name.Name*()

Type ::= *literal*

Name ::= *literal*

参考文献

[1] : *Use Case Driven Object Modeling with UML: Theory and Practice*, chapter Introduction to ICONIX Process, pp. 1–20, Apress (2007).

[2] : Web Services Business Process Execution Language Version 2.0, Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC (2007).

[3] Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., and Torroni, P.: Compliance Verification of Agent Interaction: a Logic-Based Tool, *From Agent Theory to Agent Implementation(AT2AI-4)* (2004).

[4] Alliance, O.: OpenADR 2.0 Specifications, http://www.openadr.org/specification (2013).

[5] Amal El Fallah-Seghrouchni, A. S.: CLAIM : A Computational Language for Autonomous, Intelligent and Mobile Agents, *Programming Multiagent Systems, LNAI*, Vol. 3067, Springer (2004).

[6] Ashton, K.: That 'Internet of Things' Thing, RFID Journal (2009).

[7] Baldoni, M., Boella, G. and van der Torre, L.: Importing Agent-like Interaction in Object Orientation, *Proceedings of the 7th WOA 2006 Workshop From Objects to Agents* (2006).

[8] Barbuceanu, M. and Fox, M. S.: Cool: A language for describing coordination in multiagent systems, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)* (Lesser, V. and Gasser, L., eds.), San Francisco, CA, USA, AAAI Press, pp. 17–24 (1995).

[9] Bass, L., Weber, I. and Zhu, L.: *DevOps: A Software Architect's Perspective*, Addison-Wesley Professional, 1st edition (2015).

[10] Beck, K.: Extreme Programming, *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '99, Washington, DC, USA, IEEE Computer Society, pp. 411– (1999).

[11] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D.: Manifesto for Agile Software Development (2001).

[12] Bejleri, A., Mezini, M. and Eugster, P.: Cooperative Decoupled Processes: The e-Calculus and Linearity, *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, New York, NY, USA, ACM, pp. 82–93 (2016).

[13] Bellifemine, F., Poggi, A. and Rimassa, G.: JADE - A FIPA-compliant agent framework. http://jade.cselt.it/papers/PAAM.pdf.

[14] Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., III, W. N. M. and Diao, Y.: ABLE: A toolkit for building multiagent autonomic systems, *IBM Systems Journal*, Vol. 41, No. 3, pp. 350–371 (online), 10.1147/sj.413.0350, (2002).

[15] Boix, E. G., Scholliers, C., Larrea, N. and Meuter, W. D.: Connect.js: A cross mobile platform actor library for multi-networked mobile applications, *AGERE! 2015 Programming based on Actors, Agents, and Decentralized Control* (2015).

[16] Bourque, P. and Fairley, R. E.: *Guide to the Software Engineering Body of Knowledge - SWEBOK v3.0*, IEEE CS, 2014 version edition (2014).

[17] Bradshaw, J. M.: Kaos: An open agent architecture supporting reuse, interoperability, and extensibility, *Tenth Knowledge Acquisition for Knowledge Based Systems* (1996).

[18] Braubach, L., Pokahr, A. and Moldt, D.: Goal Representation for BDI Agent Systems, *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop* (2004).

[19] Cannon, H.: Flavors: A non-hierarchical approach to object-oriented programming, Symbolics Inc. (1982).

[20] Cardelli, L.: A Semantics of Multiple Inheritance., *Proc. Of the International Symposium on Semantics of Data Types*, New York, NY, USA, Springer-Verlag New York, Inc., pp. 51–67 (1984).

[21] Cass, S.: The 2015 Top Ten Programming Languages, IEEE Spectrum, http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages (2015).

[22] Cockburn, A.: *Crystal Clear a Human-powered Methodology for Small Teams*, Addison-Wesley Professional, first edition (2004).

[23] Coelho, R., Kulesza, U., von Staa, A. and Lucena, C.: Unit Testing in Multi-agent Systems Using Mock Agents and Aspects, *Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems*, SELMAS '06, New York, NY, USA, ACM, pp. 83–90 (2006).

[24] Collins-Cope, M., Rosenberg, D. and Stephens, M.: *Agile Development with the ICONIX Process: People, Process and Pragmatism*, Springer, Dordrecht (2005).

[25] CONSORTIUM, E.: ECHONET Specifications Version 3.21, https://echonet.jp/spec_v321/ (2005).

[26] da Silva, V. T., Choren, R. and de Lucena, C. J.: A UML Based Approach for Modeling and Implementing Multi-Agent Systems, *The Third International Joint Conference on Autonomous Agents & Multi Agent Systems AAMAS2004*, pp. 914–921 (2004).

[27] Dastani, M., van Riemsdijk, B., Dignum, F. and Meyer, J.: A programming language for cognitive agents: Goal-directed 3APL, *PROMAS*, pp. 111–130 (2003).

[28] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*, Vol. 51, No. 1, pp. 107–113 (2008).

[29] DeLoach, S. A.: *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, chapter The MaSE Methodology, pp. 107–125, Springer US (2004).

[30] Dinkloh, M. and Nimis, J.: A tool for integrated design and implementation of conversations in multi-Agent systems, *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2003 Workshop* (2003).

[31] DOI, T., YOSHIOKA, N., TAHARA, Y. and HONIDEN, S.: IOM/T: An Interaction Description Language for Multi-Agent Systems, *Fourth International Joint Conference on Autonomous Agents & Multi Agent Systems AAMAS2005* (2005).

[32] Ebrahimifard, A., Amiri, M. J., Arani, M. K. and Parsa, S.: Mapping BPMN 2.0 Choreography to WS-CDL: A Systematic Method, (online), available from ⟨https://www.researchgate.net/publication/299388141⟩ (2016).

[33] Ehrler, L. and Cranefield, S.: Executing Agent UML diagrams, *The Third International Joint Conference on Autonomous Agents & Multi Agent Systems AAMAS2004*, pp. 906–913 (2004).

[34] Ekinci, E. E., Tiryaki, A. M., Çetin, O. and Dikenelli, O.: Agent-Oriented Software Engineering IX, Springer-Verlag, Berlin, Heidelberg, chapter Goal-Oriented Agent Testing Revisited, pp. 173–186 (2009).

[35] Evans: *Domain-Driven Design: Tacking Complexity In the Heart of Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003).

[36] Fertig, R. T.: *The Software revolution : trends, players, market dynamics in personal computer software*, North-Holland, New York, Amsterdam, Oxford (1985).

[37] Forum, M. P.: MPI: A Message-Passing Interface Standard, Technical report, Knoxville, TN, USA (1994).

[38] Georgeff, M. P., Pell, B., Pollack, M. E., Tambe, M. and Wooldridge, M.: The Belief-Desire-Intention Model of Agency, *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, ATAL '98, London, UK, UK, Springer-Verlag, pp. 1–10 (1999).

[39] Giunchiglia, F., Mylopoulos, J. and Perini, A.: The Tropos Software Development Methodology: Processes, Models and Diagrams, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1*, AAMAS '02, New York, NY, USA, ACM, pp. 35–36 (online), 10.1145/544741.544748, (2002).

[40] Gomaa, H.: *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition (1993).

[41] Gómez-Sanz, J. J., Botía, J., Serrano, E. and Pavón, J.: *Agent-Oriented Software Engineering IX: 9th International Workshop, AOSE 2008 Estoril, Portugal, May 12-13, 2008 Revised Selected Papers*, chapter Testing and Debugging of MAS Interactions with INGENIAS, pp. 199–212, Springer Berlin Heidelberg (2009).

[42] Harel, D. and Katz, G.: Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures, *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, New York, NY, USA, ACM, pp. 95–108 (2014).

[43] Huget, M.-P.: Generating Code for Agent UML Sequence Diagrams, Technical report, University of Liverpool Department of Computer Science (2002).

[44] Hunt, A. and Thomas, D.: *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999).

[45] Ishida, T.: Q: A Scenario Description Language for Interactive Agents, *IEEE Computer* (2002).

[46] Ishikawa, F., Doi, T., Sakamoto, K., Yoshioka, N. and Tanabe, Y.: Enlightening Test-Driven with Formal, Formal with Test-Driven through Spec-Test-Go-Round, Technical report, CENTER FOR GLOBAL RESEARCH IN ADVANCED SOFTWARE SCIENCE AND ENGINEERING, NATIONAL INSTITUTE OF INFORMATICS, 2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN (2015).

[47] Jacobson, I., Ng, P.-W., McMahon, P. E., Spence, I. and Lidman, S.: *The Essence of Software Engineering: Applying the SEMAT Kernel*, Addison-Wesley Professional, 1st edition (2013).

[48] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y. and Barreto, C.: Web Services Choreography Description Language Version 1.0, World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109 (2005).

[49] Kawamura, T., Hasegawa, T., Osuga, A. and Honiden, S.: Bee-gent: Bonding and Encapsulation Enhancement Agent Framework for Development of Distributed Systems, *Proceedings of the 6th Asia-Pacific Software Engineering Conference (APSEC 99), IEEE*, pp. 260–267 (1999).

[50] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings European Conference on Object-Oriented Programming* (Akşit, M. and Matsuoka, S., eds.), Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242 (1997).

[51] 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).

[52] Kuwabara, K., Ishida, T. and Osato, N.: AgenTalk: Describing Multiagent Coordination Protocols with Inheritance, *Proc. 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '95)* (1995).

[53] 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).

[54] Lam, D. N. and Barber, K. S.: Debugging Agent Behavior in an Implemented Agent System, *Proceedings of the Second International Conference on Programming Multi-Agent Systems*, ProMAS'04, Berlin, Heidelberg, Springer-Verlag, pp. 104–125 (2005).

[55] Learmonth & Burchett Management Systems Plc, C.: *LBMS Jackson System Development (Version 2.0): Method Manual*, John Wiley & Sons, Inc., New York, NY, USA (1992).

[56] Leavens, G. and Cheon, Y.: Design by Contract with JML (2003).

[57] LIN, H.: Computing Bisimulations for Finite-Control pi-Calculus, *Journal of Computer Science and Technology*, Vol. 15, No. 1 (2000).

[58] Long, Y. and Rajan, H.: A Type-and-effect System for Asynchronous, Typed Events, *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, New York, NY, USA, ACM, pp. 42–53 (2016).

[59] Lu, B. and DeClue, T.: Teaching Agile Methodology in a Software Engineering Capstone Course, *J. Comput. Sci. Coll.*, Vol. 26, No. 5, pp. 293–299 (2011).

[60] Marca, D. A. and McGowan, C. L.: *SADT: Structured Analysis and Design Technique*, McGraw-Hill, Inc., New York, NY, USA (1987).

[61] Marchesi, M., Ascone, A., Calavaro, G., Olsson, H. H., Armani, F., Garbajosa, J. and Cantone, G.: How to Teach Agile at University, how to Coach Agile in Industry and, in general, how to keep software engineering teaching linked to software evolution, Panel in 15th International Conference on Agile Software Development, XP2014 (2014).

[62] McCabe, F. G. and Clark, K. L.: April – agent process interaction language, *Intelligent Agents: Theories, Architectures, and Languages (LNAI volume 890)* (Wooldridge, M. and Jennings, N. R., eds.), Springer-Verlag: Heidelberg, Germany, pp. 324–340 (1995).

[63] Meyer, B.: Applying "Design by Contract", *Computer*, Vol. 25, No. 10, pp. 40–51 (1992).

[64] Meyer, B.: *Eiffel: the Language*, Prentice-Hall (1992).

[65] Milner, R.: *communicating and mobile systems: the π − calculus*, Cambridge University Press (1999).

[66] Nguyen, C. D., Miles, S., Perini, A., Tonella, P., Harman, M. and Luck, M.: Evolutionary Testing of Autonomous Software Agents, *Autonomous Agents and Multi-Agent Systems*, Vol. 25, No. 2, pp. 260–283 (2012).

[67] Nguyen, C. D., Perini, A. and Tonella, P.: Ontology-based Test Generation for Multiagent Systems, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '08, Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems, pp. 1315–1320 (2008).

[68] Núñez, M., Rodríguez, I. and Rubio, F.: Specification and Testing of Autonomous Agents in e-Commerce Systems: Research Articles, *Softw. Test. Verif. Reliab.*, Vol. 15, No. 4, pp. 211–233 (2005).

[69] Object Management Group: OMG Unified Modeling Language^{TM} (OMG UML), Superstructure (2011). http://www.omg.org/spec/UML/2.4.1/Superstructure/.

[70] Omicini, A.: SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems, *AOSE*, pp. 185–193 (2000).

[71] Omicini, A. and Zambonelli, F.: Coordination for Internet Application Development, *Autonomous Agents and Multi-Agent Systems*, Vol. 2, No. 3, pp. 251–269 (1999).

[72] Pokahr, A., Braubach, L. and Lamersdorf, W.: Jadex: A BDI Reasoning Engine, *Multi-Agent Programming* (R. Bordini, M. Dastani, J. D. and Seghrouchni, A. E. F., eds.), Springer Science+Business Media Inc., USA, pp. 149–174 (2005). Book chapter.

[73] Poppendieck, M. and Poppendieck, T.: *Lean Software Development: An Agile Toolkit*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003).

[74] Rao, A. S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World* (van Hoe, R., ed.), Eindhoven, The Netherlands (1996).

[75] Rao, A. S. and Georgeff, M. P.: BDI Agents: From Theory to Practice, *ICMAS* (1995).

[76] Rebai, S., Kacem, H. H., KarÃća, M., Pomares, S. E. and Kacem, A. H.: CDLVT: A Formal Verification Tool of Non-functional Properties for WS-CDL Specification, *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 191–196 (online), 10.1109/WETICE.2015.14, (2015).

[77] Ricci, A.: From Actor Event-Loop to Agent Control-Loop: Impact on Programming, *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, New York, NY, USA, ACM, pp. 121–132 (2014).

[78] Ricci, A., Agha, G. and Bordini, R. H.: Agere! (Actors and Agents Reloaded): Splash 2011 Workshop on Programming Systems, Languages and Applications Based on Actors, Agents and

Decentralized Control, *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, New York, NY, USA, ACM, pp. 143–146 (2011).

[79] Ross, R., Collier, R. and O'Hare, G.: AF-APL Bridging Principles & Practice in Agent Oriented Languages, *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop* (2004).

[80] Rubin, S.: Parameterised Verification of Autonomous Mobile-Agents in Static but Unknown Environments, *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems, pp. 199–208 (2015).

[81] Schwaber, K. and Beedle, M.: *Agile Software Development with Scrum*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition (2001).

[82] Shiohama, R., Washizaki, H., Kuboaki, S., Sakamoto, K. and Fukazawa, Y.: Estimate of the Appropriate Iteration Length in Agile Development by Conducting Simulation, *Agile Conference (AGILE), 2012*, pp. 41–50 (online), 10.1109/Agile.2012.16, (2012).

[83] Shore, J. and Warden, S.: *The Art of Agile Development*, O'Reilly, first edition (2007).

[84] Sommerville, I.: *Software Engineering*, Addison-Wesley Publishing Company, USA, 9th edition (2010).

[85] Stroustrup, B.: Multiple Inheritance for C++ (1999).

[86] Tahara, Y., Ohsuga, A. and Honiden, S.: Mobile Agent Security with the IPEditor Development Tool and the Mobile UNITY Language, *Proc. of Agents 2001*, ACM Press, pp. 656–662 (2001).

[87] White, T.: *Hadoop: The Definitive Guide*, O'Reilly Media, Inc., 1st edition (2009).

[88] Wooldridge, M., Jennings, N. R. and Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design, *Autonomous Agents and Multi-Agent Systems*, Vol. 3, No. 3, pp. 285–312 (2000).

[89] Woolridge, M. and Wooldridge, M. J.: *Introduction to Multiagent Systems*, John Wiley & Sons, Inc., New York, NY, USA (2001).

[90] Yu, M., Wang, Z. and Niu, X.: Verifying Service Choreography Model Based on Description Logic, *Mathematical Problems in Engineering Volume 2016*, Hindawi Publishing Corporation (2016).

[91] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA, USENIX Association, pp. 10–10 (2010).

[92] Zhang, Z., Thangarajah, J. and Padgham, L.: Automated Unit Testing Intelligent Agents in PDT, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems: Demo Papers*, AAMAS '08, Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems, pp. 1673–1674 (2008).

[93] Zhang, Z., Thangarajah, J. and Padgham, L.: Model Based Testing for Agent Systems, *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, Richland, SC, International Foundation for Autonomous Agents and Multiagent Systems, pp. 1333–1334 (2009).

関連論文の印刷公表の方法及び時期

[1] Takuo Doi, Yasuyuki Tahara, Shinichi Honiden
論文題目「IOM/T: an interaction description language for multi-agent systems」
2005 年 7 月 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)
(第 3 章の内容)

[2] 土肥 拓生, 吉岡 信和, 田原 康之, 本位田 真一
論文題目「インタラクション記述言語 IOM/T」
2005 年 9 月 電子情報通信学会論文誌, Vol.J88-D-I, No.9, pp.1299-1311
(第 3 章の内容)

[3] 土肥 拓生, 本位田 真一
論文題目「契約による設計を用いたインタラクションの実装」
2006 年 5 月　情報処理学会論文誌，Vol.47, No.5，pp.1371-1381
(第 4 章の内容)

[4] Takuo Doi, Shinichi Honiden
論文題目「IOM/T: interaction-oriented model by textual notation」
2007 年 International Journal of Agent-Oriented Software Engineering, IJAOSE Volume 1(3/4) pp.266-294
(第 3 章, 第 4 章の内容)

参考論文の印刷公表の方法及び時期

[1] Takuo Doi, Nobukazu Yoshioka, Yasuyuki Tahara, Shinichi Honiden
論文題目「Bridging the Gap between AUML and Implementation using IOM/T」
2004 年 7 月 The Second International Workshop on Programming Multiagent Systems Languages and tools (PROMAS 2004)

[2] 土肥 拓生，吉岡 信和，田原 康之，本位田 真一
論文題目「契約による設計を用いたインタラクションの実装」
2005 年 11 月 合同エージェントワークショップ＆シンポジウム 2005 (JAWS2005)

[3] 土肥拓生，石川冬樹
論文題目「インタラクションに注目したアジャイル開発における設計手法」
2015 年 11 月 第 22 回ソフトウェア工学の基礎ワークショップ (FOSE 2015)

著者略歴

平成 15 年 3 月	東京大学 理学部 情報科学科卒業
平成 15 年 4 月	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 入学
平成 17 年 3 月	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 修了
平成 17 年 4 月	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 博士課程 入学
平成 20 年 4 月	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 博士課程 単位取得の上退学
平成 25 年 10 月	電気通信大学 情報システム学研究科 社会知能情報学専攻 博士後期課程 入学
平成 28 年 9 月	電気通信大学 情報システム学研究科 社会知能情報学専攻 博士後期課程 修了見込