Design and Development of Intentional Systems with PRACTIONIST Studio

Angelo Marguglio, Giuseppe Cammarata, Susanna Bonura, Giuseppe Francaviglia, Michele Puccio and Vito Morreale.

R&D Lab
ENGINEERING Ingegneria Informatica S.p.A.

WOA 2008
dagli Oggetti agli Agenti
Evoluzione dell'agent development: metodologie, tool, piattaforme e linguaggi

Palermo 17-18 Novembre 2008
Outline

- Motivations
- PRACTITIONIST Suite
- PRACTITIONIST Agent Modelling Language (PAML)
- PRACTITIONIST Studio (PS)
Motivations

- Based upon the BDI model
- Stress aspects such as mental states like beliefs, desires and intentions
- **Lack of industry ready tools** for design and development of MAS
  - Strong tie-up with specific methodologies
  - Cover only a subset of development phases
  - Simple prototypes
  - Very limited assistance
PRACTIONIST Suite

PRACTIONIST Studio

PRACTIONIST Modelling Editors

PRACTIONIST Code Generator

Eclipse

PRACTIONIST Methodology

Development Process

PAML

UML

i* notation

PRACTIONIST Runtime & Framework

PAIT

PAM

PRACTIONIST Packages

Java

Prolog (TuProlog)
**PRACTical reasONIng sySTem**

**Built-in services**
- Belief logic
- Deliberation mechanisms
- Means-ends reasoning
- Planning
- Plan executions

**Main features**
- Goal model
- Plan library
- Perceptors
- Effectors
- Belief base
Semi-formal visual modelling language for specifying, modelling and documenting BDI multi-agent systems, designed using the development methodology defined within the PRACTical reasONI ng sySTem (PRACTIONIST)

Main features:
- Contains general meta-classes to model intentional components of BDI agents
- Includes meta-classes specific to PRACTIONIST-based systems
- **Kernel package** (artifacts, agents and their components)
- **Mental Attitudes** package (intentional attitudes such as beliefs, goals and plans)
- **BDIEntities package** (artifacts and agents)
- **Interactions package** (perceptors, perceptions, effectors and actions)
- **Planning package** (plan body activities)
- **Requirements package** (i* notation)
Main goals

- Support (i) the development methodology of the framework, (ii) and the graphic modelling of the main abstractions characterizing the PRACTIONIST agent
- Support the realization of agent-based applications according to PRACTIONIST model, from design to the code generation

Implementation

- Developed in Java (cross-platform)
- Built using the Eclipse support
- Implementation of the PAML

Release-build policies

- Open source
- Eclipse plugins or standalone RPC
PS – Developed editors

- **UML2** based
  - *Class editor*
  - *Use Case editor*

- **i* based
  - *Strategic Dependency (SD) editor*
  - *Strategic Rationale (SR) editor*

- **PRACTITIONIST** specific
  - *Goal editor*
  - *Domain editor*
  - *Plan editor*
  - *Plan Body editor*
  - *Effector/Action – Perceptor/Perception editor*
  - *Agent editor*
Integrated development using the PS-Project

Common editing facilities

Unified underlying model

Model View

Entity reuse in several views of the system

Validation check

Automatic code generation facility
PS – Main GUI
- **Custom Eclipse Project** supporting the right organization of diagrams and source code
- **Automatic creation** of the sections regarding the different phases of the methodology
- **src** folder containing the generated source code of the agents involved in the designed system
Centralized view for the model underlying a structured phase of the methodology

Used to start the drag and drop feature

Used to start the “Delete from View” e “Delete from Model” actions
Case study: PSTS

- PRACTIONIST Stock Trading System

- System (with a high complexity) supporting stock markets’ operations and decisions

- Other existing agent-based systems in critical fields such as financial and stock trading
PS - Strategic Dependency Editor
PS - Goal Editor
PS - Plan editor
PS - Effector/Perceptor Editor

[Diagram showing a network of connections between 'Effector' and 'Perceptor' with actions and nodes labeled: GetStopProfit, GetClosePriceList, GetProfitTarget, GetStopLossAction, GetRiskParameters, GetToleranceAction, GenerateGoalComputeProfit.]
PS – Agent Structure

- HoldingStocksManager
- Analyst

Agent’s Properties

**Choices**
- ManageRiskyStock
- UpdateMonitoringStockList
- ComputeStopLoss
- ComputeRisk
- ComputeStopProfit
- ComputeProfit
- ManageProfitableStock
- ManageHoldingStock
- ComputeProfitTarget

**Feature**
- ManageRiskyStock
- UpdateMonitoringStockList
- ComputeStopLoss
- ComputeRisk
- ComputeStopProfit
- ComputeProfit
- ManageProfitableStock
- ManageHoldingStock
- ComputeProfitTarget

**Buttons**
- Add
- Remove
- Up
- Down

**Buttons**
- Edit
- New

**Buttons**
- OK
- Cancel
public class ACLMessagePerceptor extends AbstractPerceptor
{

    public Perception perceive()
    {
        // TODO: Insert your conditions here in order to return
        // the perception
        if (false)
        {
            // TODO: Insert the Perception's parameters here
            return new ACLMessagePerception();
        }
        return null;
    }
}
public class ACLMessagePerceptor extends AbstractPerceptor
{

class GetRiskParameters implements Effector
{

public class GetStopLossAction
{
    // TODO: Insert your conditions here in order to return the perception
    if (false)
    {
        Action perform(Action action)
        {
            // TODO: Auto-generated method stub
            if (action instanceof GetStopLossAction)
            {
                GetStopLossAction getStopLossAction =
                    (GetStopLossAction) action;
                // TODO: Manage the action execution and its success condition
                return getStopLossAction;
            }
            if (action instanceof GetToleranceAction)
            {
                GetToleranceAction getToleranceAction =
                    (GetToleranceAction) action;
                // TODO: Manage the action execution and its success condition
                return getToleranceAction;
            }
        }
    }
    return null;
}

Source code generation of the designed system from the developed diagrams
Implementation of the interfaces and extension of the classes of the framework
Source code generation of the designed system from the developed diagrams

Implementation of the interfaces and extension of the classes of the framework
public class HoldingStocksManager extends AbstractAgent {

    protected void initialize(){
        //addBeliefSet("C:/PS_Example/pl/holdingStocksManager.pl");
        //***************Perceptors***************
        // TODO:Remember to put the perceptor's parameters here
        addPerceptor(new ACLMessagePerceptor());
        //***************Effectors***************
        // TODO:Remember to put the effector's parameters here
        addEffector(new GetRiskParameters());
        //***************Goals*****************
        // TODO:Remember to put the goal's parameters here
        registerGoal(new ManageRiskyStock(), "");
        //***************Goals**Relations**********
        // TODO:Remember to put the goal's parameters here
        addPlan(TopLevelPlan.class, "TopLevelPlan");
        // TODO:Remember to put your commit goal here
        commit(new Start(null));
    }
}
MO2L – MOdelling TOOLs

- MO2L Architecture
  - Exploit Eclipse support
  - Common infrastructure

- BilikUML
  - UML2 compliant
  - To be improved with other diagrams (especially sequence and activity)

- RequiSuite
  - Requirements analysis tools
  - Lack of use case description facilities
Future works

- Other diagrams (with focus on interaction)
- Improve automatic code generation
- Reverse engineering
- Documentation management
- More applications