特別研究報告書

大規模マルチエージェントシステムの
シナリオ実行制御

指導教員　　石田 亨 教授

京都大学工学部情報学科

山根 昇平

平成17年2月10日

大規模マルチエージェントシステムのシナリオ実行制御

山根 昇平

内容梗概

大規模災害における避難誘導システムや，広域にわたる交通流制御を行うナビゲーションシステムを評価するために，実際に実験を行うのは非常に困難である．そこで，人間をエージェントに置き換えて大規模マルチエージェントシステムを用いたシミュレーションを行うことにより，評価を容易にすることができる．

大規模シミュレーションをマルチエージェントシステム上で行う際には，以下のような機能が必要である．

1. シミュレーション及びエージェントの実行管理
   大規模シミュレーションでは，エージェント数が多くなるだけでなく，一度のシミュレーションにおける，シミュレーション内の時間も長くなると考えられる．そこで，シミュレーションをどう進行させるかを記述する必要がある．また，それに伴った，エージェントの動作変化などの管理や，シミュレーションを適切に進行させるためのエージェントのデバッグなど，シミュレーションの実行に関わる管理機能が必要となる．
2. 時間管理
   シミュレーションを行うとき，必要なときだけ速度を遅くし，それ以外のときはできるだけ速く進行させることで効率を上げるといったことが行われる．また，シミュレーションが重要な局面に達したとき，時間を止めるなどして詳細に観測を行うといった状況が考えられる．
   そこで，このようなシミュレーション速度の切り替えを自動的に判断して行う，時間管理機能が必要である．

以上のような問題を解決するために，マルチエージェントシステムのエージェント記述に対してメタプログラミングを行うことで，エージェントの動作を制御する．また同時に，シミュレーションの実行状況の観測を行い，シミュレーションの制御を行うことを考えた．

本研究ではまず，エージェント記述言語に対するメタプログラミングとシミュレーションの制御管理を統合して行うアーキテクチャを考案した．そして，エージェント記述言語としてシナリオ記述言語Qを用いてこのアーキテクチャを実

装する，メタシナリオを開発した．開発に当たり，シナリオ実行制御に関するもの，エージェントシステムに対する環境設定・取得を行うものなど，メタシナリオの機能を明確に定義した．ここでQ言語を用いたのは，Q言語は母言語としてSchemeを用いており，Schemeがメタプログラミングの実装に適していたからである．

メタシナリオによって，以下のような機能が実現できる．

1. メタプログラミングによるシナリオ実行制御方式
   エージェント記述言語に対してメタプログラミングを行うことによって，エージェントの行動の観測・制御を行うことができる．また，エージェントシステムに対する環境の設定・取得によって，シミュレーション全体の観測・制御が可能となる．これらを統合して行うアーキテクチャによって，シミュレーション全体の実行状況やエージェントのシナリオ実行を観測しそれに基づいたシナリオ実行制御方式が実現できる．これにより，エージェントの動作変更の管理や，動作修正，デバッグなどを行うことができる．
2. シナリオ実行制御方式を用いたシミュレーションの時間管理
   メタシナリオではエージェントシステムに対する観測機能として，シナリオ実行制御によるエージェントの行動の観測及び，シミュレータの状況の観測を行うことができる．そしてこれらの観測に基づいてエージェントシステムに対する環境設定を行うことが可能となる．これによって，シミュレーション速度の切り替えが必要となる場面を自動的に判断して切り替え行うことで，時間管理機能が実現できる．

最後に，特に時間管理機能に関して，実際にシナリオ・メタシナリオを記述して動作させた．これにより，メタシナリオによって時間管理機能が実現可能であることを示した．

Controlling Scenario Execution in Massively Multiagent Systems

Shohei YAMANE

Abstract

To evaluate large-scale navigation systems, such as evacuation navigation systems for disaster and car navigation systems for wide-area traffic control, it is very hard to perform experiments in real space. Performing simulations that use a large-scale multi agent system and replace users by agents makes this evaluation easier.

To perform large-scale simulations on a multi agent system, following functions are required.

1. Execution control of simulations and agents

   In a large-scale simulation, period of a simulation can be long, as well as the number of agents is large. So the description about how to perform a simulation is required. For this description, control functions of simulation are required. Control functions include management of changes of the behavior of agents and debug for agents to perform simulation properly.

2. Time management

   In performing a simulation, it can be performed efficiently if it is run as fast as possible except when it need to be run slowly. In addition, if a simulation can be stopped or slower when the simulation is in an important phase, we can observe it carefully.

   For these purposes, time management function that switches simulation speed is required.

To solve these problems, this paper proposed to apply metaprogramming to an agent description language to control behavior of agents. And at the same time, a simulation is controlled by observing execution environment of a simulation.

At first, this paper proposed the architecture to integrate metaprogramming for an agent description language and simulation control. Next, we developed meta scenario that implements this architecture using *scenario description lan-*

guage $\mathcal{Q}$ as an agent description language. On developing meta scenario, we defined functions of meta scenario such as scenario execution control and setting and acquisition of information about environment of an agent system. The reason why we use $\mathcal{Q}$ language as an agent description language is that the mother language of $\mathcal{Q}$ language is scheme and scheme is suitable to implement metaprogramming. meta scenario can realize following functions.

1. Scenario execution control method by metaprogramming

   By metaprogramming for an agent description language, the behavior of agents can be observed and controlled. In addition, by setting and acquisition of information about environment of an agent system, whole of a simulation can be also observed and controlled. The architecture that integrates these observations and controls realized scenario execution control method based on observations of whole of a simulation and scenario execution of agents. this made it possible to change, modify and debug the behavior of agents.

2. Time management by scenario execution control method

   As observation functions to an agent system, meta scenario can observe the behavior of agents by scenario execution control method and can observe condition of simulator. And meta scenario made it possible to set the environment of an agent system based on these observation. This realized time management that decide when the simulation speed should be switched and switch it automatically.

Finally, about time management, we described scenario and meta scenario and performed the simple simulation. This showed that meta scenario can realize time management.

大規模マルチエージェントシステムのシナリオ実行制御

目次

第1章　はじめに

近年，携帯電話などの普及により，ユビキタス環境が整いつつある．これらのツールを用いて大規模なマルチエージェントナビゲーションシステムを構築することで，様々な利益が得られると考えられる．例えば，地震などの大規模災害時において，現在地や健康状態などの個人情報を元にエージェントが的確な非難経路を選択し誘導を行うシステムを構築できれば，被害を最小限に抑えることができる．また，広範囲に渡る交通流を状況に応じて車一台単位でナビゲーションすることにより，渋滞の緩和に貢献できると考えられる．このようなシステムを評価するにあたっては，実際に人や車を集めて実験を行うのが非常に困難である．そこで社会的インタラクションを行うエージェントを含むシステムの設計手法として，社会中心設計 (Sociaty-Centered Design) が提案されている [1].

社会中心設計では，社会的なシステム開発の過程としてシステムの評価を行うために．マルチエージェントシステムを用いたシミュレーションや参加型シミュレーションを繰り返し行う．これによって，ユーザをシミュレートするエージェントの動作モデルや，ナビゲーションなどのサービスを行うエージェントの動作の，改良を行う．ここで言う参加型シミュレーションとは，ユーザをシミュレートするエージェントの一部を実際に現実世界にいる人間が操作することで，ユーザがシステムに対してどのように振舞うかを観察して，エージェントの動作モデルを改良するものである．

しかし，大規模マルチエージェントシステムを用いてシミュレーションを行うには，いくつかの問題点がある．そこで，以下のような機能を実現することを考えた．

1. シミュレーション及び，エージェントの実行管理
   大規模マルチエージェントシミュレーションでは，エージェント数が膨大になったり，シミュレーションの舞台となる空間が広域になったりするだけでなく，仮想世界の中での時間も長くなると考えられる．
   そこで，シミュレーション全体の状況やエージェントの動作を監視して，それに応じたシミュレーションやエージェントの実行の制御・管理を行うシナリオを記述する言語が必要である．これにより，エージェントの入退場や動作シナリオの切り替えをはじめとするエージェントの動作管理や，期

待されない動作によってシミュレーションの進行の妨げになる動作を行ったエージェントに対する動作の修正やデバッグを，シミュレーション全体の状況に応じて行う機能が，実現できる．

2. 時間管理

シミュレーションを行うとき，必要なときだけ速度を遅くし，それ以外のときはできるだけ速く進行させることで効率を上げるといったことが行われる．例えば，参加型シミュレーションでは一部のエージェントは現実世界にいるユーザによって操作される．このため，そのエージェントに対して何らかの意思決定を求める必要が発生した場合，シミュレーション速度が速い状態のままでは，適切な意思決定を行う時間が確保できない．一方で，シミュレーション速度を現実世界の時間と同じ速度でシミュレーションを行うと，シミュレーションの効率が著しく低下してしまう．そこで，人間の操作するエージェントが何らかの意思決定を行っている状態ではシミュレーションの速度を現実世界と同程度にし，意思決定が完了すれば速度を元の速さに戻す必要がある．また，シミュレーションが重要な局面に達したとき，時間を止めるなどして詳細に観測を行うといった状況が考えられる．そこで，このようなシミュレーション速度の切り替えを自動的に判断して行う，時間管理機能が必要である．

本研究では，以上で挙げたような機能の実現のため，マルチエージェントシステムのエージェント記述を読み取ることでエージェントの動作の観測や制御を行う，メタプログラミングを行うことを考えた．

また，シミュレーション全体の状況の観測を行ったり，エージェントシステムに対するアクションの実行したりする機能も必要である．そこでこれらの機能とメタプログラミングによるエージェントの動作制御を統合して記述することにより，シミュレーションやエージェントの実行制御を行う言語を設計し，実装を行った．

設計・実装に当たっては，エージェント記述言語として，京都大学で開発されたシナリオ記述言語Q [2] を採用した．これは，Q 言語は母言語として Scheme が用いられており，Scheme はメタプログラミングの実装に適しているからである．

さらに，非常に多数のエージェントを扱うことのできるエージェントサーバ

として，IBM で開発された Caribbean[3][4] がある．Q 言語は，トランスレータなどを介して，Q 言語で記述されたシナリオをこの Caribbean におけるイベント駆動型オブジェクトとして動作させることで，大規模マルチエージェントシステムとして，Caribbean との結合がなされている [5].

本稿では，Q 言語のシナリオ記述に対するメタプログラミングを行う言語，すなわちメタシナリオに関して詳細に説明する．まず第二章でメタシナリオのシナリオ実行制御アーキテクチャを説明し，第三章でメタシナリオの持つ機能について説明する．最後に，第四章では時間管理の例に関して実際にシミュレーションを行うメタシナリオを記述した.

第2章　シナリオ実行制御アーキテクチャ

本研究では，エージェントの動作制御の手法として，エージェント記述言語に対してメタプログラミングを行う方法を採用した．これは，エージェントの動作を規定するエージェント記述に対して直接アクセスすることで，柔軟なエージェントの動作制御を実現するためである.

本研究で開発したメタシナリオは，このようなメタプログラミングによるシナリオ実行制御と，エージェントシステムに対する制御管理機構を統合して行う言語である.

本章では，このようなメタシナリオを実現するアーキテクチャを詳細に説明する．さらにシナリオ実行制御の流れ，及び，このアーキテクチャによって実現可能となる例題をいくつか示す.

2.1　アーキテクチャ

図 1 に，本研究で用いたアーキテクチャを示す．これ以降本節では，この図に関して詳しく説明していく.

シナリオ記述は，エージェント記述言語で記述されたエージェント記述である．ここでは，エージェントに対するセンサ・アクションの起動をどのように実行していくかを記述する.

シナリオインタプリタはシナリオ記述を入力とし，それを解釈することで実行すべきエージェントのセンサ・アクションを決定し，実行する．シナリオインタプリタはエージェントシステム上の特定のエージェントと対応しており，こ

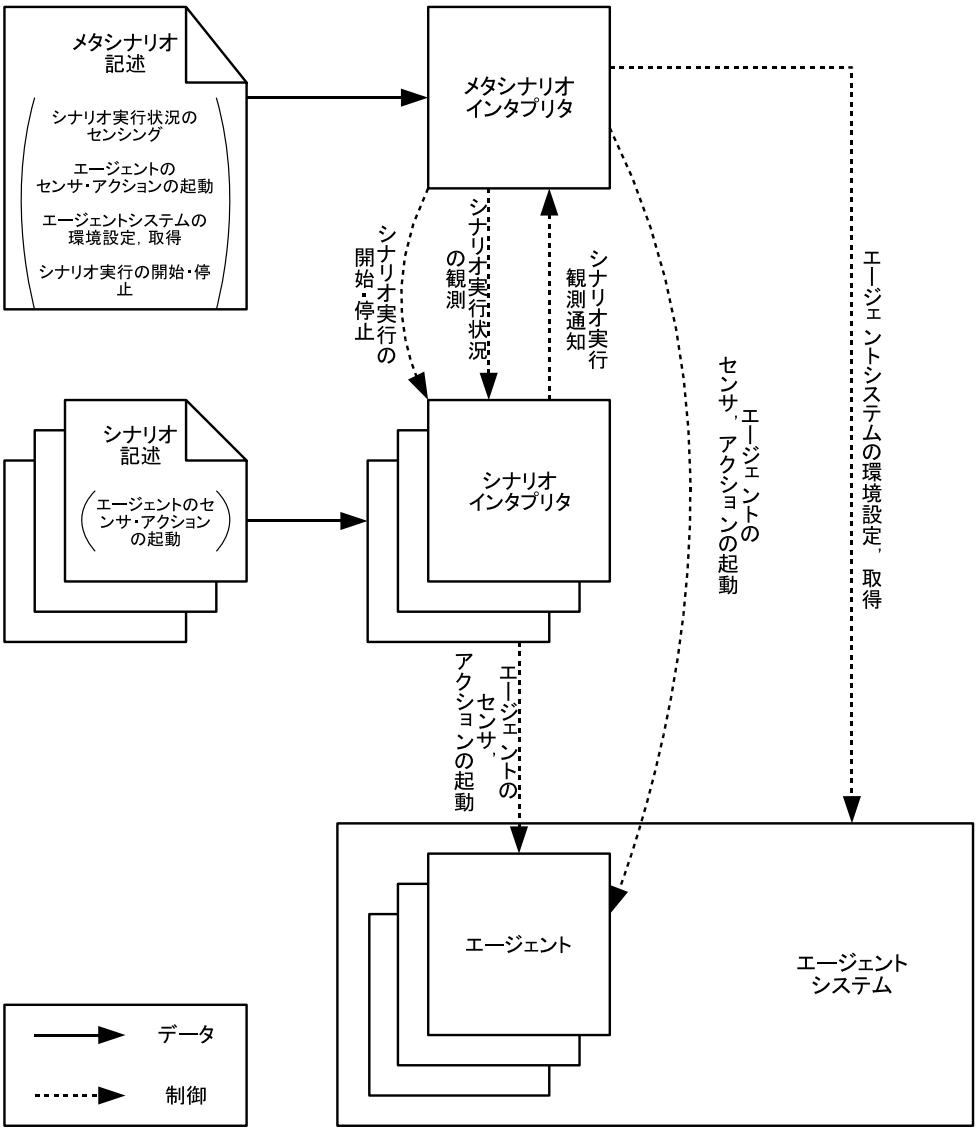

図 1: シナリオ実行制御アーキテクチャ

のシナリオインタプリタがシナリオ記述を解釈実行することで，エージェントの動作は進行する．

本研究では，このシナリオ記述に対してメタプログラミングを行うため，図 1 で示したようなアーキテクチャを考案した．このアーキテクチャでは，エージェントの実行を行うシナリオ記述とシナリオインタプリタに対して，それらの制御を行うメタシナリオ記述とメタシナリオインタプリタを導入している．

メタシナリオ記述は，シナリオ記述に対してどのようにシナリオ実行制御を行うか記述するものである．また同時に，シミュレーションの制御をどのように行うかも記述する．具体的には，シナリオ実行状況のセンシング，シナリオ実行の開始・停止，エージェントのセンサ・アクションの起動，エージェントシステムの環境設定・取得，をどのように実行するかを記述する．

メタシナリオインタプリタでは，このメタシナリオ記述を入力して解釈実行を行う．シナリオ実行状況のセンシングはシナリオインタプリタに対して，エージェントのセンサ・アクションの起動はエージェントシステム上のエージェントに対して，エージェントシステムの環境設定・取得はエージェントシステムに対して，それぞれ実行依頼を行う．

シナリオ実行状況のセンシングに関しては，依頼を受け取ったシナリオインタプリタがシナリオ実行観測通知を返すことで実行される．この観測は，シナリオインタプリタがシナリオ記述を解釈して実行すべきコマンドを決定したとき，それを実行する前にメタシナリオインタプリタからの観測依頼と比較することでなされる．

このようにしてメタシナリオインタプリタがメタシナリオ記述を解釈実行することによって，エージェント及びエージェントシステムの制御が行われる．

2.2 シナリオ実行の流れ

本節では，メタプログラミングによるシナリオ実行制御機構の導入によってどのようにシミュレーションの実行がなされるか，その順序を説明する．前節で説明したように，シナリオやシミュレーションの実行制御はメタシナリオインタプリタによってメタシナリオ記述が解釈実行なされることで行われる．そこで以下では，図 2 のシーケンス図を見ながら，メタシナリオの実行手順について説明する．

1. メタシナリオインタプリタは，メタシナリオ記述を読み込んで解釈するこ

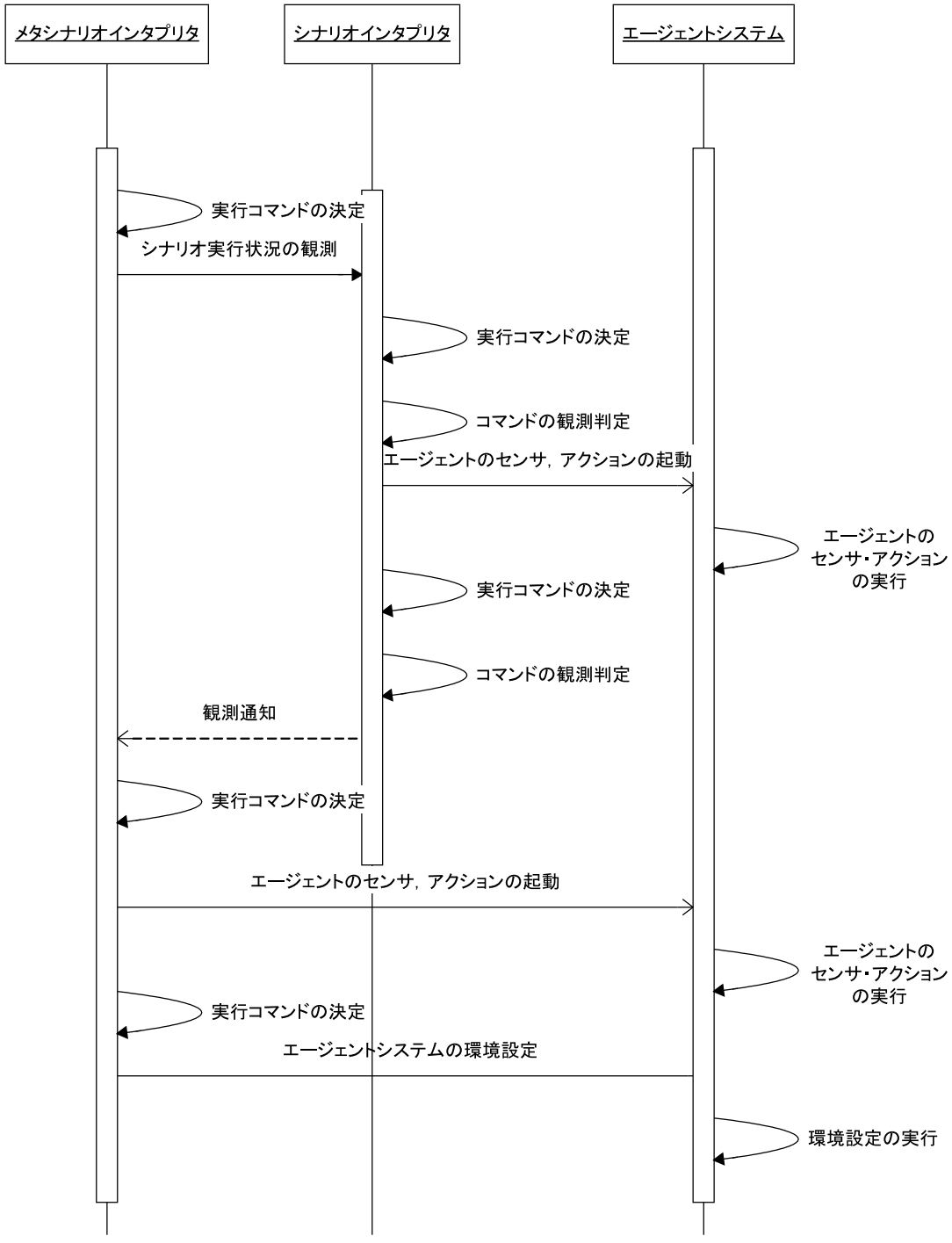

図 2: シナリオ実行のシーケンス図

とで，次に実行すべきコマンドを決定する．図 2 ではまず，エージェントの動作を観測するために，シナリオインタプリタに対してシナリオ実行状況の観測を行っている．

2. 一方でシナリオインタプリタは，エージェントの実行を行うため，シナリオ記述を読み込んで次に実行すべきコマンドを決定する．ここで決定されたコマンドは，実行される前にメタシナリオインタプリタが観測しているコマンドと一致するかどうかの判定がなされる．
3. 図 2 では，2. で選択されたコマンドはエージェントのセンサ・アクションの起動である．そしてこれは，メタシナリオインタプリタで観測しているコマンドではないと判定されている．このコマンドは，そのままエージェントシステムに対して実行される．
4. エージェントシステムはコマンドを受け取り，エージェントを動作させる．
5. コマンド実行後，シナリオインタプリタはさらにエージェントの動作を続けるため，シナリオ記述から次に実行すべきコマンドを決定する．そして 2. と同様，メタシナリオインタプリタが観測しているかどうかの判定がなされる．
6. 図 2 では，5. で決定されたコマンドはメタシナリオインタプリタで観測しているものと一致すると判定さたため，シナリオ実行の観測通知がメタシナリオインタプリタへと返される．
7. シナリオインタプリタからの観測通知を受け取ったメタシナリオインタプリタは，次のコマンドを実行するため，再びメタシナリオ記述を読み込んでメタシナリオ記述の実行を続ける．図 2 では，エージェントに対するセンサ・アクションの起動が選択され，エージェントシステムに対して実行される．
8. さらに同様にして，メタシナリオインタプリタはメタシナリオ記述を解釈実行することで，メタシナリオの実行を続ける．図 2 では，最後にエージェントシステムに対する環境設定を行うコマンドが選択され，エージェントシステム上で実行されている．

以上の手順を繰り返すことにより，メタシナリオ，シナリオの実行がなされる．

2.3　実現される機能

本節では，このアーキテクチャを用いることによって実現できるいくつかの機能について，具体的な例を挙げて説明する．

2.3.1　全体の状況の観測による実行制御

大規模環境でシミュレーションを行う場合，エージェント数が膨大になったり，シミュレーションの舞台となる空間が広域になったりするだけでなく，仮想世界の中での時間も長くなると考えられる．それに伴い，シミュレーション全体の状況を観測することでシミュレーションの進行状況を把握し，それに従ってエージェントの動作を変化させるような機能が必要となってくる．たとえば，渋滞緩和のためのナビゲーションサービスを考えた場合，シミュレーション全体の状況を観測し，渋滞が発生していればサービスを開始するなどの機能がある．

このアーキテクチャでは，メタシナリオによって，エージェントシステムに対する環境設定・取得や，メタプログラミングによるエージェントの動作観測及び，制御を行うことができる．これらを用いれば，シミュレーションやエージェントの動作を観測し，それに従ってシナリオ実行制御を行うことで，シミュレーションの，全体の状況の観測による実行制御が実現できる．

2.3.2　エージェントの動作補正

シミュレーション実行時において，エージェントの期待されない動作や動作に関する既知の問題によって，シミュレーションの実行に影響を与える場合が起こり得る．このときに，動作補正を行うことができれば，シミュレーションの実行を円滑に行うことができると考えられる．

このアーキテクチャでは，メタプログラミングを行うことで，エージェント記述の実行制御を可能とする．問題が既知である場合には，修正するプログラムをあらかじめ記述することで回避できる．また，未知である場合にも，エージェント記述を表示することでデバッグを支援したり，その場でエージェント記述に変更を加えたりすることで，問題を回避するような機能が実現できる．

2.3.3　参加型シミュレーションの時間管理

参加型シミュレーションを行う中で，人間の操作するエージェントに何らかの意思決定を行う必要が生じた場合，シミュレーションの速度を遅くする必要がある．

このアーキテクチャ上では，メタシナリオインタプリタから，エージェント

システムの環境取得を行ったり，シナリオの実行状況を観測したりすることができる．これにより，当該エージェントが意思決定が必要な場面にあるかどうかを判定することができる．この判定に基づいて，エージェントシステムに対して環境設定を行うことによって，シミュレーション時間を遅くすることができる．また，速度を元に戻す場合も同様に，意思決定が完了したかどうかを判定し，エージェントシステムに対してシミュレーション時間を元に戻すという環境設定をすることで，時間管理機能が実現できる．

第3章　メタシナリオ

本研究では，前章で述べたようなアーキテクチャによって，メタプログラミングとシミュレーションの制御・管理を行う言語を，シナリオ記述言語Qに実装した．

ここで，エージェント記述言語としてQ言語を用いたのは，次の二つの理由による．ひとつは，Q言語がSchemeを母言語としており，Schemeの性質を利用すれば，メタプログラミングが容易に実装できるため．もうひとつは，Q言語は，大規模マルチエージェントプラットフォームとして，非常に多数のエージェントを扱うことのできるエージェントサーバであるCaribbeanとの結合もなされているためである．

本章では，このQ言語に対して実装された，メタシナリオに関する，機能や言語仕様などを詳しく説明する．

3.1　動作モデルと機能定義

本節では，メタシナリオの実行がどのようになされるか，その動作モデルを説明する．

すでに述べたとおり，メタシナリオはシナリオ記述言語Q上に実装した．Q言語では，エージェントの動作は拡張状態遷移機械モデルで記述され，次のように動作する．エージェントはいくつかの状態を持っており，シナリオの各状態では複数の事象を同時並行的に待ち受けている．その中のいずれかが観測されると，どれが観測されたかに応じて，状態遷移を行う．このような観測をQではキューと呼ぶ．また，状態遷移の際には外部に何らかの作用を及ぼす．Qではこのような外部への作用をアクションと呼ぶ．

メタシナリオはQ言語と同様の記述方式を採用している．つまり，メタシナリオもまた，拡張状態遷移機械モデルで記述される．メタシナリオの各状態において，観測の対象となるものは大きく分けて二つある．すなわち，エージェントシステムの環境と，エージェントのシナリオ実行状況である．ここでエージェントのシナリオ実行状況の観測は，シナリオの状態遷移及びそれに伴う外部への作用を観測するものである．メタシナリオの状態遷移に伴う作用としては，エージェントシステムの環境設定，エージェントのセンサ・アクションの起動，シナリオの実行開始・停止などがある．

メタシナリオにおける観測と，外部への作用を表 1 にまとめた．表 1 においては，?で始まるコマンドはキューを表し，！で始まるコマンドはアクションを表している．この表において，メタ環境とはメタシナリオ実行者のレベルの環境である．これは例えば，シミュレーション実行者からの入力受付を行う必要がある場合に用いることができる．表 1 の?observeScenario は，他のシナリオ実行状況の観測のいずれかの観測をもって観測とするものである．

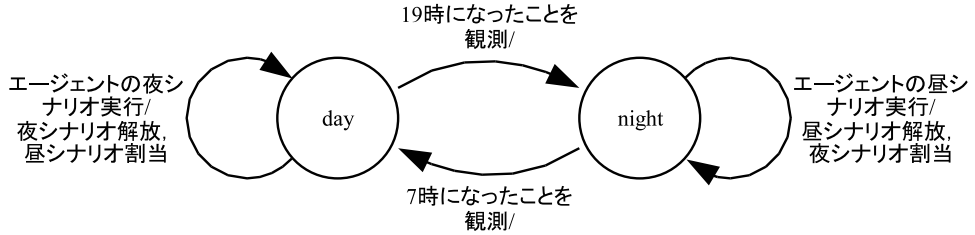

図 3: メタシナリオの例

メタシナリオの例を図 3 に示す．図中では，各状態を円で表し，状態遷移を矢印で表した．また，状態遷移のきっかけとなるキューの観測及び，状態遷移に伴うアクションの実行は，(キューの観測/アクションの実行) の形で状態遷移の矢印に添えて示した．

この例では，メタシナリオは昼と夜の 2 状態を持つ．また，エージェントには昼用のシナリオと夜用のシナリオが用意されている．このメタシナリオでは，昼に夜用のシナリオ実行を観測すると，昼用のものに切り替える．夜の場合も同様である．

表 1: メタシナリオの機能定義

シナリオ実行状況の観測	
?observeAction	指定された外部作用の実行を観測する
?observeCue	指定された観測の実行を観測する
?observeTransition	シナリオの状態遷移を観測する
?observeScenario	指定されたシナリオの実行を観測する
シナリオ実行の開始・停止	
!releaseScenario	エージェントからシナリオを解放する
!assignScenario	エージェントにシナリオを割り当てる
エージェントのセンサ・アクションの起動	
!runCue	エージェンのセンサを起動する
!runAction	エージェントのアクションを起動する
エージェントシステムの環境設定・取得	
!createAgent	エージェントを作成する
!deleteAgent	エージェントを削除する
!createCrowd	群集を作成する
!createAvatar	アバターを作成する
!startSimulation	シミュレーションを開始する
!stopSimulation	シミュレーションを停止する
?observeEnvironment	シミュレーション環境を観測する
!getEnvironment	シミュレーション環境情報を取得する
!setEnvironment	シミュレーション環境の設定をする
メタ環境の観測・情報取得	
?observeMetaEnvironment	メタ環境を観測する
!getMetaEnvironment	メタ環境情報を取得する

3.2 言語仕様

メタシナリオはQ言語のシナリオと同様の記述方式を採用しているため，本節では，Q言語について説明する．Q言語は，京都大学で開発されたシナリオ記述言語である．

Q言語では，他のエージェント記述言語と違い，エージェントの内部メカニズムを記述するのではなく，エージェントと外部とのインタラクションをシナリオとして記述し，エージェントに対してシナリオ記述に従って振舞うことを依頼するものである．

以下では，このQ言語の言語機能について説明する．

(1) **キュー**

キュー (cue) とは，外界への観測を行うものであり，インタラクションのきっかけとなるものである．また，外界への副作用はない．記述例を以下に示す

```
(?hear :name Taro :word "Hello")
```

ここで，リストの先頭の?hear はキューの名前を表し，残りの要素は引数となる．コロン (:) で始まる要素はキーワード引数の名前を表し，次の要素が引数の値となる．つまり，この例ではキーワード引数 name の値を Taro，word の値を"Hello"として，?hear が実行される．

このことは，次のアクションに関しても同様である．

(2) **アクション**

アクション (action) とは，外界へなんらかの作用を行うものである．記述例を以下に示す．

```
(!speak :sentence "Bye" :to Hanako)
```

これらキュー，アクションはユーザによって定義され，依頼を受け取ったエージェントがどのように動作するかはエージェントの実装によって異なる．

(3) **ガード付きコマンド**

ガード付きコマンド (guarded command) は，複数のキューに対して探索・待ち受けを行うのに用いられる．記述例を以下に示す．

```
(guard
 ((?hear :name Taro :word "Hello")
  (!speak :sentence "Hello" :to Taro))
```

```
((?hear :name Taro :word "Bye")
 (!speak :sentence "Bye" :to Taro))
(otherwise
 (!speak :sentence "Hi" :to Taro)))
```

上記の例では，(?hear :name Taro :word "Hello") 及び
(?hear :name Taro :word "Bye") を並行して待ち受け，いずれかのキューが観測されれば後続する形式が実行される．また，otherwise 節は，どのキューも観測されなかった場合に実行される．

(4) **シナリオ**

Q 言語のシナリオは状態遷移形式で記述され，状態遷移は go 式によってなされる．記述例を以下に示す．

```
(defscenario chat   (&key (message "Oh! Taro, bad boy!")
                          &pattern ($x #f))
  (greeting
   ((?hear :name $x :word "Hello") (go identification))
   ((?hear :name $x :word "Bye") (go farewell)))
  (identification
   ((?equal $x Taro) (!speak :sentence message :to Taro))
   (otherwise (!speak :sentence "Hello" :to (refval $x))))
  (farewell
   (otherwise (!speak :sentence "Bye" :to (refval $x)))))
```

このシナリオは greeting，identification，farewell の 3 つの状態から成り，chat という名前で定義される．シナリオ中の各状態は，ガード付きコマンドと同様の構文によって記述され，同様に処理される．例えば，このシナリオの状態 greeting では，この状態に遷移するとまず，(?hear :name $x :word "Hello") と (?hear :name $x :word "Bye") を並行して待ち受ける．

メタシナリオは，Q 言語と同様の言語仕様に従って記述される．すなわち，シミュレーションの進行状態が状態遷移によって記述される．そして，シナリオ実行状況の観測やエージェントシステムに対する環境設定などの機能は，表 1 で示したキュー・アクションとして定義・実装されている．

メタシナリオは，これらの定義済のキュー・アクションのみを用いて記述さ

れ，ユーザによって新たにキューまたはアクションを定義することはできない．ここで，これらのキュー，アクションを用いて，図 3 のメタシナリオを記述したものを図 4 に示す．

```
(defmetascenario day-and-night (&pattern (agent #f))
  (init
    (otherwise
      (!createCrowd :name 'man :population 10000)
      (!assignScenario :name 'dayScenario :agent man)
      (go day)))
  (day
    (; 仮想環境内での時間が 19:00 になるのを観測する
     (?observeEnvironment :name "time" :value "19:00")
     (go night))
    (; nightScenario の実行を観測し，エージェントを変数 agent に代入する
     (?observeScenario :name 'nightScenario :agent agent)
     ; エージェント agent からシナリオ nightScenario を解放する
     (!releaseScenario :name 'nightScenario :agent agent)
     ; エージェント agent にシナリオ dayScenario を割り当てる
     (!assignScenario :name 'dayScenario :agent agent)
     (go day))
  (night
    (; 仮想環境内での時間が 7:00 になるのを観測する
     (?observeEnvironment :name "time" :value "7:00")
     (go day))
    (; dayScenario の実行を観測し，エージェントを変数 agent に代入する
     (?observeScenario :name 'dayScenario :agent agent)
     ; エージェント agent からシナリオ dayScenario を解放する
     (!releaseScenario :name 'dayScenario :agent agent)
     ; エージェント agent にシナリオ nightScenario を割り当てる
     (!assignScenario :name 'nightScenario :agent agent)
     (go night)))
```

図 4: メタシナリオ記述例

図 4 では，まず状態 init でエージェントの作成など，シミュレーションの初期化を行った後，図 3 で示したようなシナリオの切り替え管理を行う．

3.3 実装

本研究では，これまでに述べてきたような機能を持つメタシナリオを，Q言語処理系上で実装を行った．

実装に当たっては，メタプログラミングの実装が容易であること，Q言語処理系がSchemeで実装されていることを考慮し，言語としてSchemeを選択した．

実装は以下のように行った．

- シナリオインタプリタの実装
  Q言語処理系既存のものでは，メタプログラミングの実装が困難であった．そこで，シナリオやシナリオコマンドを読み込んで評価し，実行を行うシナリオインタプリタ (`s-eval`) を実装した．ここでは，与えられたデータがアクションやキューであれば実行し，シナリオや状態定義であればコマンドを切り分ける．そしてそれぞれに対して，自身を再帰的に呼び出す．
- メタシナリオインタプリタの実装
  `s-eval`同様の処理を行う，メタシナリオやメタシナリオコマンドを読み込んで評価し実行を行う，メタシナリオインタプリタ`ms-eval`実装した．
- メタプログラミングの実装
  シナリオに対するメタプログラミングは，次のように`s-eval`を変更することで実装した．まず，メタシナリオからは，?observeActionや? observeCueなどのシナリオ観測依頼を受け取る．この情報は，シナリオ実行中保持しておく．そして`s-eval`が呼び出されたとき，与えられたデータとメタシナリオからのシナリオ観測依頼を比較することで，実行しようとしているコマンドがメタシナリオで観測されているかどうかを判断する．そして観測されていればメタシナリオのキューが観測されたことをメタシナリオに通知し，観測されていなければそのまま実行する．
- その他の機能
  その他の機能に関しては，エージェントシステムとの通信を必要とするため，エージェントシステムに応じて自由に記述できるようにする必要がある．
  シナリオ関しては，Q言語処理系に用意されているQコネクタをそのまま呼び出すことで実装し，メタシナリオのシミュレーション環境に対するキュー・アクションに関してもそれと同等のものを作成した．
  また，メタシナリオのエージェント等の作成機能はQ言語処理系で用意さ

れているものを用い，シナリオの割り当ては上に述べた s-eval を起動することで実装した．

これらのプログラムのソースコードは，本稿の付録に掲載した．

メタシナリオを用いてシミュレーションを行うには，まずQ言語を用いる場合と同様に，Qコネクタを記述する．Qコネクタは，Q言語処理系からのキュー・アクション・ガード付きコマンドの実行依頼や，エージェントやアバターの作成依頼をエージェントシステム上で実行するように変換を行うものである．さらにQコネクタに加えて，メタシナリオによるシミュレーション環境に対するキュー・アクションの実行をエージェントシステムで実行できるよう，変換を行う記述をする．以上の記述を行うことで，シミュレーションを実行できる環境を構築することができる．そして，必要なシナリオ及びメタシナリオの記述を行い，メタシナリオの処理を開始することで，シミュレーションが実行される．

第4章　評価

本章では，これまでの章で説明してきたメタシナリオの具体的な記述例として，メタシナリオによって実現される特徴的な機能の一つである時間管理機能について取り上げる．この時間管理機能について，簡単なシナリオ及びメタシナリオを記述することにより，実現できることを示した．また，本章でいくつか状態遷移図を記述したが，記述法は前章の図 3と同様である．

4.1　シミュレーションの説明

この例で行うシミュレーションは，京都市に見られるような格子状の道路上に，エージェントを配置し通行させるものである．この中にアバターを配置し，参加型シミュレーションを行う．アバターが特定の交差点に近づいたら，意思決定を行うためシミュレーション速度を遅くする．そして右折あるいは左折の決定をしたら速度を元に戻す．以上のような時間管理を行うメタシナリオを記述する．

このシミュレータのGUIを図5に示す．図中，右側のウィンドウが道路となっている．また，道路上の小さな黒い円が車エージェント，色の違う円がアバターとなっている．この道路は右端と左端，上端と下端がつながっており，エージェントとアバターはこの中を移動し続ける．図中の左側のウィンドウは，アバター

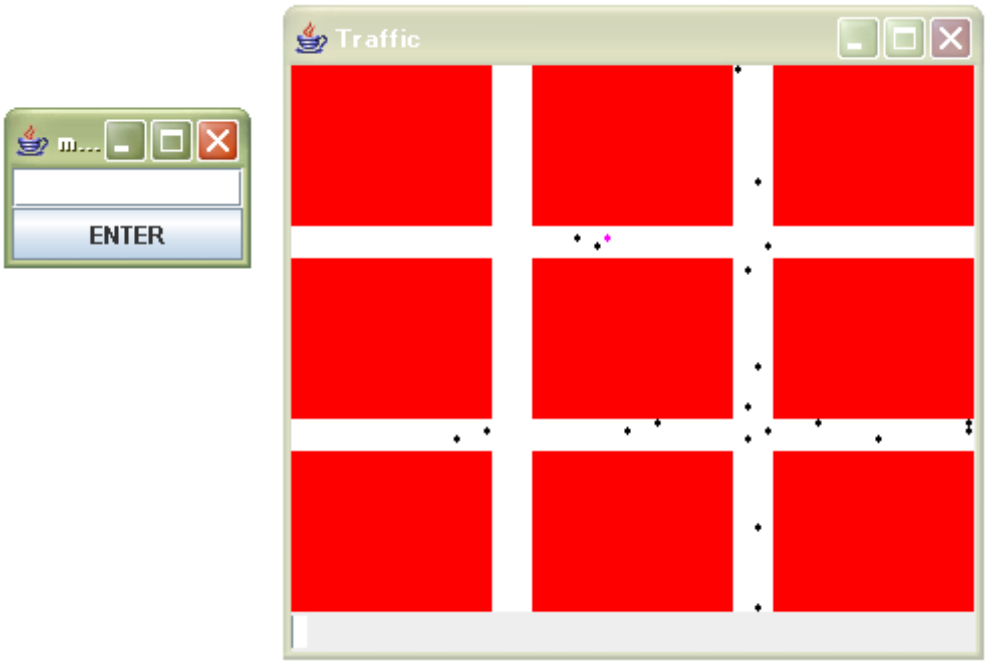

図 5: シミュレータ GUI

の操作ウィンドウである．このウィンドウ上のテキストボックスに「right」あるいは「left」と入力し ENTER をクリックすることで，アバターに右折または左折の指示を出すことができる．

4.2 エージェントのシナリオ

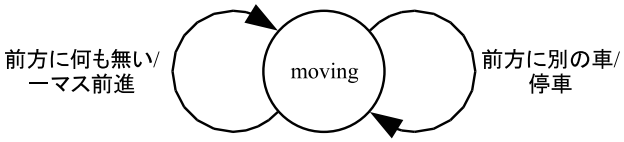

図 6: エージェントのシナリオ

エージェントの動作シナリオの状態遷移図を図 6 に示す．

エージェントのシナリオは非常に単純であり，一状態のみである．エージェントは通行する場所に関わらず，前方に別の車が存在すれば停止し，そうでなければ一マス前進する．これを繰り返すことで直進し続ける．

4.3 アバターのシナリオ

アバターのシナリオの状態遷移図を図 7 に示す．アバターもエージェント同様，前方に別の車があるときのみ停止し，そうでなければ進み続ける．しかし一方で，アバターでは図 5 で示した GUI によって，ユーザからの入力を受け付けている．この入力を読み取ることで，交差点で右折あるいは左折する機能を持つ．入力がなかった場合はそのまま直進する．

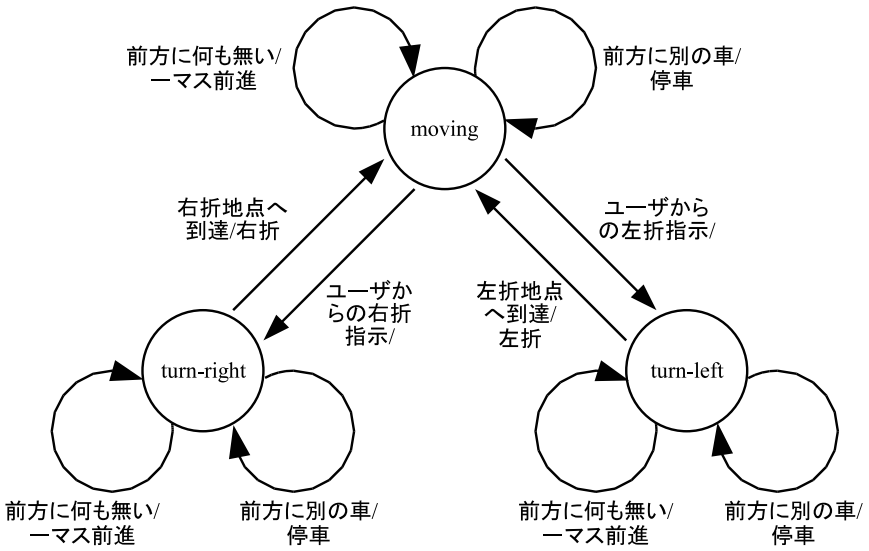

図 7: アバターのシナリオ

図 7 で示したように，ユーザから「右折する」という指示があった場合は `turn-right` 状態へ,「左折する」という指示があった場合は `turn-left` 状態へ遷移する．`turn-right`/`turn-left` 状態では，交差点の右折/左折が可能な地点まで前進を続ける．そして到達したら，右折/左折を行い `moving` 状態へと遷移する．

4.4 時間管理を行うメタシナリオ

以上で説明したシミュレーションにおいて，アバターが特定の交差点に近づくことで右折するか左折するかという意思決定が発生するとする．この状況下において，アバターが交差点の近くにいて意思決定を行っている間だけシミュレーションの速度を遅くし，意思決定が完了するとシミュレーション速度を元に戻したい．

このような時間管理を行うメタシナリオを記述した．状態遷移図は図 8 に示す．さらに，図 8 のように動作するメタシナリオを図 9 に示した．

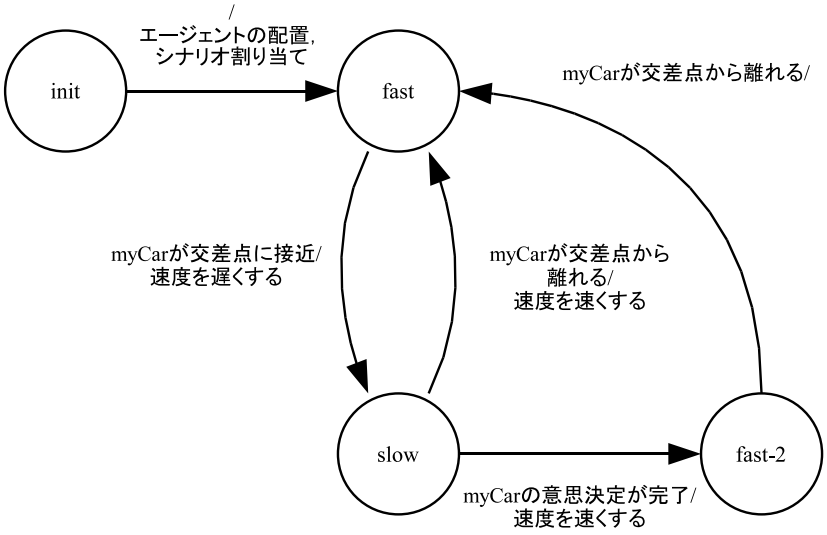

図 8: 時間管理のメタシナリオの状態遷移図

図 8 に示したメタシナリオの各状態について，詳しく説明する．

- `init`

  シミュレーションの初期化を行う状態である．シミュレーション上に名前 `Cars` で定義されるエージェント群を 20 体配置し，同時に名前 `myCar` で定義されるアバターも配置する．そしてそれぞれに前述のエージェントのシナリオまたはアバターのシナリオを与え，シミュレーションの実行を開始する．

- `fast`

  シミュレーションが高速に進行している状態である．この状態ではアバターは意思決定を行う時間を確保することはできない．ここで，?observeEnvironment によってシミュレータを観測することで，アバターが特定の交差点に近づくのを観測し，観測されたらシミュレーション速度を落として `slow` へと遷移する．

  この記述例では，?observeEnvironment に引数として `(list myCar 1 1)` を与えることによって，アバターが図 5 中の左上 (左から 1 番目，上から 1 番目) の交差点に近づくのを観測している．

```
(defmetascenario meta-traffic ()
  (init
   (otherwise
    (!createCrowd :name 'Cars :population 20
                  :agentmanager (repeat 20 "rmi://localhost/am0"))
    (!createAvatar :name 'myCar
                   :agentmanager "rmi://localhost/am0")
    (!assignScenario :name 'car :agent Cars)
    (!assignScenario :name 'car-avatar :agent myCar)
    (!startSimulation)
    (go fast)))
  (fast
   (; myCar が交差点 (1,1) に近づくのを観測する
    (?observeEnvironment :name 'onCrossing :args (list myCar 1 1))
    ; シミュレーション速度を遅くする
    (!setEnvironment :name 'sim-speed :args "slow")
    (go slow)))
  (slow
   (; myCar が交差点 (1,1) から遠ざかるのを観測する
    (?observeEnvironment :name 'outOfCrossing :args (list myCar 1 1))
    ; シミュレーション速度を速くする
    (!setEnvironment :name 'sim-speed :args "fast")
    (go fast))
   (; myCar が car-avatar シナリオの turn-right 状態に遷移するのを観測する
    (?observeTransition :scenario 'car-avatar :scene 'turn-right
                        :agent myCar)
    ; シミュレーション速度を速くする
    (!setEnvironment :name 'sim-speed :args "fast")
    (go fast-2))
   (; myCar が car-avatar シナリオの turn-left 状態に遷移するのを観測する
    (?observeTransition :scenario 'car-avatar :scene 'turn-left
                        :agent myCar)
    ; シミュレーション速度を速くする
    (!setEnvironment :name 'sim-speed :args "fast")
    (go fast-2)))
  (fast-2
   (; myCar が交差点 (1,1) から遠ざかるのを観測する
    (?observeEnvironment :name 'outOfCrossing :args (list myCar 1 1))
    (go fast))))
```

図 9: 時間管理のメタシナリオ

- slow
  シミュレーションの速度を落としている状態であり，この間にアバターの意思決定がなされる．意思決定完了の判定は，アバターの状態が turn-right あるいは turn-left に遷移したときである．このときユーザからの入力受付が完了し，意思決定が完了したとみなされる．これは，?observeTransition によって，アバターのシナリオの状態遷移を観測することによって判定される．意思決定の完了が判定されたとき，あるいはアバターが交差点から遠ざかったとき，シミュレーションの速度を元に戻す．
- fast-2
  アバターの意思決定は完了し，シミュレーションは高速に進行しているが，アバターがまだ交差点から遠ざかってない状態である．意思決定が完了後直ちに fast へ遷移すると，そのまま再び slow へと遷移してしまう．これを防ぐため，この状態でアバターが交差点から遠ざかるのを待って fast へと遷移する．

第5章　おわりに

本研究では，大規模マルチエージェントシステムを用いて，大規模環境下でシミュレーションを行うことを考えて，それらを行うために必要と考えられる機能の実現を目指した．

この実現のために，マルチエージェントシステムのエージェント記述言語に対するメタプログラミングを行うというアプローチを取った．さらに，メタプログラミングと，エージェントシステムに対する環境設定や取得といったシミュレーション環境の制御・管理機能とを統合させて，シミュレーション全体の制御を行うアーキテクチャを設計した．

そして，エージェント記述言語としてシナリオ記述言語Qを用いて，Q言語に対してメタプログラミングとエージェントシステムの制御・管理機能を統合して行う，メタシナリオの機能を明確に定義し実装を行った．

これにより，次のような機能が実現された．

- メタプログラミングによるシナリオ実行制御方式
  メタシナリオでは，エージェントの観測・制御及び，エージェントシステムの観測・制御を統合して行うアーキテクチャを用いている．このうち，エー

ジェントの観測・制御に関しては，エージェントの動作を規定するものであるエージェント記述言語に対してメタプログラミングを行う．メタプログラミングによって，柔軟なエージェントの観測・制御が可能となる．このアーキテクチャにより，シミュレーション全体の状況を把握し，その状況やエージェントのシナリオ実行の観測に基づいたシナリオの制御・管理ができる．これによって，シミュレーションをどのように進行させるかを記述することができる．また，それに伴った，エージェントの動作の変更の管理や，動作修正，デバッグ機能などが実現できる．

- シナリオ実行制御方式を用いたシミュレーションの時間管理
  メタシナリオではエージェントシステムに対する観測機能として，シナリオ実行制御によるエージェントの行動の観測及び，シミュレータの状況の観測を行うことができる．この機能によって，シミュレーションの状況を観測してシミュレーション速度を変更する必要がある状況にあるかどうかを判断することができる．さらに，エージェントシステムに対して環境設定行う機能を持ち，シミュレータ状況の観測に基づいて環境設定を行うことができる．この機能によって，シミュレーションがシミュレーション速度を変更する必要がある状況にある場合は，エージェントシステムに対して速度を変更させることができる．このようにして，時間管理機能が実現できる．

そして，時間管理機能に関しては，実際にシナリオ及びメタシナリオを記述し，簡単なシミュレーションを行うことで，実現可能であることを示した．しかしながら，これらの実装はQ言語処理系上での実装にとどまっており，大規模マルチエージェントシステムとして用いることはできない．また，実際のシミュレーションを想定したメタシナリオの評価も十分とは言えない．

そこで，今後の課題として，Q言語と Caribbean の結合システムに対してメタシナリオを適用するとともに，大規模マルチエージェントシステムを想定したメタシナリオの総合的な評価が必要であると考えられる．

謝辞

本研究を行うに当たって，熱心にご指導していただきました石田亨教授に厚く御礼を申し上げます．

また，日頃より議論していただき，有益な助言を与えてくださいました石田研究室の皆様方に心より感謝致します.

参考文献

[1] Ishida, T.: Society-Centered Design for Socially Embedded Multi-Agent Systems, *International Workshop on Cooperative Information Agents (CIA-04), Lecture Notes in Computer Science*, pp. 16–29 (2004).

[2] Ishida, T.: Q: A Scenario Description Language for Interactive Agents, *IEEE Computer*, Vol. 35, No. 11, pp. 42–47 (2002).

[3] Yamamoto, G. and Nakamura, Y.: Architecture and Performance Evaluation of a Massive Multi-Agent System, *International Conference on Autonomous Agents (Agents-99)*, pp. 319–325 (1999).

[4] Yamamoto, G. and Tai, H.: Performance Evaluation of an Agent Server Capable of Hosting Large Numbers of Agents, *International Conference on Autonomous Agents (Agents-01)*, pp. 363–369 (2001).

[5] 中島悠, 椎名宏徳, 山根昇平, 山本晃成, 石田亨: 大規模マルチエージェントシミュレーションのためのプラットフォーム構築, 合同エージェントワークショップ＆シンポジウム (JAWS2004), pp. 59–65 (2004).

付録：ソースコード

A.1　メタシナリオ処理系

A.1.1　メタシナリオインタプリタ

```
;; start meta scenario
(define (start filename)
  (ms-eval (read (open-input-file filename))))

(define (load-meta filename)
  (letrec ((load-meta-0
            (lambda (port)
              (let ((exp (read port)))
                (if (eof-object? exp)
                    #t
                    (begin
                      (cond
                       ((not (list? exp))
                        (eval exp))
                       ((eq? (car exp) 'defmetascenario)
                        (eval
                         `(define (,(cadr exp))
                            (make-scenario-observer ',exp)
                            (future (ms-eval ',exp)))))
                       (else (eval exp)))
                      (load-meta-0 port)))))))
    (load-meta-0 (open-input-file filename))))

;; +-------------------------+
;; | META SCENARIO EVALUATOR |
;; +-------------------------+

(define *meta-cue-action-list*
  '(!createAgent
    !deleteAgent
    !createCrowd
    !createAvatar
    ?observeAction
    !runAction
    ?observeCue
    !runCue
    ?observeTransition
    ?observeScenario
    !releaseScenario
    !assignScenario
    !startSimulation
    !stopSimulation
    !setEnvironment
    ?observeEnvironment
    !getEnvironment))

(define *meta-scene* #f)
```

```
(define (ms-eval exp)
  (cond
   ((and (list? exp) (equal? 'defmetascenario (car exp)))
    (ms-eval-scenario exp))
   ((and (list? exp) (equal? 'guard (car exp)))
    (ms-eval-gurad exp))
   ((and (list? exp) (equal? 'begin (car exp)))
    (ms-eval-begin (cdr exp)))
   ((and (list? exp) (eq? (car exp) 'go))
    (if (not (eq? *meta-scene* (cadr exp)))
        (synchronized *observation-queue-lock*
                      (clear-observation-queue)))
    (set! *meta-scene* (cadr exp))
    (ms-eval-scene
     (eval (symbol-append "-" 'meta
                          (cadr exp)))))
   ((and (list? exp) (memq (car exp) *meta-cue-action-list*))
    (apply (eval (car exp)) (cdr exp)))
   (else (eval exp))))

;; evaluate scenario
(define (ms-eval-scenario scenario)
  (let* ((scenario-args (list-ref scenario 2))
         (scene-list (list-tail scenario 3))
         (init-scene (car scene-list)))
    (for-each
     (lambda (scene)
       (eval `(define ,(symbol-append "-"
                                      'meta
                                      (car scene))
                ',scene)))
     scene-list)
    (set! *meta-scene* (car init-scene))
    (ms-eval-scene init-scene))
  (set! *meta-scene* #f))

;; evaluate scene
(define (ms-eval-scene scene)
  (let ((scene-name (list-ref scene 1)))
    (ms-eval-guard (cdr scene))))

;; evaluate guard
(define (ms-eval-guard guard)
  (if (and (= (length guard) 1)
           (list? (car guard))
           (equal? (caar guard) 'otherwise))
      (ms-eval `(begin ,@(cdar guard)))
      (let ((cue-id
             (ms-run-guard (make-meta-guard-request guard))))
        (ms-eval `(begin ,@(cdr (list-ref guard cue-id)))))))

(define (make-meta-guard-request guard)
```

```
  (map
   (lambda (cue-action-list)
     (let ((cue (car cue-action-list)))
       (cond ((and (list? cue)
                   (memq (car cue) (list '?observeCue
                                         '?observeAction
                                         '?observeTransition
                                         '?observeScenario
                                         '?observeEnvironment)))
              (cons 'cue cue))
             ((eq? cue 'otherwise)
              '(cue ?timeout))
             (else
              (cons 'exp (cdr cue))))))
   guard))

(define *time-out* 300)
(define *time-out-counter* 0)

(define (ms-run-guard guard)
  (set! *time-out-counter* 0)
  (do ((cue-id 0)
       (guard-1 guard)
       (observed-num #f))
      ((or observed-num (ms-run-guard-1 (car guard-1)))
       (if observed-num observed-num cue-id))
    (cond
     ((null? (cdr guard-1))
      (sleep 0.01)
      (set! cue-id 0)
      (set! guard-1 guard)
      (set! observed-num (check-observation guard)))
     (else
      (set! cue-id (+ cue-id 1))
      (set! guard-1 (cdr guard-1))))))

(define (check-observation guard)
  (if (not (null? *observation-queue*))
      (let* ((observation (synchronized *observation-queue-lock*
                                        (dequeue-observation)))
             (observed-num (car observation))
             (cue (list-ref guard observed-num))
             (agent (cadr observation))
             (agent-var (get-keyword-arg :agent (cddr cue) #f))
             (wait (caddr observation)))
        (if agent-var
            (eval `(set! ,agent-var ,agent)))
        (invoke wait 'finished)
        observed-num)
      #f))

(define (ms-run-guard-1 cue)
  (cond
```

```
   ((eq? (car cue) 'cue)
    (apply (eval (cadr cue)) (cddr cue)))
   ((eq? (car cue) 'exp)
    (ms-eval (cdr cue)))
   (else
    (ms-eval cue))))

;; evaluate begin
(define (ms-eval-begin exp)
  (cond ((null? exp))
        ((null? (cdr exp)) (ms-eval (car exp)))
        (else (ms-eval (car exp))
              (ms-eval-begin (cdr exp)))))

;; +-------------------+
;; | SCENARIO OBSERVER |
;; +-------------------+

(define *observation-queue* '())
(define *observation-queue-lock* #f)

(define *released-list* '())

; (make-scenario-observer
;  '(defmetascenario ()
;   (init
;    (otherwise
;     (!createCrowd :name 'Ants :population 10
;                :agentmanager (repeat 10 "rmi://localhost/am0"))
;     (!assignScenario :name antsort :agent Ants)
;     (go scene1)))
;   (scene1
;    ((?observeAction :name '!putsugar :agent agentx)
;     (!runAction :name '!hoge :agent agentx)
;     (!runAction :name '!putsugar :agent agentx)
;     (go scene1))
;    ((?observeScenario :name 'antsort :agent agentx)
;     (!runAction :name '!hoge :agent agentx)
;     (go scene1))))

; =>

; (define s-eval-0 s-eval)
; (define (s-eval agent scenario-name exp)
;   (cond
;    ((cadr
;      (assoc (list agent scenario-name)
;           *released-list*)))
;    ((eq? *meta-scene* 'init)
;     (let
;       ((observed-num
;         (observed agent scenario-name exp
;                   '(otherwise))))
```

```
;       (if observed-num
;         (send-observation agent scenario-name exp
;                           observed-num)
;         (s-eval-0 agent scenario-name exp))))
;    ((eq? *meta-scene* 'scene1)
;     (let
;       ((observed-num
;         (observed agent scenario-name exp
;                   '((?observeAction name: (quote !putsugar) agent:
;                                     agentx)
;                     (?observeScenario name: (quote antsort) agent:
;                                       agentx)))))
;       (if observed-num
;         (send-observation agent scenario-name exp
;                           observed-num)
;         (s-eval-0 agent scenario-name exp))))
;    (else (s-eval-0 agent scenario-name exp))))

(define (make-scenario-observer meta-s)
  (let ((make-scenario-observer-0
         (lambda (scene)
           `((eq? *meta-scene* ',(car scene))
             (let ((observed-num
                    (observed agent scenario-name exp
                              ',(map car (cdr scene)))))
               (if observed-num
                   (begin
                     (send-observation agent scenario-name exp
                                       observed-num)
                     (s-eval-0 agent scenario-name exp))
                   (s-eval-0 agent scenario-name exp)))))))
    (eval
     `(define s-eval-0 s-eval))
    (eval
     `(define (s-eval agent scenario-name exp)
        (cond
         ((released? agent scenario-name))
         ,@(map make-scenario-observer-0 (cdddr meta-s))
         (else (s-eval-0 agent scenario-name exp)))))))

(define (observed agent scenario-name exp meta-cue-list)
  (letrec
      ((observed-0
        (lambda (num mcl)
          (if
           (null? mcl)
           #f
           (if
            (observed-1? agent scenario-name exp (car mcl))
            num
            (observed-0 (+ num 1) (cdr mcl)))))))
    (observed-0 0 meta-cue-list)))
```

```
(define (observed-1? agent scenario-name exp meta-cue)
  (cond
   ((and (list? meta-cue)
         (eq? (car meta-cue) '?observeAction))
    (if (and (list? exp) (memq (car exp) *action-list*))
        (match-args agent exp (cdr meta-cue))
        #f))
   ((and (list? meta-cue)
         (eq? (car meta-cue) '?observeCue))
    (if (and (list? exp) (memq (car exp) *cue-list*))
        (match-args agent exp (cdr meta-cue))
        #f))
   ((and (list? meta-cue)
         (eq? (car meta-cue) '?observeTransition))
    (and (list? exp) (eq? (car exp) 'go)
         (eq? scenario-name
              (eval (get-keyword-arg :scenario (cdr meta-cue) #f)))
         (eq? (cadr exp)
              (eval (get-keyword-arg :scene (cdr meta-cue) #f)))
         (eq? agent
              (eval (get-keyword-arg :agent (cdr meta-cue) #f)))))
   ((and (list? meta-cue)
         (eq? (car meta-cue) '?observeScenario))
    (eq? scenario-name (eval (get-keyword-arg :name (cdr meta-cue) #f))))
   (else #f)))

(define (match-args agent exp meta-cue-args)
  (if (null? meta-cue-args)
      #t
      (let ((name (eval (car meta-cue-args)))
            (value (cadr meta-cue-args)))
        (if (cond
             ((eq? name ':name)
              (eq? (eval value) (car exp)))
             ((eq? name ':agent)
              #t)
             (else
              (eq? value (eval (get-keyword-arg name (cdr exp) #f)))))
            (match-args agent exp (cddr meta-cue-args))
            #f))))

(define-class <WaitMeta> ()
  ((waitMeta) :: <void>
   (synchronized this
                 (invoke this 'wait)))
  ((finished) :: <void>
   (synchronized this
                 (invoke this 'notify))))

(define (send-observation agent scenario-name exp num)
  (let ((wait (make <WaitMeta>)))
    (synchronized
     *observation-queue-lock*
```

```
     (enqueue-observation (list num agent wait scenario-name exp)))
    (invoke wait 'waitMeta)))

(define (enqueue-observation elem)
  (set! *observation-queue* (append *observation-queue* (list elem)))
  *observation-queue*)

(define (dequeue-observation)
  (if (null? *observation-queue*)
      #f
      (let ((ret (car *observation-queue*)))
        (set! *observation-queue* (cdr *observation-queue*))
        ret)))

(define (clear-observation-queue)
  (cond
   ((not (null? *observation-queue*))
    (dequeue-observation)
    (clear-observation-queue))))

;; +------------------------------+
;; | META SCENARIO CUES & ACTIONS |
;; +------------------------------+

;; actions
(define (!createAgent . args)
  (let ((name (get-keyword-arg :name args #f)))
    (eval (list 'define (eval name)
                (apply make-agent (eval name) args)))))

(define (!createCrowd . args)
  (let ((name (get-keyword-arg :name args #f)))
    (eval (expand-defcrowd (eval name) args))))

(define (!createAvatar . args)
  (let ((name (get-keyword-arg :name args #f)))
    (eval (list 'define (eval name)

        (apply make-avatar (eval name) args)))))

(define (!assignScenario . args)
  (let ((scenario-name (eval (get-keyword-arg :name args #f)))
        (agent (eval (get-keyword-arg :agent args #f))))
    (if (list? agent)
        (for-each
         (lambda (agent)
           (released-off agent scenario-name)
           (future (s-eval agent scenario-name (eval scenario-name)))
           (sleep 0.1))
         agent)
        (begin
          (released-off agent scenario-name)
          (future (s-eval agent scenario-name (eval scenario-name)))))))
```

```
(define (!releaseScenario . args)
  (let ((scenario-name (eval (get-keyword-arg :name args #f)))
        (agent (eval (get-keyword-arg :agent args #f))))
    (if (list? agent)
        (for-each
         (lambda (agent)
           (released-on agent scenario-name))
         agent)
        (released-on agent scenario-name))))

(define (!runAction . args)
  (let ((name (get-keyword-arg :name args #f))
        (agent (get-keyword-arg :agent args #f)))
    (eval
     `(request-action ,name ,agent ,@args))))

(define (!runCue . args)
  (let ((name (get-keyword-arg :name args #f))
        (agent (get-keyword-arg :agent args #f)))
    (eval
     `(request-cue ,name ,agent ,@args))))

(define (?observeAction . args) #f)
(define (?observeCue . args) #f)
(define (?observeTransition . args) #f)
(define (?observeScenario . args) #f)

(define (!startSimulation)
  (start-simulation))

(define (!stopSimulation)
  (stop-simulation))

(defmacro define-observe-environment (name val-name . body)
  `(define (,(string->symbol
              (string-append
               "observe-environment-" (symbol->string name)))
            ,val-name)
     ,@body))

(define (?observeEnvironment . args)
  (let ((name
         (string->symbol
          (string-append
           "observe-environment-"
           (symbol->string (eval (get-keyword-arg :name args #f))))))
        (args (get-keyword-arg :args args #f)))
    (eval (list name args))))

(defmacro define-set-environment (name val-name . body)
  `(define (,(string->symbol
              (string-append
```

```
              "set-environment-" (symbol->string name)))
            ,val-name)
     ,@body))

(define (!setEnvironment . args)
  (let ((name
         (string->symbol
          (string-append
           "set-environment-"
           (symbol->string (eval (get-keyword-arg :name args #f))))))
        (args (get-keyword-arg :args args #f)))
    (eval (list name args))))

(defmacro define-get-environment (name val-name . body)
  `(define (,(string->symbol
              (string-append
               "get-environment-" (symbol->string name)))
            ,val-name)
     ,@body))

(define (!getEnvironment . args)
  (let ((name
         (string->symbol
          (string-append
           "get-environment-"
           (symbol->string (eval (get-keyword-arg :name args #f))))))
        (args-name (get-keyword-arg :args args #f)))
    (eval `(set! ,args-name (,name ,args-name)))))

(define (?timeout . args)
  (if (= *time-out* *time-out-counter*)
      #t
      (begin
        (set! *time-out-counter* (+ *time-out-counter* 1))
        #f)))

(defmacro delete-keyword-arg (key args)
  `(delete-keyword-arg-0 ,key ,args))

(define (delete-keyword-arg-0 key args)
  (letrec
      ((dka (lambda (left right)
              (cond
               ((null? right) left)
               ((eq? (car right) key)
                (append left (cddr right)))
               (else
                (dka `(,@left ,(car right) ,(cadr right))
                     (cddr right)))))))
    (dka '() args)))

(define (released-on agent scenario-name)
  (set! *released-list*
```

```
        (remove *released-list*
                (assoc (list agent scenario-name) *released-list*))))

(define (released-off agent scenario-name)
  (set! *released-list*
        (cons (list (list agent scenario-name) #t) *released-list*)))

(define (released? agent scenario-name)
  (not (assoc (list agent scenario-name) *released-list*)))

(define (remove list elem)
  (letrec ((remove-0
            (lambda (list res)
              (if (null? list)
                  (reverse list)
                  (if (equal? (car list) elem)
                      (remove-0 (cdr list) res)
                      (remove-0 (cdr list) (cons (car list) res)))))))
    (remove-0 list '())))

;; for test.

(define (hoge1)
  (apply !createAgent '(:name 'Ants :population 10
                              :agentmanager "rmi://localhost:am0")))

(define (create)
  (apply !createCrowd
         '(:name 'Ants
                 :population 10
                 :agentmanager (repeat 10 "rmi://localhost/am0"))))

(define (assign)
  (apply !assignScenario
         '(:name 'antsort :agent Ants)))
```

A.1.2 シナリオインタプリタ

```
;; +--------------------+
;; | SCENARIO EVALUATOR |
;; +--------------------+

(define (s-eval agent scenario-name exp . args)
  (cond
   ((and (list? exp) (equal? 'defscenario (car exp)))
    (apply s-eval-scenario agent scenario-name exp args))
   ((and (list? exp) (equal? 'guard (car exp)))
    (s-eval-guard agent scenario-name (cdr exp)))
   ((and (list? exp) (equal? 'begin (car exp)))
    (s-eval-begin agent scenario-name (cdr exp)))
   ((and (list? exp) (memq (car exp) *action-list*))
    (eval `(request-action ',(car exp) ,agent ,@(cdr exp))))
    ;(apply request-action (car exp) agent (cdr exp)))
   ((and (list? exp) (memq (car exp) *cue-list*))
```

```
    (eval `(request-cue ',(car exp) ,agent ,@(cdr exp))))
    ;(apply request-cue (car exp) agent (cdr exp)))
   ((and (list? exp) (eq? (car exp) 'go))
    (s-eval-scene agent scenario-name
                  (eval (symbol-append "-"
                                       scenario-name
                                       (cadr exp)))))
   ((and (list? exp) (eq? (car exp) 'if))
    (s-eval-if agent scenario-name exp))
   (else (eval exp))))

;; evaluate scenario
(define (s-eval-scenario agent scenario-name scenario . args)
  (let* ((scenario-0 (replace-scenario-args agent scenario-name scenario))
         (scenario-name (list-ref scenario-0 1))
         (scenario-args (list-ref scenario-0 2))
         (scene-list (list-tail scenario-0 3))
         (init-scene (car scene-list)))
;;--
    (apply define-args agent scenario-name scenario-args args)
;;--
    (s-eval-scene agent scenario-name init-scene)))

(define (define-args agent scenario-name args . given-args)
  (call-with-values
    (lambda ()
      (split-args args '() '() '() '()))
    (lambda (args2 keys auxs patterns)
      (let ((given-key-args (list-tail given-args (length args2))))
        (apply define-scenario-args
               agent scenario-name args2 given-args)
        (apply define-scenario-key-args
               agent scenario-name keys given-key-args)
        (apply define-scenario-aux-args
               agent scenario-name auxs given-key-args)
        (apply define-scenario-pattern-args
               agent scenario-name patterns given-key-args)))))

(define (define-scenario-args agent scenario-name args2 . given-args)
  (if (null? args2)
      #t
      (begin
        (eval `(define ,(car args2) ,(car given-args)))
        (apply define-scenario-args
               agent scenario-name
               (cdr args2) (cdr given-args)))))

(define (define-scenario-key-args
          agent scenario-name keys . given-key-args)
  (for-each
   (lambda (key)
     (let ((key-name (car key))
           (key-value (cadr key)))
```

```
       (eval `(define ,key-name
                ,(get-keyword-arg
                  (string->keyword (symbol->string key-name))
                  given-key-args
                  key-value)))))
   keys))

(define (define-scenario-aux-args
          agent scenario-name auxs . given-key-args)
  (for-each
   (lambda (aux)
     (let ((aux-name (car aux))
           (aux-value (cadr aux)))
       (eval `(define ,aux-name
                ,(get-keyword-arg
                  (string->keyword (symbol->string aux-name))
                  given-key-args
                  aux-value)))))
   auxs))

(define (define-scenario-pattern-args
          agent scenario-name patterns . given-key-args)
  (for-each
   (lambda (pattern)
     (let ((pattern-name (car pattern))
           (pattern-value (cadr pattern)))
       (eval `(define ,pattern-name
                ,(get-keyword-arg
                  (string->keyword (symbol->string pattern-name))
                  given-key-args
                  pattern-value)))))
   patterns))

;; evaluate scene
(define (s-eval-scene agent scenario-name scene)
  (let ((scene-name (list-ref scene 1)))
    (s-eval-guard agent scenario-name
                  (replace-scene-args agent scenario-name (cdr scene)))))
;    (s-eval-guard agent scenario-name (cdr scene))))

;; evaluate guard
(define (s-eval-guard agent scenario-name guard)
  (if (and (= (length guard) 1)
           (list? (car guard))
           (eq? (caar guard) 'otherwise))
      (s-eval agent scenario-name (cons 'begin (cdar guard)))
      (let ((cue-id
             (request-guard agent
                            (make-guard-request guard))))
        (s-eval agent scenario-name
                `(begin ,@(cdr (list-ref guard cue-id)))))))

(define (make-guard-request guard)
```

```
  (map
   (lambda (cue-action-list)
     (let ((cue (car cue-action-list)))
       (cond ((and (list? cue)
                   (memq (car cue) *cue-list*))
              (cons 'cue cue))
             ((eq? cue 'otherwise)
              '(cue ?timeout))
             (else
              (cons 'exp (cdr cue))))))
   guard))

;; evaluate begin
(define (s-eval-begin agent scenario-name exp)
  (cond ((null? exp))
        ((null? (cdr exp)) (s-eval agent scenario-name (car exp)))
        (else (s-eval agent scenario-name (car exp))
              (s-eval-begin agent scenario-name (cdr exp)))))

;; evaluate if
(define (s-eval-if agent scenario-name exp)
  (let ((test (list-ref exp 1))
        (true-case (list-ref exp 2))
        (false-case (list-ref exp 3)))
    (if (s-eval agent scenario-name test)
        (s-eval agent scenario-name true-case)
        (s-eval agent scenario-name false-case))))

;; +-----------------+
;; | SCENARIO LOADER |
;; +-----------------+

(define (load-scenario filename)
  (letrec
      ((load-scenario-0
        (lambda (port)
          (let ((exp (read port)))
            (if (eof-object? exp)
                #t
                (begin
                  (if (and (list? exp)
                           (eq? (car exp) 'defscenario))
                      (begin
                        (eval `(define
                                 ,(list-ref exp 1)
                                 ',exp))
;;--
                        (define-scenario-symbol-replace exp)
;;--
                        (define-scene exp))
                      (eval exp))
                  (load-scenario-0 port)))))))
    (load-scenario-0 (open-input-file filename))))
```

```
(define (define-scene scenario)
  (for-each
   (lambda (scene)
     (eval
      `(define ,(symbol-append "-"
                               (cadr scenario)
                               (car scene))
         ',scene)))
   (list-tail scenario 3)))

;; test.

(define (test-expand-guard cue-action-list-list)
  `(case (request-guard
          self
          (list
           ,@(map (lambda (cue-action-list)
                    (let ((cue (car cue-action-list)))
                      (cond ((and (list? cue)
                                  (memq (car cue) *cue-list*))
                             `(list 'cue ',(car cue) ,@(cdr cue)))
                            ((eq? cue 'otherwise)
                             `(list 'cue '?timeout))
                            (else
                             `(list 'exp ,@(cdr cue))))))
                  cue-action-list-list)))
     ,@(do ((n 0 (+ n 1))
            (l cue-action-list-list (cdr l))
            (result '()
                    `(,@result ((,n) ,@(cdar l)))))
           ((null? l) result))
     (else (error "error in guard"))))

(define (test-start num)
  (s-eval (list-ref Ants num) 'antsort antsort))

(define (test-all agents)
  (for-each
   (lambda (agent)
     (future (s-eval agent 'antsort antsort)))
   agents))
```

A.1.3 その他のライブラリ

```
;; display exp-list and return last element of exp-list
(define (test . exp-list)
  (letrec
      ((test-0
        (lambda (ls)
          (cond
           ((null? ls) (newline) '())
           ((null? (cdr ls)) (display (car ls)) (newline) (car ls))
           (else (display (car ls)) (test-0 (cdr ls)))))))
```

```
    (test-0 exp-list)))

;; (symbol-append "-" 'hoge 'foo 'bar)
;; =>
;; 'hoge-foo-bar

(define (symbol-append s symbol . symbols)
  (letrec ((symbol-append-0
            (lambda (res symbols)
              (if (null? symbols)
                  res
                  (symbol-append-0
                   (string-append
                    res s (symbol->string (car symbols)))
                   (cdr symbols))))))
    (string->symbol
     (symbol-append-0 (symbol->string symbol) symbols))))

; (replace-symbol
;  '(hoge foo bar) '(a b c)
;  '(cond
;    ((eq? hoge 'hoge) (display "hoge"))
;    ((eq? foo 'foo)
;     (let ((hoge 10) (bar 20))
;       (display foo)
;       (lambda (foo) (+ hoge foo bar))))
;    (else
;     (display bar))))

; =>

; (cond ((eq? a 'hoge) (display "hoge"))
;  ((eq? b 'foo)
;   (let ((hoge 10) (bar 20))
;     (display b)
;     (lambda (foo) (+ hoge foo bar))))
;  (else
;   (display c)))

(define (replace-symbol-1 from-list to-list sym)
  (if (null? from-list)
      sym
      (if (eq? (car from-list) sym)
          (car to-list)
          (replace-symbol-1 (cdr from-list) (cdr to-list) sym))))

(define (rs-mapper from-list to-list remove-list)
  (letrec ((rs-mapper-0
            (lambda (f t f-res t-res)
              (if (null? f)
                  (lambda (ls) (replace-symbol f-res t-res ls))
                  (if (member (car f) remove-list)
                      (rs-mapper-0 (cdr f) (cdr t) f-res t-res)
```

```
                    (rs-mapper-0 (cdr f) (cdr t)
                                 (cons (car f) f-res)
                                 (cons (car t) t-res)))))))
    (rs-mapper-0 from-list to-list '() '())))

(define (replace-symbol from-list to-list ls)
  (if
   (not (list? ls))
   (replace-symbol-1 from-list to-list ls)
   (cond
    ((eq? (car ls) 'lambda)
     (if (list? (cadr ls))
         (map (rs-mapper from-list to-list (cadr ls)) ls)
         (map (rs-mapper from-list to-list '()) ls)))
    ((memq (car ls) '(let let* letrec))
     (map (rs-mapper from-list to-list (map car (cadr ls))) ls))
    ((eq? (car ls) 'quote) ls)
    (else
     (map (rs-mapper from-list to-list '()) ls)))))

;; +-------------------------+
;; | Argument Symbol Replace |
;; +-------------------------+

(define (split-arg-names args)
  (letrec ((split-arg-names-0
            (lambda (args-0 res)
              (if (null? args-0)
                  res
                  (case (car args-0)
                    ((&key &aux &pattern)
                     (split-arg-names-1 (cdr args-0) res))
                    (else
                     (split-arg-names-0 (cdr args-0) (cons (car args-0) res)))))))
           (split-arg-names-1
            (lambda (args-0 res)
              (if (null? args-0)
                  res
                  (case (car args-0)
                    ((&key &aux &pattern)
                     (split-arg-names-1 (cdr args-0) res))
                    (else
                     (split-arg-names-0 (cdr args-0) (cons (caar args-0) res))))))))
    (split-arg-names-0 args '())))

(define (define-scenario-symbol-replace scenario)
  (let ((scenario-name (cadr scenario))
        (args (caddr scenario)))
    (eval
     `(define ,(symbol-append "-"
                              scenario-name
                              'args-replace)
        ',(split-arg-names args)))))
```

```
(define (replace-scenario-args agent scenario-name exp)
  (let* ((from-list (eval (symbol-append "-"
                                         scenario-name
                                         'args-replace)))
         (to-list (map
                   (lambda (sym)
                     (symbol-append "-"
                                    (string->symbol (get-agent-name agent))
                                    scenario-name sym))
                   from-list)))
    (list (car exp)
          (cadr exp)
          (replace-arg-symbol from-list to-list (caddr exp))
          (replace-symbol from-list to-list (cadddr exp)))))

(define (replace-scene-args agent scenario-name scene)
  (let* ((from-list (eval (symbol-append "-"
                                         scenario-name
                                         'args-replace)))
         (to-list (map
                   (lambda (sym)
                     (symbol-append "-"
                                    (string->symbol (get-agent-name agent))
                                    scenario-name sym))
                   from-list)))
    (replace-symbol from-list to-list scene)))

(define (replace-arg-symbol from-list to-list args)
  (letrec
      ((replace-arg-symbol-1
        (lambda (arg)
          (cond
           ((list? arg)
            (cons (replace-symbol from-list to-list (car arg))
                  (cdr arg)))
           ((or (eq? arg '&key) (eq? arg '&aux) (eq? arg '&pattern))
            arg)
           (else
            (replace-symbol-1 from-list to-list arg))))))
    (map replace-arg-symbol-1 args)))

(define (replace-scenario-args-test agent scenario-name exp)
  (let ((replace-list (eval (symbol-append "-"
                                           scenario-name
                                           'args-replace))))
    `(replace-symbol
     ',replace-list
     ',(map
        (lambda (sym)
          (symbol-append "-" (string->symbol (get-agent-name agent))
                         scenario-name sym))
        replace-list)
```

```
    ',exp)))
```

A.2　時間管理システム

A.2.1　シナリオ・メタシナリオのキュー・アクションの実装

```
(define x 2)
(define y 2)
(define w 4)
(define l 20)
(define *sim-speed* 0.02)
(define *sim-start* #f)
(define *time* 0)

(define (traffic-sleep time)
  (let ((end (+ *time* time)))
    (do ()
        ((>= *time* end))
      (sleep *sim-speed*))))

(define (timer-start)
  (future
   (do ()
       (#f)
     (sleep *sim-speed*)
     (if *sim-start*
         (set! *time* (+ *time* 1))))))

(timer-start)

(define (tail ls)
  (if (null? ls)
      ls
      (if (null? (cdr ls))
          ls
          (tail (cdr ls)))))

(define (make-cyclic-list ls)
  (let ((ls2 ls))
    (set-cdr! (tail ls2) ls2)
    ls2))

(define *light-timer* (make-cyclic-list '(0 1 2 3 4 5 6 7 8 9)))

(define (start-light-timer)
  (future
   (do ()
       (#f)
     (set! *light-timer* (cdr *light-timer*))
     (traffic-sleep 50))))

(define (make-tf-list t-num . tf-num)
  (letrec ((make-t-list
```

```
            (lambda (t-num tf-num)
              (if (= 0 t-num)
                  (if (null? tf-num)
                      '()
                      (make-f-list (car tf-num) (cdr tf-num)))
                  (cons #t (make-t-list (- t-num 1) tf-num)))))
           (make-f-list
            (lambda (f-num tf-num)
              (if (= 0 f-num)
                  (if (null? tf-num)
                      '()
                      (make-t-list (car tf-num) (cdr tf-num)))
                  (cons #f (make-f-list (- f-num 1) tf-num))))))
    (make-t-list t-num tf-num)))

(define (make-ft-list f-num . tf-num)
  (apply make-tf-list 0 f-num tf-num))

(define *v-light* (make-tf-list 6 4))
(define *h-light* (make-tf-list 1 4 5))

(define (random n)
  (inexact->exact
   (floor
    (* n (invoke-static <java.lang.Math> 'random)))))

(define *map* (make <Collision> "Traffic" x y w l))

(define (set-flash x)
  (invoke *map* 'setFlash x))

(define (begin-summarize)
  (future
   (do ()
       (#f)
     (invoke *map* 'summarize)
     (sleep 10))))

'(begin-summarize)

(define (flash) (invoke *map* 'flash))

(define (random-x dir)
  (cond
   ((eq? dir 'vertical)
    (let ((randx (random (* x w))))
      (+ (* (floor (/ randx w)) l) l randx)))
   (else
    (random (* (+ (* x w) (* (+ x 1) l)))))))

(define (random-y dir)
  (cond
   ((eq? dir 'horizonal)
```

```
    (let ((randy (random (* y w))))
      (+ (* (floor (/ randy w)) l) l randy)))
   (else
    (random (* (+ (* y w) (* (+ y 1) l)))))))

(define (random-xy)
  (let ((dir (if (> (invoke-static <java.lang.Math> 'random)
                    0.5)
                 'horizonal 'vertical)))
    (cons (random-x dir) (random-y dir))))

(define (random-dx dir)
  (if (eq? dir 'vertical)
      0
      (if (> (invoke-static <java.lang.Math> 'random) 0.5) 1 -1)))

(define (random-dy dir)
  (if (eq? dir 'vertical)
      (if (> (invoke-static <java.lang.Math> 'random) 0.5) 1 -1)
      0))

(define (get-dir x y p)
  (let ((xx (modulo x (+ w l)))
        (yy (modulo y (+ w l))))
    (if (< xx l)
        'horizonal
        (if (< yy l)
            'vertical
            (if (> (invoke-static <java.lang.Math> 'random) p)
                'horizonal
                'vertical)))))

(define (get-dx y dir)
  (if (eq? dir 'horizonal)
      (if (< (modulo y (+ w l))
             (+ l (floor (/ w 2))))
          1 -1)
      0))

(define (get-dy x dir)
  (if (eq? dir 'vertical)
      (if (< (modulo x (+ w l))
             (+ l (floor (/ w 2))))
          -1 1)
      0))

(define (move-car car)
  (do ()
      (#f)
    (invoke car 'move)
    (traffic-sleep 1)))

(defagenthandler (name args agent)
```

```
  '(debug-output (list name args agent))
  (let* ((xy (random-xy))
         (x (get-keyword-arg :x args (car xy)))
         (y (get-keyword-arg :y args (cdr xy)))
         (dir (get-dir x y 0.5))
         (dx (get-dx y dir))
         (dy (get-dy x dir)))
         ;(dx (get-keyword-arg :dx args (get-dx y dir)))
         ;(dy (get-keyword-arg :dy args (get-dy x dir))))
    (do ()
        ((not (invoke *map* 'isCar x y)))
      (set! x (+ x dx))
      (set! y (+ y dy)))
    (set-attribute! agent 'dir dir)
    (set-attribute! agent 'dx dx)
    (set-attribute! agent 'dy dy)
    (set-attribute! agent 'car (make <Car> x y dx dy *map* 0))))

'(defactionadapter !move (self)
  (let ((car (get-attribute self 'car)))
    (invoke car 'changeDirection
            (get-attribute self 'dx)
            (get-attribute self 'dy))))

(defactionadapter !move (self)
  (traffic-sleep 1)
  (invoke (get-attribute self 'car) 'move))

(defactionadapter !stop (self)
  (traffic-sleep 2))

(defactionadapter !wait (self)
  (traffic-sleep 4))

(defactionadapter !turnRight (self)
  (let ((new-dx (- (get-attribute self 'dy)))
        (new-dy (get-attribute self 'dx))
        (new-dir (case (get-attribute self 'dir)
                   ((horizonal) 'vertical)
                   ((vertical) 'horizonal))))
    (set-attribute! self 'dx new-dx)
    (set-attribute! self 'dy new-dy)
    (set-attribute! self 'dir new-dir)
    (invoke (get-attribute self 'car)
            'changeDirection new-dx new-dy)))

(defactionadapter !turnLeft (self)
  (let ((new-dx (get-attribute self 'dy))
        (new-dy (- (get-attribute self 'dx)))
        (new-dir (case (get-attribute self 'dir)
                   ((horizonal) 'vertical)
                   ((vertical) 'horizonal))))
    (set-attribute! self 'dx new-dx)
```

```
    (set-attribute! self 'dy new-dy)
    (set-attribute! self 'dir new-dir)
    (invoke (get-attribute self 'car)
            'changeDirection new-dx new-dy)))

(defactionadapter !display (self &key (word #f))
  (debug-output (list self word)))

;; cue adapters

(defcueadapter ?timeout (self)
  (let-polling
   ((true #t))
   #t))

'(defcueadapter ?crossing (self)
  (let-polling
   ((crossed
     (case (get-attribute self 'dir)
       ((horizonal)
        (= 0 (modulo (invoke (get-attribute self 'car) 'getX) 5)))
       (else
        #f))))
   (debug-output (invoke (get-attribute self 'car) 'getX))
   #t
   ))

(define (get-x agent)
  (invoke (get-attribute agent 'car) 'getX))

(define (get-y agent)
  (invoke (get-attribute agent 'car) 'getY))

(defcueadapter ?crossing (self)
  (let-polling
   ((crossed
     (case (get-attribute self 'dir)
       ((horizonal)
        (if (= (get-attribute self 'dx) 1)
            (= (- l 1)
               (modulo (get-x self)
                       (+ l w)))
            (= 0
               (modulo (get-x self)
                       (+ l w)))))
       ((vertical)
        (if (= (get-attribute self 'dy) 1)
            (= (- l 1)
               (modulo (get-y self)
                       (+ l w)))
            (= 0
               (modulo (get-y self)
                       (+ l w))))))))
```

```
   #f))

'(defcueadapter ?carExists (self)
  (let-polling
   ((car-exists
     (case (get-attribute self 'dir)
       ((horizonal)
        (if (< (modulo (get-x self) (+ l w)) (- l 1))
            (invoke *map* 'isCar
                    (+ (get-x self) (get-attribute self 'dx))
                    (get-y self))
            #f))
       ((vertical)
        (if (< (modulo (get-y self) (+ l w)) (- l 1))
            (invoke *map* 'isCar
                    (get-x self)
                    (+ (get-y self) (get-attribute self 'dy)))
            #f)))))
   #f))

(defcueadapter ?carExists (self)
  (let-polling
   ((car-exists
     (case (get-attribute self 'dir)
       ((horizonal)
        (invoke *map* 'isCar
                (+ (get-x self) (get-attribute self 'dx))
                (get-y self)))
       ((vertical)
        (invoke *map* 'isCar
                (get-x self)
                (+ (get-y self) (get-attribute self 'dy)))))))
   #f))

(defcueadapter ?redLight (self)
  (let-polling
   ((red-light
     (case (get-attribute self 'dir)
       ((horizonal)
        (list-ref *h-light* (car *light-timer*)))
       ((vertical)
        (list-ref *v-light* (car *light-timer*))))))
   #f))

(defcueadapter ?crossingAt (self &key (x 1) (y 1))
  (let-polling
   ((t (let ((agent-x (get-x self))
             (agent-y (get-y self))
             (left (+ (* l x) (* w (- x 1))))
             (right (- (* (+ l w) x) 1))
             (top (+ (* l y) (* w (- y 1))))
             (bottom (- (* (+ l w) y) 1)))
         (and (>= agent-x left)
```

```
             (<= agent-x right)
             (>= agent-y top)
             (<= agent-y bottom)))))
   #f))

(defcueadapter ?turnPoint (self &key (dir "right"))
  (let-polling
   ((t (let ((agent-x (modulo (- (get-x self) l) (+ l w)))
             (agent-y (modulo (- (get-y self) l) (+ l w)))
             (agent-dir (get-attribute self 'dir)))
         (cond
          ((or (and (eq? agent-dir 'horizonal) (equal? dir "right"))
               (and (eq? agent-dir 'vertical) (equal? dir "left")))
           (= (+ agent-x agent-y) (- w 1)))
          (else
           (= agent-x agent-y))))))
   #f))

'(defcueadapter ?turnPoint (self &key (dir 'right))
  (let-polling
   ((t #t)) #f))

(defcueadapter ?turning (self &key (dir ""))
  (let-polling
   ((t (equal? dir (get-attribute self 'turning))))
   (set-attribute! self 'turning #f)
   #t))

; +----------+
; | FOR META |
; +----------+

(define-get-environment car-exists pos
  (invoke *map* 'isCar (car pos) (cdr pos)))

(define-set-environment sim-speed speed
  (set! *sim-speed* speed))

(define *sw* #f)
(define *sw-lock* '())

(define (sw-on)
  (synchronized *sw-lock*
                (set! *sw* #t)))

(define (sw-off)
  (synchronized *sw-lock*
                (set! *sw* #f)))

(define-observe-environment switch val
  (eq? val *sw*))

(define-set-environment switch val
```

```
  (if val
      (sw-on)
      (sw-off)))

(define (start-simulation)
  (set! *sim-start* #t))

(define (stop-simulation)
  (set! *sim-start* #f))

(define-set-environment timer val
  (cond
   ((number? val) (set! *time* val))))

(define-observe-environment timer val
  (>= *time* (/ val *sim-speed*)))

(define *input-sim-start* #f)
(define (sim-start)
  (set! *input-sim-start* #t))

(define-observe-environment sim-start val
  (if *input-sim-start*
      (begin (set! *input-sim-start* #f) #t)
      #f))

(define-set-environment light-timer val
  (debug-output (list 'light-timer val))
  (if (eq? val 'start)
      (start-light-timer)))

(define-observe-environment onCrossing args
  (let* ((agent (car args))
         (agent-x (get-x agent))
         (agent-y (get-y agent))
         (cx (cadr args))
         (cy (caddr args))
         (left (+ (* l cx) (* w (- cx 1)) (- 5)))
         (right (+ (* (+ l w) cx) 5))
         (top (+ (* l cy) (* w (- cy 1)) (- 5)))
         (bottom (+ (* (+ l w) cy) 5)))
    (and (<= agent-x right)
         (>= agent-x left)
         (<= agent-y bottom)
         (>= agent-y top))))

(define-observe-environment outOfCrossing args
  (let* ((agent (car args))
         (agent-x (get-x agent))
         (agent-y (get-y agent))
         (cx (cadr args))
         (cy (caddr args))
         (left (+ (* l cx) (* w (- cx 1)) (- 5)))
```

```
        (right (+ (* (+ l w) cx) 5))
        (top (+ (* l cy) (* w (- cy 1)) (- 5)))
        (bottom (+ (* (+ l w) cy) 5)))
    (or (> agent-x right)
        (< agent-x left)
        (> agent-y bottom)
        (< agent-y top))))
```

A.2.2 アバターの実装

```
(defavatarhandler (name args avatar)
  (let ((avatar-ui (make <TrafficAvatarUI> name avatar))
        (car (make <Car> 0 (+ l 1) 1 0 *map* 1)))
    (set-attribute! avatar 'car car)
    (set-attribute! avatar 'dir 'horizonal)
    (set-attribute! avatar 'dx 1)
    (set-attribute! avatar 'dy 0)
    (set-attribute! avatar 'ui avatar-ui)
    (set-attribute! avatar 'turning #f)
    '(do ()
        (#f)
      (invoke car 'move)
      (sleep *sim-speed*))))

(define (change-dir! avatar)
  (if (eq? (get-attribute avatar 'dir)
           'horizonal)
      (set-attribute! avatar 'dir 'vertical)
      (set-attribute! avatar 'dir 'horizonal)))

'(define (right avatar)
  (debug-output (list 'right avatar))
  (let ((dx (- (get-attribute avatar 'dy)))
        (dy (get-attribute avatar 'dx)))
    (set-attribute! avatar 'dx dx)
    (set-attribute! avatar 'dy dy)
    (change-dir! avatar)
    (invoke (get-attribute avatar 'car)
            'changeDirection dx dy)))

(define (right avatar)
  (set-attribute! avatar 'turning "right"))

'(define (left avatar)
  (let ((dx (get-attribute avatar 'dy))
        (dy (- (get-attribute avatar 'dx))))
    (set-attribute! avatar 'dx dx)
    (set-attribute! avatar 'dy dy)
    (change-dir! avatar)
    (invoke (get-attribute avatar 'car)
            'changeDirection dx dy)))

(define (left avatar)
  (set-attribute! avatar 'turning "left"))
```

```
(define (stop avatar)
  (invoke (get-attribute avatar 'car)
          'changeDirection 0 0))

(define (start avatar)
  (invoke (get-attribute avatar 'car)
          'changeDirection
          (get-attribute avatar 'dx)
          (get-attribute avatar 'dy)))

(define (avatar-input? avatar)
  (invoke (get-attribute avatar 'ui)
          'isAvatarInput))

(define-observe-environment avatar-input val
  (avatar-input? val))

(define-observe-environment avatar-not-input val
  (not (avatar-input? val)))
```