Reference Documentation

SIMON

AI Learning Framework

Seo Jaesuk, Park Jinsang,
Oh Hyeseong, Kim Chulpyo

Version 1.0

User Manual

June 20, 2014

SIMON Framework v1.0 Reference Documentation

Overview of SIMON Framework

1. Introduction

SIMON Framework는 게임 AI 모델링과 구현에 주로 사용될 목적으로 제작된 Open-source framework입니다. 본 프레임워크는 Seo Jaesuk, Park Jinsang, Oh Heysung, Kim Chulpyo 로 구성된 SIMON Project Team에 의해서 초기 개발되었습니다.

SIMON Framework는 당신의 프로젝트에서 Non Player Character(이하 NPC)의 행동에 대해서 예측되지 못한 행동과 속성값을 갖고 독자적인 라이프사이클을 가질 수 있도록 구현해줍니다. 또한 프레임워크 기능을 통해서 높은 퀄리티의 자유도를 얻을 수도 있을 것입니다.

SIMON은 AI 적용 오브젝트에 대한 학습 루틴을 쉽게 구현할 수 있도록 설계되었습니다. 프레임워크는 자체 루틴과 개발자가 정의한 환경에 대한 연결 작업을 수행하는 핵심 루틴을 갖고 있습니다. 프레임워크에 대해 이해한 후에는 쉽게 게임에 AI를 적용할 수 있을 것입니다.

1.1 Background

많은 종류의 AI에 대한 모델은 다수의 'if' 문과 'else if' 문으로 구성된 Rule-based Model을 따르고 있습니다. 긍정적인 측면에서 Rule based Model 은 deterministic 하며, 신뢰할 수 있는 예측 가능한 결과를 도출해냅니다. 그러나 이 모델은 높은 퀄리티의 자유도를 만들어내기 쉽지 않고 많은 경우에 획일적인 패턴을 만들어냅니다. 이 모델을 보다 개량한 FSM 모델 등도 보다 나아진 환경을 제공하기는 하지만, 높은 자유도의 구현에 있어서 제한된 상태 범주를 갖습니다. 이에 따라 많은 AI 모델은 여러 가지 모델의 혼용으로 높은 자유도와 합리성을 추구합니다.

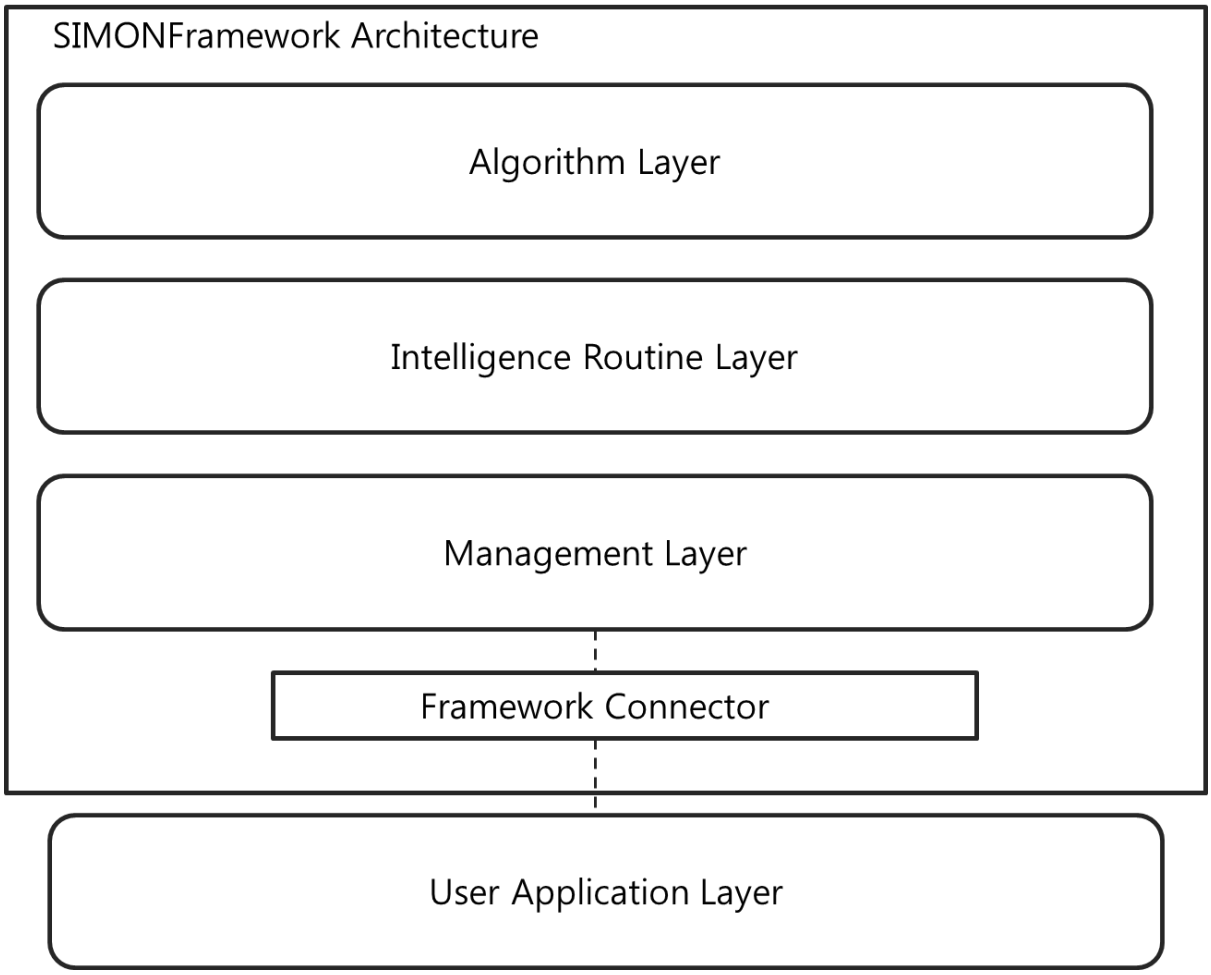
1.2 Purpose

SIMON Framework는 Non deterministic Algorithm 등을 이용해서 SIMON routine 을 통해 높은 자유도를 갖는 환경을 만들어내는데 목적이 있습니다.

SIMON Framework는 개발자가 AI 모델만 이해하고 잘 구성한다면 Framework에서 대부분의 루틴과정을 수행하도록 설계되었습니다.

1.3 Framework Architecture

SIMON Framework는 Manager의 동작 하에 여러 개의 모듈이 독립적인 Layer를 갖고 각자의 Routine을 실행합니다. 각 계층들은 그림에 묘사되어 있습니다.



Framework Connector

Framework Connector 모듈은 구현된 SIMON Framework 를 해당 동작 Application 또는 툴과 연결시키는데 사용됩니다. Framework는 현재 버전에서 C# 을 기반으로 구현되어 있으며 Unity 등 여러 툴에서 바로 사용할 수 있도록 호환성을 제공하는 Connector를 제공합니다.

Connector 모듈은 porting 대상 Application에 맞게 제공되어야 하며 Framework 자체와 상호 연결성이 존재합니다. 모듈 내부는 Manager 루틴을 Singleton 방식으로 사용하게끔 구성되어 있습니다.

사용하는 Application 의 환경에 따라서 구현은 모두 다르게 되어있으며 Manager 루틴을 사용 하는 구현 인터페이스는 동일합니다.

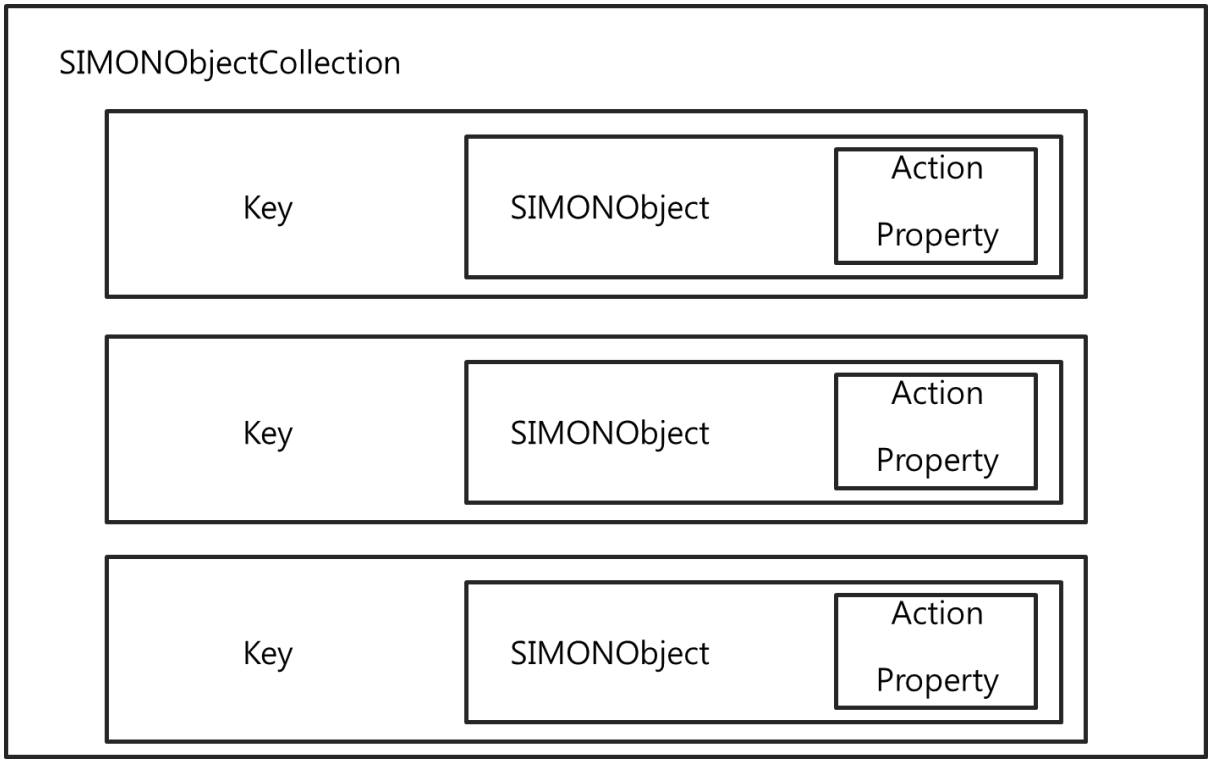
Management Layer

Management Layer는 SIMON Framework가 제공하는 모든 모듈들에 대한 기능을 총괄하는 루틴을 구현합니다. User Application에 적용되는 구현은 Connector에 의해 이루어지지만, 기능은 Management Layer의 모듈을 이용합니다. Management Layer는 다음과 같은 3가지의 주된 모듈로 구성됩니다.

- SIMON Object 관리 모듈
- 학습 환경 구축 모듈
- Intelligence Routine 수행 모듈

SIMON Object 관리 모듈

SIMON Framework에서 AI를 적용시키는 모델 객체에 대한 정의는 SIMONObject 클래스를 통해서 이루어집니다. 해당 SIMONObject 객체들은 Management Layer에서 다음 그림과 같이 Dictionary 형태로 Collection을 통해서 관리됩니다.



SIMONObjectCollection을 구현하는데 사용되는 Key값은 Default로 SIMONObject의 고유 식별번호(ObjectID)로 이루어져 있습니다. SIMONObject를 구성하는 주된 요소는 Action과 Property가 있습니다.

- Action

SIMONObject의 행동을 의미하는 관념으로써 Action은 SIMONAction 이라는 클래스 형태로 명시됩니다. SIMONAction 클래스는 다음과 같은 멤버들을 갖고 있습니다.

```
public class SIMONAction
{
    public String ActionName { get; set; }
    public String ActionFunctionName { get; set; }
    public String ExecutionFunctionName { get; set; }
    public String FitnessFunctionName { get; set; }
    public List<SIMONGene> ActionDNA { get; set; }

    .
    .
    .
}
```

클래스는 Action의 이름과 Action에 적용되는 ActionFunction, ExecutionFunction, FitnessFunction의 이름을 갖고 있으며 Action에 대한 가중치 DNA값을 List로 갖고 있습니다.

```
public class SIMONGene
{
    [XmlElement("ElementName")]
    public String ElementName { get; set; }
    [XmlElement("ElementWeight")]
    public Double ElementWeight { get; set; }

    .
    .
    .

}
```

SIMONGene은 위와 같이 가중치의 이름과, 가중치 값으로 구성된 클래스로 SIMONAction의 DNA List를 구성합니다.

- Property

SIMONObject가 갖는 User가 정의하는 속성값을 명시하는 관념으로 SIMONProperty 라는 클래스 형태로 제공됩니다. SIMONProperty 클래스는 다음과 같은 멤버들을 갖고 있습니다.

```
public class SIMONProperty
{
    [XmlElement("PropertyName")]
    public String PropertyName { get; set; }
    [XmlElement("PropertyValue")]
    public Double PropertyValue { get; set; }
    [XmlElement("Inherit")]
    public Boolean Inherit { get; set; }

.
.
.

}
```

클래스는 Property의 이름과 Property에 적용되는 PropertyValue, 그리고 Property의 유전성 여부를 결정하는 Inherit Flag로 구성되어 있습니다.

학습환경 구축 모듈

SIMONFramework는 학습 루틴을 수행하기 위해서 SIMONObject를 비롯한 몇가지 도구를 제공합니다. SIMONFramework에서 특징적으로 제공하는 도구에는 다음과 같은 것들이 있습니다.

- SimonFunction 리스트

SIMON Framework는 Dynamic Invoke 방식을 통해서 User Application과 Framework 간의 Linking 을 수행합니다. 이를 위해서 Management Routine에서는 Simon Function의 Delegate들을 Dictionary 형태로 구축해서 저장합니다. Default 키 값으로는 Function의 이름이 저장되고, Value 값으로 SIMONFunctionInterface 를 구현한 함수의 Delegate가 저장됩니다. SIMONFunctionInterface의 Delegate는 다음과 같은 형태를 지닙니다.

```
public delegate Object SIMONFunction(SIMONObject One, SIMONObject[] TheOthers);
```

함수는 자기자신에 대한 SIMONObject Parameter와 그룹내의 다른

SIMONObject 객체 들에 대한 Parameter를 전달받아 Object를 반환합니다.

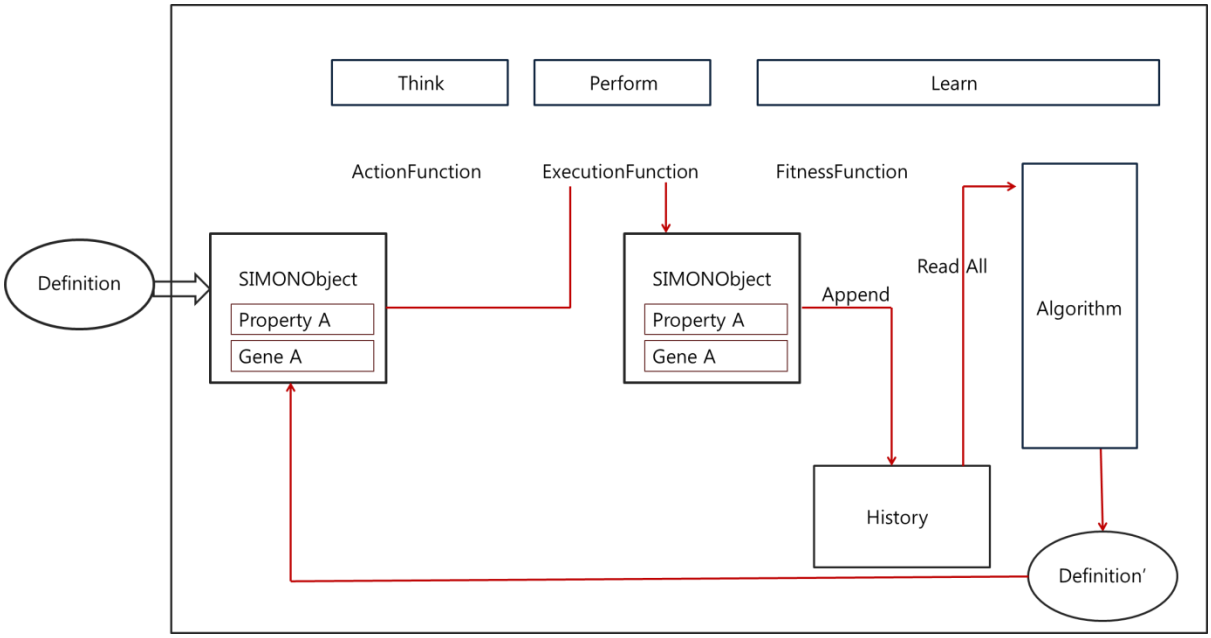
- Simon Definition 파일

SIMON Framework는 SIMONObject의 구성을 Local에 저장할 수 있는 구현 방법으로 XML Serialize / Deserialize 를 사용합니다. Definition 의 구현은 SIMONObject에 따라 Hierarchical Tree 형태로 구성되며, Default Encoding Set으로 UTF-8 을 사용해서 구현되어 있습니다.

Framework 는 Serialize와 Deserialize에 대한 구현을 Utility 클래스를 통해서 제공하며, SIMONUtility 클래스는 Management Routine 전역에서 Singleton으로 사용하게끔 구현되어 있습니다.

Intelligence Routine 수행 모듈

Management Routine에서는 직접적인 학습 알고리즘의 수행을 구현하지 않습니다. 하지만 Framework의 Main Sequence를 만들고, Intelligence Routine을 수행시키는 모듈을 구현하여 정의된 Workspace 환경에 대해서 Framework의 Procedure를 수행하게끔 구현합니다. 다음은 SIMON Framework 가 수행하는 Main Sequence Procedure를 표현한다.

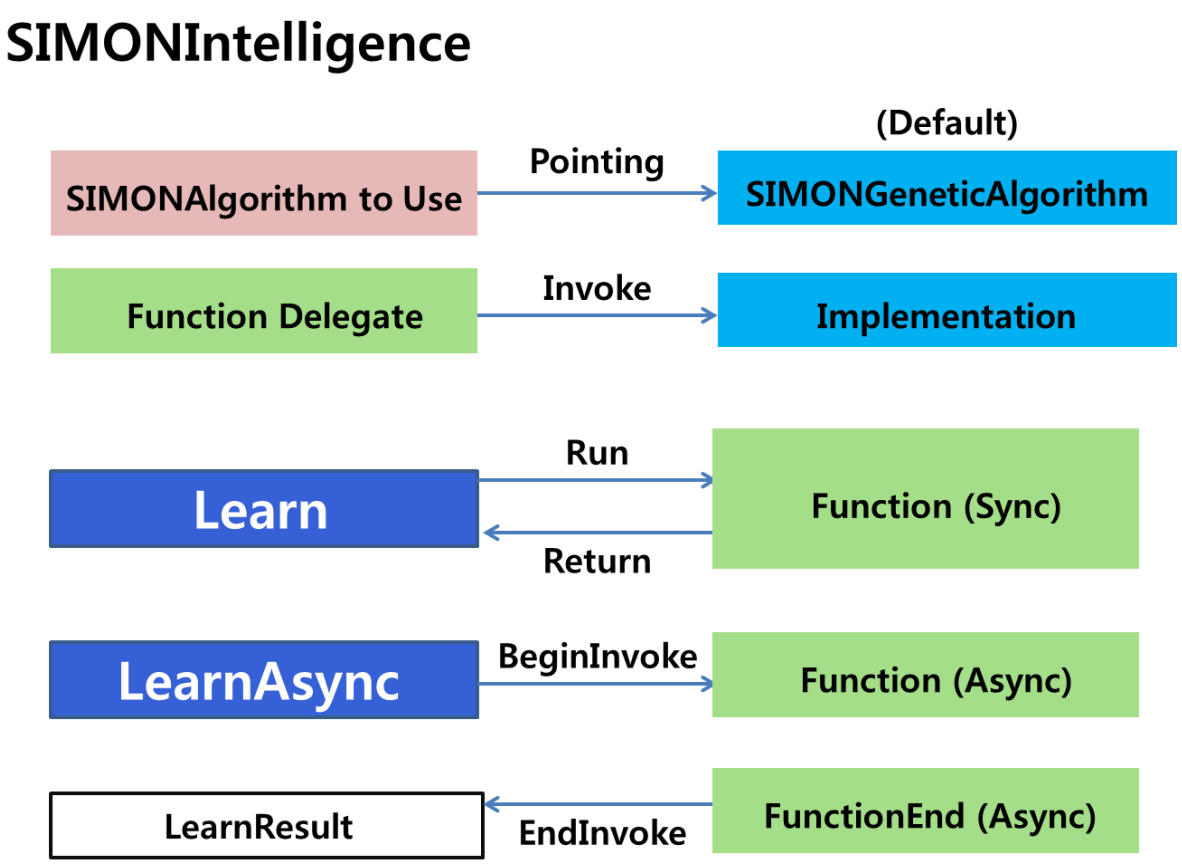


Management Routine Layer에서 수행되는 모듈의 Architecture와 Sequence Flow는 위와 같으며 모듈의 세부 동작에 대해서 내부 Layer에 들어가서 수행하게 된다.

Intelligence Routine Layer

Intelligence Routine Layer는 Delegate를 통해서 SIMON Framework 가장 깊숙한 계층에 위치하는 Algorithm Layer의 구현을 제어하는 역할을 수행합니다. Delegate 들의 관리와 제어에 있어서 동기방식 및 비동기 방식을 모두 지원하며 비동기 작업에 대한 Callback Method를 지원합니다.

사용하게 되는 Algorithm에 대한 열거형 변수 지정을 통해서 사용할 알고리즘을 한정하고 AI가 적용될 Algorithm 클래스에 대한 Property 값들을 설정할 수 있게 끔 Bridge 역할을 해주는 Layer를 구현합니다.



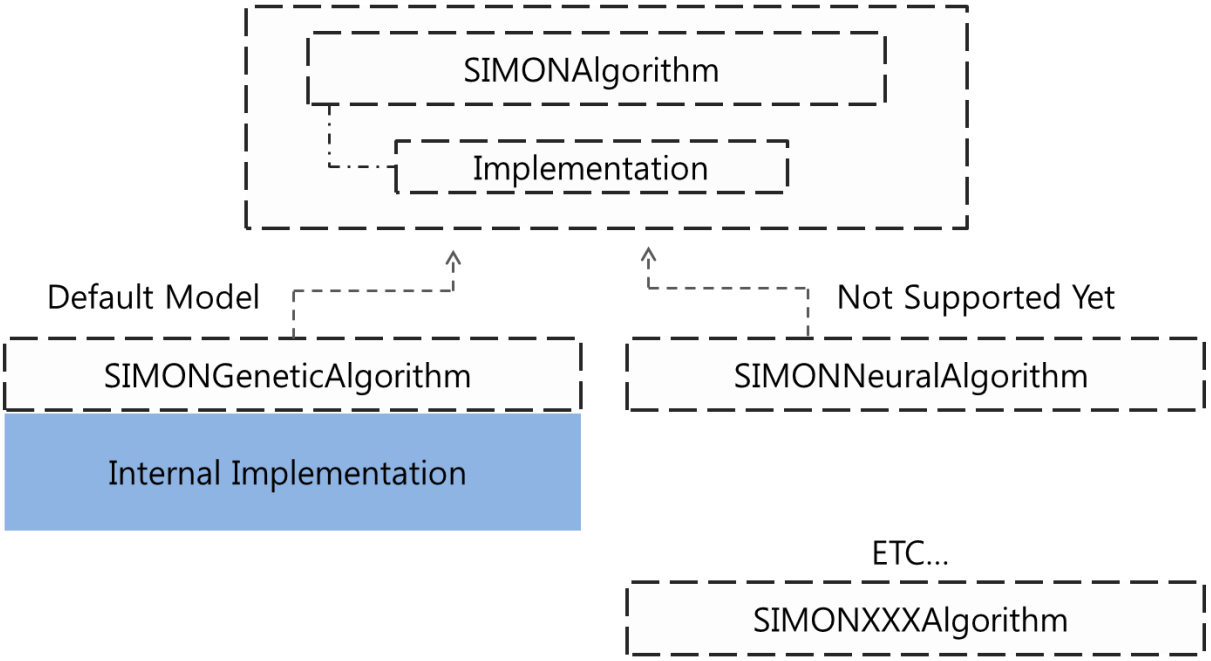
Intelligence Routine Layer는 Intelligence 학습 모듈만으로 구성되며 위의 그림과 같은 기능을 제공합니다.

Algorithm Layer

Algorithm Layer는 SIMON Framework 가장 깊숙한 계층에서 수행되는 학습 알고리즘을 구현하고 실행합니다.

현재 Version에서 기본적으로 사용되는 AI Algorithm의 클래스는 Genetic Algorithm을 사용하며 Algorithm Class에 대한 구현이 Intelligence Routine Layer에서 Delegate로 Pointing되어 있고, Invoke 또는 BeginInvoke를 통한 동기 / 비동기적 호출 방식으로 제어됩니다.

SIMONAlgorithm Interface



SIMONAlgorithm Interface를 구현한 SIMONGeneticAlgorithm 클래스가 Layer에서 Non-deterministic 방식으로 학습에 사용됩니다. 차후 Version에서 Neural Network를 통한 Algorithm 의 구현이 제공됩니다. SIMONAlgorithm Interface를 구현한 어떤 형태의 클래스라도 Intelligence Routine에 의해 실행될 수 있습니다.

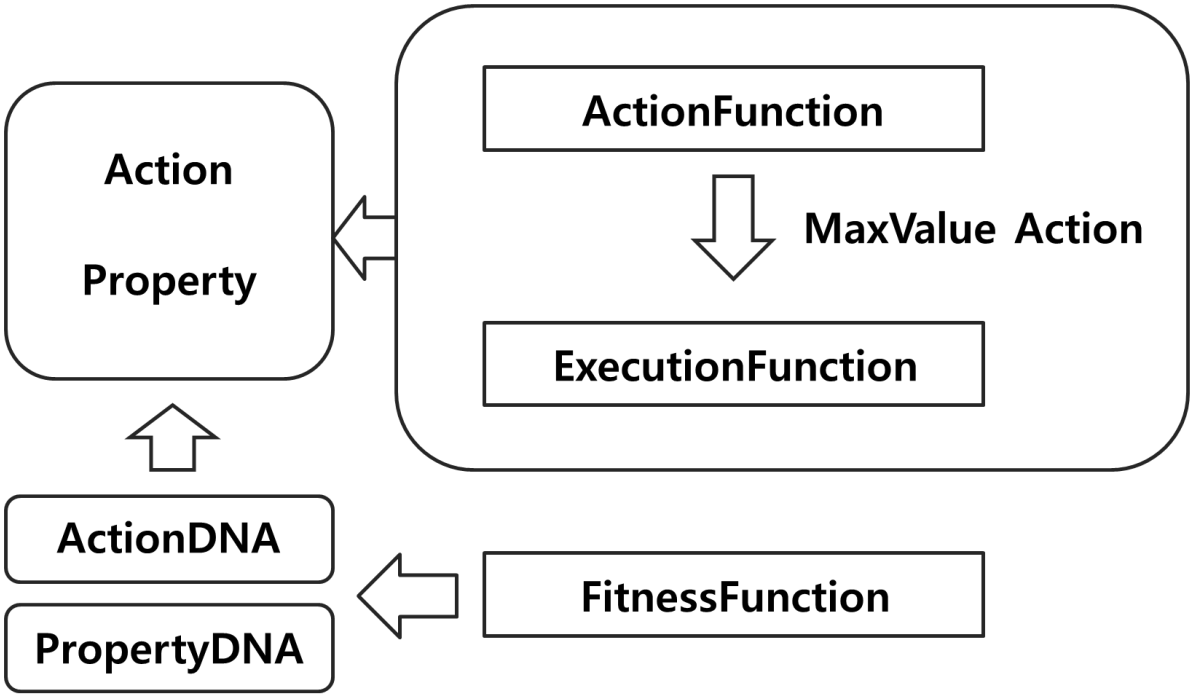
SIMONGeneticAlgorithm 의 Property로 지정된 Option 값으로는 Mutation 확률과 Mutation 적용 비율이 있으며 Genetic Algorithm 의 구축에서 Mutation 연산에 직접 사용되게 됩니다.

2. Core technology of SIMON Framework

2.1 SIMON Routine

SIMON Framework 는 해당 SIMONObject Collection 그룹에 대한 판단 및 학습 루틴을 메인 Sequence 루틴으로 구현합니다. Collection 내의 각 element 요소들의 행동에 대해서 평가할 함수로는 ActionFunction을 이용하며, 평가에 따른 판단은 ActionFunction의 Output의 최댓값으로 결정하게 됩니다. 결정한 판단에 대한 행동은 ExecutionFunction으로 이루어지며, 이런 식의 판단 – 행동이 메인 Sequence를 구성합니다. 학습루틴은 별도로 작용하며 학습에 관여하는 함수로 FitnessFunction을 사용하게 됩니다.

판단 – 행동 – 학습의 과정이 SIMON Framework가 제공하는 Main Routine이며 다음과 같은 Flow를 갖습니다.



Routine에서 사용되는 각 함수들은 SIMON Function의 Delegate를 만족해야 하

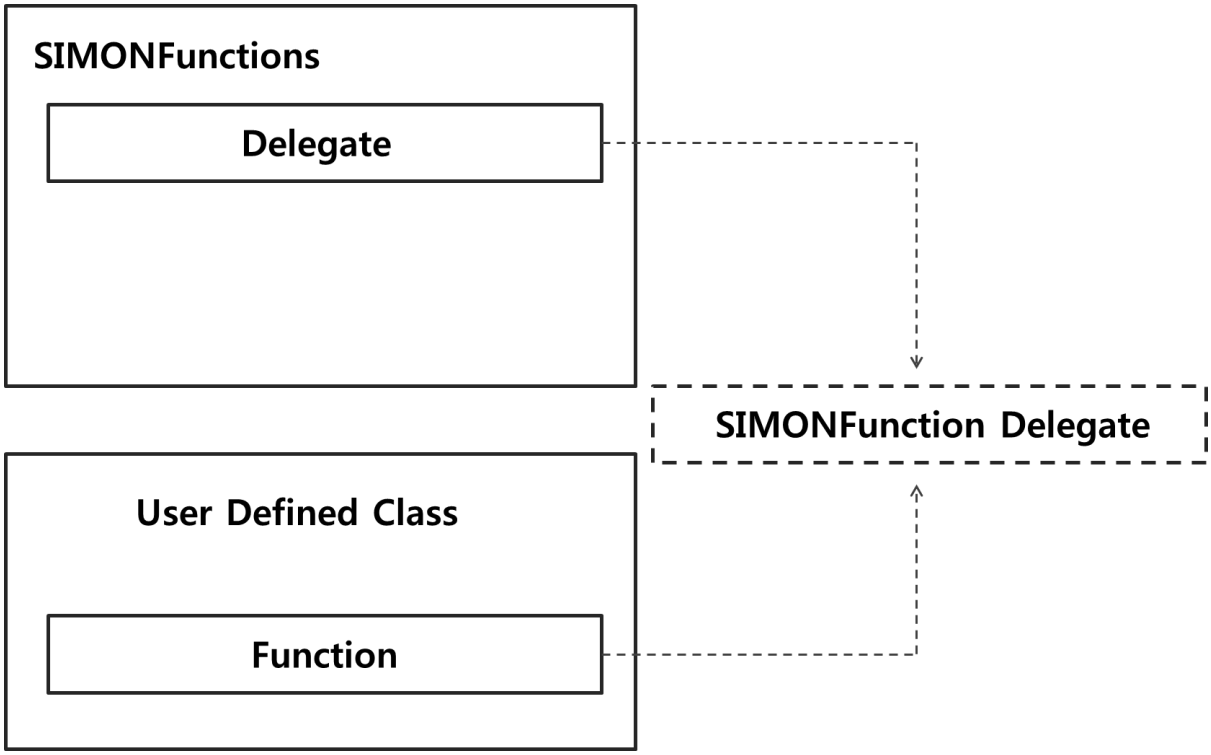
며 SIMONManager에 메서드를 등록하는 과정을 통해 SIMONManager가 Routine을 돌면서 조회 가능해야 합니다. Default로 참조되는 클래스는 SIMONFunctions라는 이름으로 정의되지만, 직접 Delegate를 명시함으로써 구현에 대한 Customizing을 할 수 있습니다.

2.1 Dynamic Linking and Dependency Injection of SIMON Framework

User가 원하는 모델을 통한 AI 구축 서비스를 제공함에 있어서 SIMON Framework는 Delegate의 개념과 Reflection 을 통한 코드의 동적 참조를 이용합니다.

Delegate는 특정 메서드 자체를 캡슐화하는 포인터 함수와 같은 개념으로 Pointing된 해당 메서드를 대신 수행해주는 역할을 합니다. Reflection은 실행 중에 클래스 및 객체의 타입에 대한 조사 및 접근과 참조 기능을 제공하는 역할을 합니다.

정해진 SIMON Running Routine 내에서 User level로부터 올려받은 함수들에 대한 Delegate를 지정하는 측면에서 프로그램의 제어는 User Level에서 Framework Level로 상향되서 수행되게 됩니다. 구현 객체에 대한 정보를 Framework의 매니저에서 관리하고, 사용자가 정의한 함수들의 Routine 마저도 매니저에서도 실행을 관리하고 제어하기 때문에 각각의 컴포넌트들은 독립적으로 상호작용할 수 있게 됩니다. 또한 Interface를 만족하는 요소들에 대해서 Setter 함수(Insert-)를 제공함으로써 User가 정의한 기능들이 자연스럽게 정해진 제어 루틴에 포함될 수 있게 제공합니다.



이러한 측면에서 SIMON Framework 는 완전한 IoC 컨테이너라고 할 수는 없지만 Dependency Injection의 개념을 갖고 있다고 볼 수 있습니다.

3. How to use SIMON Framework

3.1 User Scenario

SIMON 은 개발자가 정의한 AI 모델에 대한 학습을 지원해줄 수 있는 .Net 기반의 프레임워크입니다. 현재 버전에서 .Net framework 3.5 이상 버전의 프로젝트에서 사용 가능하게 지원합니다.

Install Framework

SIMON Framework의 최신 버전은 http://210.118.74.81/SIMON/ 에서 다운로드 할 수 있습니다.

- .Net 프로젝트에서 사용하기 위해서는 SIMON Framework 를 다운로드 합니다.
- Unity 프로젝트에서 사용하기 위해서는 SIMON Framework Export Package 를 다운로드 합니다.

Link SIMON

SIMON Framework를 .Net 기반 프로젝트에서 사용하기 위해서는 SIMON Framework를 프로젝트에 추가시킨 후, namespace를 using 지시문을 통해 포함시켜주기만 하면 사용이 가능합니다.

```
1
2  using System;
3  using System.IO;
4  using SIMONFramework;
5
```

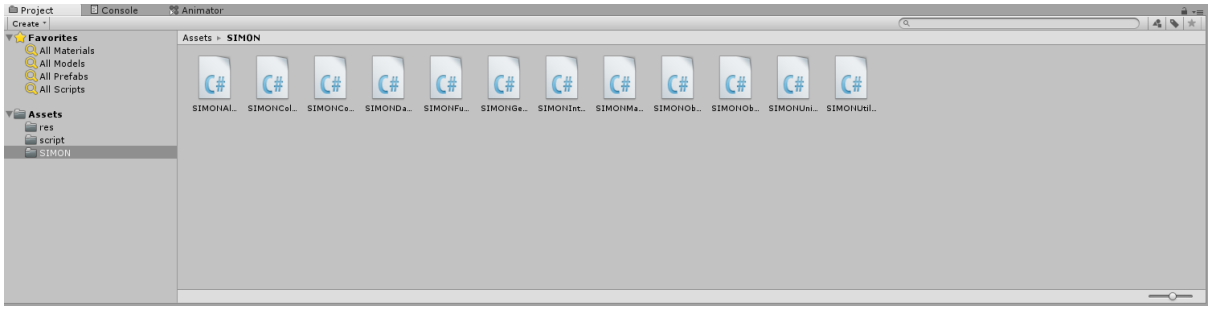
SIMON Framework의 요소들은 전부 앞에 prefix로 SIMON을 명기하고 있으며, 이를 통해 편하게 사용할 수 있습니다.

SIMON은 다른 외부 개발 툴에서도 얼마든지 사용이 가능하며, 다음은 설치 및

사용 방법에 대해 설명합니다.

- Unity에서 SIMON Framework의 사용

Unity에서 SIMON을 사용하기 위해 Export Package를 다운로드 받아서 Unity에서 Import 시키면 다음과 같이 SIMON 디렉터리 안에 프레임워크가 설치되게 됩니다.



프레임워크가 설치되면 Unity 내 Script에서 SIMON Framework를 namespace를 통해서 사용할 수 있습니다. 하지만 본 프레임워크를 사용하는데 있어서 다음 전역 클래스를 이용한 사용을 권장합니다.

```
1
2 using UnityEngine;
3 using System.Collections;
4 using System.Collections.Generic;
5 using SIMONFramework;
6
7 public static class SIMON{
8     public static SIMONUnity GlobalSIMON = new SIMONUnity();
9 }
10
```

Framework는 Unity에서 사용되기 위해서 SIMONUnity.cs 라는 Connector 소스를 제공하며, Connector를 통해서 SIMON 의 모든 기능을 Unity 환경에 맞게 사용할 수 있습니다. 또한 그림과 같이 Singleton 으로 구현되어있기 때문에 구조의 무결성을 보장할 수 있습니다.

Create SIMONObject

SIMONObject는 SIMON Framework를 이용하기 위한 가장 작은 단위입니다.

SIMONObject는 Property와 Action, 그리고 각각의 DNA 들로 구성되어 있으며, Framework 가 사용 가능한 어떤 위치에서도 선언할 수 있습니다. SIMONObject 는 다음과 같이 정의합니다.

```
public Human()
{
    SIMONObject smartOne = new SIMONObject();
    smartOne.ObjectID = "Tom";
    smartOne.Properties.Add(new SIMONProperty("health", 100, true));
    smartOne.Properties.Add(new SIMONProperty("joy", 10, false));
    smartOne.Properties.Add(new SIMONProperty("wealth", 50, true));
    smartOne.Actions.Add(new SIMONAction("Sleep", "SleepAction", "SleepExecution", "SleepFitness", null));
    smartOne.Actions.Add(new SIMONAction("Play", "PlayAction", "PlayExecution", "PlayFitness", null));
    smartOne.Actions.Add(new SIMONAction("Work", "WorkAction", "WorkExecution", "WorkFitness", null));

}
```

그림에서 smartOne이라는 SIMONObject는 "Tom" 이라는 이름을 갖고 있으며, "health", "joy" 그리고 "wealth" 라는 Property 를 갖고 있습니다. 또한 이 인물의 행동 양식으로는 "Sleep", "Play" 그리고 "Work" 가 정의되어 있습니다.

다음은 이 인물을 SIMON 을 통해서 동작할 수 있도록 함수들을 정의하고 연결 시키겠습니다. 연결을 위해 초기화 부분에 다음과 같은 코드를 추가시킵니다.

```
smartOne.ObjectFitnessFunctionName = "TomFitness";
smartOne.UpdatePropertyDNA();

SimonManager.AddMethod("TomFitness", new SIMONFunction(TomFitness));
SimonManager.AddMethod("SleepAction", new SIMONFunction(SleepAction));
SimonManager.AddMethod("SleepExecution", new SIMONFunction(SleepExecution));
SimonManager.AddMethod("SleepFitness", new SIMONFunction(SleepFitness));
SimonManager.AddMethod("PlayAction", new SIMONFunction(PlayAction));
SimonManager.AddMethod("PlayExecution", new SIMONFunction(PlayExecution));
SimonManager.AddMethod("PlayFitness", new SIMONFunction(PlayFitness));
SimonManager.AddMethod("WorkAction", new SIMONFunction(WorkAction));
SimonManager.AddMethod("WorkExecution", new SIMONFunction(WorkExecution));
SimonManager.AddMethod("WorkFitness", new SIMONFunction(WorkFitness));
SimonManager.RegisterSIMONObject(smartOne);
```

추가한 부분은 SIMONObject의 동작을 실제 함수와 Delegate를 이용한 동적 매칭을 시켜주는 부분입니다. 그리고 그 사실을 SIMONManager에게 통지하고 Object를 등록하는 과정입니다.

```
public object TomFitness(SIMONObject one, SIMONObject[] theOthers)
{
    return one.GetPropertyElement("health") + one.GetPropertyElement("joy") + one.GetPropertyElement("wealth");
}
public object SleepAction(SIMONObject one, SIMONObject[] theOthers)
{
    return -one.GetPropertyElement("health");
}
public object SleepExecution(SIMONObject one, SIMONObject[] theOthers)
{
    one.SetPropertyElement("health", 150);
    return (double)1;
}
public object SleepFitness(SIMONObject one, SIMONObject[] theOthers)
{
    return one.GetPropertyElement("health") * one.GetPropertyElement("joy");
}
public object PlayAction(SIMONObject one, SIMONObject[] theOthers)
{
    return -one.GetPropertyElement("joy") + one.GetPropertyElement("wealth");
}
public object PlayExecution(SIMONObject one, SIMONObject[] theOthers)
{
    one.SetPropertyElement("joy", 150);
    return (double)1;
}
public object PlayFitness(SIMONObject one, SIMONObject[] theOthers)
{
    return one.GetPropertyElement("joy") - one.GetPropertyElement("wealth");
}
public object WorkAction(SIMONObject one, SIMONObject[] theOthers)
{
    return one.GetPropertyElement("wealth") + one.GetPropertyElement("health");
}
public object WorkExecution(SIMONObject one, SIMONObject[] theOthers)
{
    one.SetPropertyElement("wealth", 150);
    return (double)1;
}
public object WorkFitness(SIMONObject one, SIMONObject[] theOthers)
{
    return one.GetPropertyElement("health") + one.GetPropertyElement("joy");
}
```

SIMONFunction으로 매핑한 함수의 구현에 개발자가 정의한 AI 모델링에 대한 공식을 구현하면 SIMONObject가 Framework에서 동작할 준비가 끝납니다.

Call SIMONManager

SIMON을 통해서 학습시키기 위해서는 학습 대상을 Group을 통해 범위를 정해줘야 합니다. 단일 객체도 Group이 될 수 있으며 학습과 판단의 루틴은 Group 단위로 진행됩니다. Group을 만드는 방법은 다음과 같이 SIMONCollection 클래스를 이용합니다.

```
SIMONCollection HumanGroup = SimonManager.CreateSIMONGroup();
HumanGroup.Add(smartOne.ObjectID, smartOne);
```

Group 은 잘 관리하기만 한다면 언제든지 동적으로 추가 및 삭제가 가능하며, Key값으로 구별이 되므로, Object ID를 Unique한 값으로 하여 관리하는 것을 권장합니다. Group에 대한 학습은 SIMONManager를 이용합니다.

```
while (true)
{
    SimonManager.LearnRoutine(HumanGroup);
    System.Threading.Thread.Sleep(1000);
}
```

위와 같이 반복문안에서 루틴을 수행할 경우, 예제의 그림에서는 1초마다 학습이 진행되게 됩니다.

LearnSimulate() 함수를 이용하면 학습이 비동기적으로 수행되고, 학습률에 대한 조절도 할 수 있게 됩니다. 사용하는 측에서의 비동기 스케줄링이 잘 될 경우에 내부 루틴이 보다 나은 성능을 제공할 수 있습니다.

Configure Learning

SIMON 에서는 개발자가 학습의 방식을 동적으로 설정할 수 있도록 합니다. 즉, 개발자가 프레임워크 내에서 제공하는 알고리즘에 대해서 정해져 있는 계수 값을 변경함으로써 Framework가 제공하는 선 안에서의 Tuning 작업이 가능하게 됩니다.

```
SimonManager.ConfigureGeneticLearn(new SIMONGeneticAlgorithm.GeneOption(100, 100));
```

가령 위와 같은 함수를 사용하면, SIMON Framework에서 제공하는 유전 알고리즘의 돌연변이 확률과 돌연변이 발생시 변이 정도를 External Layer에서 Dependency에 안전하게 변경할 수 있습니다.

Use Definition Tool

Definition Making Tool을 이용하면 SIMONObject를 Tool을 이용해서 편하게 만들고, 배포 및 재사용할 수 있습니다.

Definition Making Tool에서는 Property와 Action의 추가, 삭제 및 수정을 지원하고 Project 단위의 관리를 통해서 여러 개의 SIMONObject를 효과적으로 관리하고 재사용할 수 있게 만들어줍니다.

Definition Making Tool의 최신 버전은 http://210.118.74.81/SIMON/ 에서 다운로드 하실 수 있습니다.

3.2 Design AI Model

Design ActionFunction

ActionFunction 은 해당 Action이 어떤 경우에 어느 정도의 Weight를 갖고 실행되어야 하는 지를 나타냅니다. SIMON 에서는 해당 함수의 구현을 Input에 대하여 조건문의 분기에 따라서 정해진 Output Value를 만들어내는 함수가 아닌 연속적인 함수식 또는 점화식의 형태로 구현하는 것을 권장합니다.

함수식은 주어지는 SIMONObject 주체와 SIMONObject 객체 배열을 이용하며, 구현에 따라서 자유롭게 프로그램의 변수들을 이용합니다.

```
public object SleepAction(SIMONObject one, SIMONObject[] theOthers)
{
    return -one.GetPropertyElement("health");
}
```

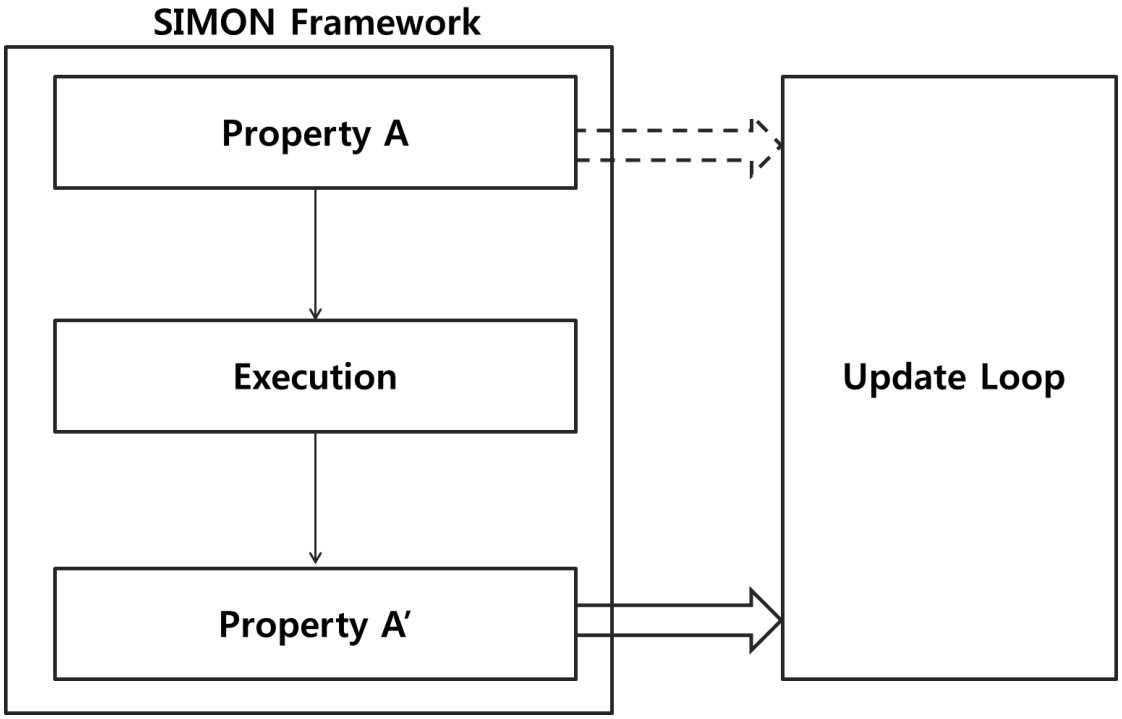
SleepAction 함수는 반환값으로 health의 Inverse 값을 return 합니다. 이 뜻은 SIMONObject 주체의 "health" Property 값이 낮을수록 함수가 큰 값을 반환한다는 것을 의미합니다. 따라서, 이는 SIMON Framework에서 다음과 같이 해석됩니다.

"health 가 낮을수록 SleepAction 이 실행될 확률이 높아진다."

이와 같이 Property 값들을 이용해서 연속적인 함수를 구현하면 다차원적인 상황까지도 Framework 에서 모두 판단하도록 처리할 수 있습니다.

Design ExecutionFunction

ExecutionFunction 이란 해당 Action이 수행되었을 때, 변하는 Property 환경에 대한 내용을 포함합니다. SIMON 은 External developing tool 측의 권한을 침해하지 않고 오직 SIMONObject 내의 Property 값에만 영향을 주는 방식으로 구현하도록 합니다.



그림에서 보이는 형태는 주로 Game AI를 SIMON Framework를 이용하여 구현할 때 보이게 됩니다. Update Loop에서는 SIMONObject의 Property 값에 맞게 Update / Draw 연산을 수행해주며, ExecutionFunction에서는 해당 Developing tool과는 완전히 독립적인 작업을 하게 됩니다.

Design FitnessFunction

FitnessFunction은 ActionFunction과 ExecutionFunction이 '판단' 에 작용하는 것과 달리 '학습' 에 영향을 주는 함수입니다. 학습에 패턴을 형성하고 일정한 방향성으로 수렴하게끔 하는 Fitness Value를 정의하는 함수입니다. 가령 Tom이 Sleep하는 Action에 대해 다음과 같이 정의를 할 수 있습니다.

```
public object SleepFitness(SIMONObject one, SIMONObject[] theOthers)
{
    return one.GetPropertyElement("health") * one.GetPropertyElement("joy");
}
```

위 함수의 의미는 Sleep 이라는 Action에 대한 행동의 가중치가 health와 joy의 곱 연산만큼의 영향을 갖는 다는 의미입니다. 즉, DNA가 어떻게 되던 Sleep이라는 행동에 높은 가중치를 주는 유전자는 health와 joy의 영향에 따라 학습된다는 것을 알 수 있습니다.

Design PropertyFitnessFunction

위에 언급한 3가지의 함수가 Action에 대한 Modeling Function들이었다면, PropertyFitnessFunction은 Property 자체에 대한 FitnessFunction입니다. 프로그램의 AI 모델이 객체가 갖는 행위가 아닌 속성값에 의해서만 단순히 결정될 경우에 PropertyFitnessFunction을 구현한다면 간편하게 원하는 기능을 하게 만들 수 있

습니다. 보다 단순한 AI 모델에 대해서는 PropertyFitnessFunction 과 Property DNA의 사용을 권장합니다.

3.3 Extended Implements

Use Dynamic Invoke

SIMON은 Runtime에 함수를 매핑하여 Invoke 시키는 방식으로 사용자 측에서 대부분의 동작을 구현합니다. 즉, 동적으로 실행되는 환경 자체가 기본적으로 갖추어져 있고, 이는 개발 시 목적에 맞게 여러 형태로 응용될 수 있습니다.

```
public class SIMONUserFunction : SIMONFunctionInterface
{
    public string[] GetFunctionList()
    {
        string[] arr = { "Move" };
        return arr;
    }
    public int GetFunctionCount()
    {
        return 1;
    }
    public object Move(SIMONObject one, SIMONObject[] theOthers)
    {
        Console.WriteLine("Move Calls");
        return null;
    }
}
```

주어진 그림은 SIMONFramework에서 제공하는 FunctionInterface를 구현한 SIMONUserFunction 클래스입니다. 이렇듯 함수들을 하나의 Class에 전부 정의하여 코드 관리를 용이하게 할 수 있고, 코드 파일 별로 독립적인 함수를 구현하거나, Interface를 새로 정의한 뒤, 구현한 클래스를 Container로 사용하는 방식으로도 응용할 수 있습니다.

Use Definition Factory

Definition 파일들이 프로젝트 단위로 존재하고, 그 파일들에 대한 Workspace를 통째로 만들고 싶을 때는 Definition 파일을 통해서 SIMONManager에 모델링된 Factory 를 이용할 수 있습니다. SIMONManager 내에 존재하는 Factory에서는 Workspace의 Definition 각각에 대한 정의를 이용해서 복사된 객체들을 생산(Produce)합니다.

```
SIMONObject Jack = SimonManager.CopyDefinitionObject("Tom");
```

위와 같은 한 줄의 문장 만으로 SIMONObject Jack은 SIMONManager에 등록된 Tom에 대한 정의를 모두 복사해서 똑 같은 객체로 될 수 있습니다. 위와 같은 기능의 제공을 통해 AI 모델에 대한 생산 및 관리, 그리고 재배포를 용이하게 할 수 있습니다.

3.4 Constraints

Fitness Value Boundary

SIMONFramework에서는 Fitness Value에 대한 상한 / 하한을 정의합니다. 이는 알고리즘적 구현과, 그에 따라 External developing tool 등에서 발생할 수 있는 Memory 예외에 대한 문제 때문으로, 각각 +9999.9999와 -9999.9999로 정의하고 있습니다. 현재 버전에서는 double 형 자료의 특징을 이용해서 함수의 비교값을 잘 설정하기를 권장하고 있습니다.

Async Result Handling

SIMONFramework에서는 비동기적 작업에 대한 실행과 완료에 대한 처리를 모두 제공하지만, 비동기 객체에 대한 작업이 필요할 때 그 처리는 온전히 개발자의 몫으로 넘깁니다. 따라서 개발자는 비동기적 작업이 필요할 경우 해당 작업에 대한 결과를 따로 관리하기를 권장합니다.