# Design and Development of Intentional Systems with PRACTIONIST *Studio*

Angelo Marguglio*, Giuseppe Cammarata*, Susanna Bonura*, Giuseppe Francaviglia*
Michele Puccio*, and Vito Morreale*.
**Intelligent Systems* unit - R&D Laboratory
ENGINEERING Ingegneria Informatica S.p.A.

***Abstract*— In this paper we present PRACTIONIST *Studio*, which is an integrated design and development environment for BDI agent-based systems, providing facilities and tools to represent the concepts and intentional elements underlying such a model as well as several common features offered by UML-based tools.**

**PRACTIONIST *Studio* aims at bridging the gap between the increasing trend of developing BDI-based multi-agent systems and the availability of tools for their design. It supports developers from early requirements analysis to automatic code generation.**

**More in detail, we first give an overview of the modelling editors provided with PRACTIONIST *Studio*. Then some fragments of the modelling and development approach when applied to a real-world implementation are presented. Such a complex system is the PSTS (PRACTIONIST Stock Trading System), which is aimed to monitor investors' stock portfolio by managing risk and profit and supporting decisions for on-line stock trading, on the basis of investors' trading rules and their risk attitude.**

## I. INTRODUCTION

Recently the increasing complexity and the introduction of new Web and networking technologies are making it difficult for designers to entirely model systems and for operators to handle effectively all unpredictable situations. The effort of scientific communities is towards the building of systems where interactions among components cannot be thoroughly planned and anticipated.

In other words an open issue is to investigate the modelling of systems where the collective behavior of their parts is related to the emergence of properties that can hardly, if not at all, be infered from properties of the parts. Aristotele stated that *"The whole is more than the sum of its parts"*; with this assertion he had already, more than two thousand years ago, defined what complex systems are.

Several authors (e.g. McCharty [1]) have argued that in certain situations, the so-called intentional stance [2] of systems can aid to efficiently predict, explain, or define their behaviour, without having to understand how they actually work. Therefore, some systems may be better explained in terms of mental qualities or attitudes, rather than in terms of conventional physical phenomena or design artifacts, i.e. by specifying the so-called intentional stance of systems.

In the context of the development of intentional systems, the agent-oriented approach plays a central role, due to the vast number of theories and models that have been developed for twenty years. Moreover, with regards to complex systems, Georgeff [3] asserts that *"the notions of complexity and change will have a major impact on the way we build computational systems, and that software agents - in particular BDI agents - provide the essential components necessary to cope with the real world"*.

The Belief-Desire-Intention (BDI) architecture [4] suggests that the development of agents should rely on the specification of some mental states, i.e. beliefs, desires, and intentions, which are very intuitive for people to understand. Indeed, beliefs represent information the agent has about the world; desires represent state of affairs the agent wishes to bring about and intentions are desires that it has committed to achieving.

Although BDI model has become a very attractive approach for dealing with the complexity of modern software applications, engineering such systems is still a challenge due to the lack of effective tools and actual implementations of very interesting and fascinating theories and models.

In past years we developed the PRACTONIST Framework [5], which is a set of Java libraries to develop agent-based systems according to the BDI model. PRACTIONIST adopts a goal-oriented approach and a clear separation between the deliberation and the means-ends reasoning, and consequently between the states of affairs to pursue and the way to do it. Moreover, PRACTIONIST allows developers to implement agents able to reason about their beliefs and the other agents' beliefs, expressed by modal logic formulas.

Due to the differences between the objects and agents [6], design tools used to model object-oriented systems do not represent the best way to design and develop agent-oriented softwares, especially BDI agent systems. In addition, several existing MAS modelling tools (e.g. INGENIAS Development Kit [7]) suffer a too strong tie-up with specific methodologies for the development of MAS. Moreover, some of them cover well only a subset of development phases (e.g. TROPOS Tool for Agent Oriented visual Modeling [8]). Other tools are simple prototypes and do provide a very limited assistance when developing agents and their components. In practice, none of them can be directly adopted (or extended) to design and develop multi-agent systems according to the PRACTIONIST model.

Thus we developed the PRACTIONIST *Studio*, which is the novel visual environment to model, design and develop PRACTIONIST-based systems. The PRACTIONIST *Studio*

has been developed by using several Eclipse[1] plug-ins, such as: UML2, Eclipse Modelling Framework (EMF), Graphical Editing Framework (GEF), Graphical Modeling Framework (GMF) and other Eclipse extensibility features. It supports the representation of the concepts underlying the BDI model and part of UML 2.0 meta-model as well as several features common to (commercial) well-known UML-based CASE tools, such as unified underlying model for all diagrams within an project, consistency check within diagrams, editing facilities (e.g. cut and paste, unlimited undo and redo, and so forth).

In this paper we present an overview of the modelling editors and facilities included in PRACTIONIST *Studio*, along with some fragments of modelling and development of a real system, i.e. the PSTS (PRACTIONIST Stock Trading System).

The paper is organized as follows: in section II we first present the PSTS as a running example. Then we give an overview of the PRACTIONIST suite (section III), while in section IV PRACTIONIST *Studio* is described in details; in section 4 some of the models of the PSTS developed with PRACTIONIST *Studio* are shown, while in section 5 we present how PRACTIONIST *Studio* has supported the implementation of the PSTS. Finally, we point out our intended future work and give some conclusions.

## II. Running example

Systems supporting stock markets' operations and decisions are an example of systems with a high complexity. Here elementary building blocks can be individual traders, each making buying and selling decisions from his/her own perspective.

It should be noted that systems for stock trading management have been implemented by adopting agent technology and related approaches. Among them, Wang et al. [9] have presented a lightweight, distributed, intelligent agent-based financial monitoring system that monitors and reports on transactions within an organization. In such a prototype system, the intelligent agents are assisted by a formal conceptual model that makes up an unambiguous understanding of the institution, the transactions, the instruments involved, and the business processes.

In [10], Feng and Jo present a system, called AST (Agent-based Stock Trader), which is a stock-trading expert based on intelligent agents using the BDI model of agency. Finally, Davis et al. [11] have designed a system around portfolio management tasks that include eliciting user profiles, collecting information on the users portfolio position, monitoring the environment on behalf of the user, and making decision suggestions to meet the users investment goals.

The existence of such implementations confirm that agent-based systems can benefit the development of complex systems even in critical fields such as financial and stock trading. For this reason we chose to use such an application domain to test and evaluate the PRACTIONIST *Studio* by designing the PRACTIONIST Stock Trading System (PSTS), which is also used as a running examble throughout the paper.

[1]http://www.eclipse.org/

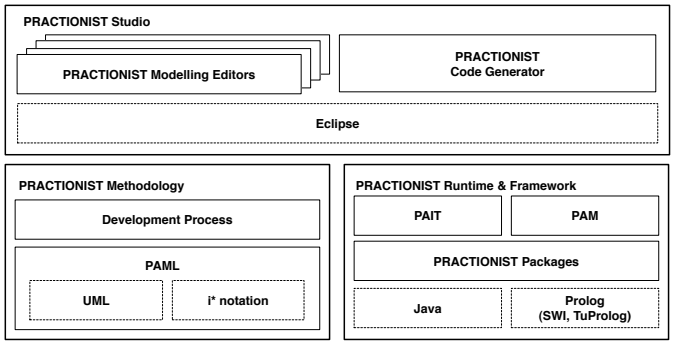


Fig. 1. PRACTIONIST Suite.

PSTS is a PRACTIONIST-based system, which is able to monitor investors' stocks portfolio, to monitor and manage risks, to manage and optimize profit and to support decisions regarding on-line stock trading, by taking into account investors' trading preferences and rules (i.e. stop loss, stop profit, profit target, tolerance, maximum budget to be inevested per week) and their degree of willingness to risk.

## III. PRACTIONIST Suite

As stated above, in order to exploit the full potential offered by the agent-oriented paradigm, and particularly the BDI model, the support of efficient integrated development tools and methodologies is required to produce real-world (and sometime complex) software systems.

Our solution to this issue is PRACTIONIST (PRACTIcal reasONIng sySTem), which is an integrated suite providing the following tools (see Fig. 1):

- *PRACTIONIST Methodology*, including the *PRACTIONIST Agent Modelling Language (PAML)*, which is a UML-based modelling language, and an iterative development process;
- *PRACTIONIST runtime and framework (PRF)*, which provides the APIs to develop PRACTIONIST-based agent systems by defining the execution logic and providing the builtin components according to such a logic;
- *PRACTIONIST Studio*, a visual modelling, design and development environment supporting the representation and specification of the concepts underlying the BDI model as well as several features present in other UML-based tools.

The focus of this paper is on modelling facilities provided by PRACTIONIST *Studio*, which is described in details in the following sections, while in the remaining part of this section an overview of the other two components of PRACTIONIST is given.

### A. Methodology

PRACTIONIST Methodology is based on an iterative and incremental development process supporting developers from early requirements analysis to coding, debugging and testing of agents and artefacts (according to the A2A approach [12]). It is the result of the following interacting tasks: (i) theoretical analysis of requirements that similar processes should meet,

(ii) theoretical analysis of novel features introduced with PRACTIONIST that need to be specified at the design time and (iii) practical application of PRACTIONIST in real cases as well as its integration with other technologies, such as Web services.

It should be noted that the development process is still a work in progress and our research is going towards the definition of a more general framework for process and software engineers, with the aim of providing tools to define/customize processes as well as full support to the usage of them during development phases.

As an important part of our methodology, PAML is a UML-based visual modelling language for specifying, modelling and documenting BDI multi-agent systems. Its meta-model contains general metaclasses to model intentional components of BDI agents, such as beliefs, goals and relations among them, plans and so forth. It also includes metaclasses specific to PRACTIONIST and the development of related systems.

PAML extends the Agent modelling Language (AML) [13], a semi-formal visual modelling language for specifying, modelling and documenting systems that incorporate general concepts drawn from the Multi-Agent Systems (MAS) theory. AML can be used to build models that consist of autonomous entities able to observe and interact with their environment using complex interactions and aggregated services.

Thus rather than extending the UML and building a new modelling language, PAML extends the AML, particularly for the concepts underlying the Belief-Desire-Intention (BDI) model. Indeed, AML already provides a Mental section, that lets the modelling of mental attitudes of autonomous entities having deliberative and motivational states. Moreover, PAML also extends the UML [14], in order to meet specific requirements of for developing PRACTIONIST systems.

The overall package structure of the PAML is depicted in the Figure 2.

The detailed description of PAML is out of the scope of this paper. In brief, the `Kernel` package defines the metaclasses to model artefacts, agents and their components as well as architectutal aspects of multi-agents systems. More in detail, the `Mental Attitudes` package defines the metaclasses to model intentional attitudes of PRACTIONIST agents (i.e. belief, desires, intentions, goals and plans), extending the `Mental` package of AML. The `Interactions` package defines the metaclasses to model ways and means agents use to interact with the environment where they live, including perceptors, to listen to relevant external stimuli (i.e. perceptions) and actions, to act over the environment and the effectors that actually execute such actions. The `Planning` package defines the metaclasses to model the body of plans; indeed the body represents the actual sequence of act being executed by the agent. The `BDIEntities` package defines the metaclasses to model artefects and agents, which are the building blocks of the system. Finally, the `Requirements` package defines the metaclasses to support the requirement analysis phase according to the i* notation [15] and use case model.

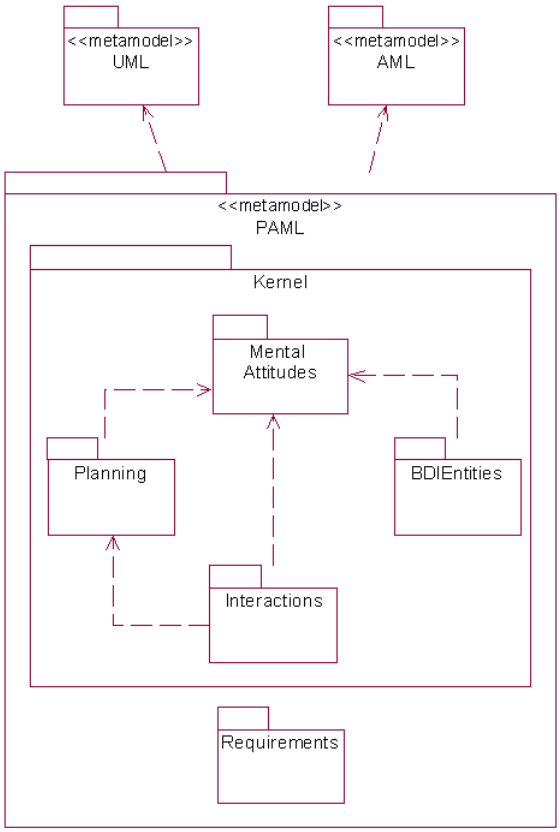


Fig. 2. Overall package structure of the PAML metamodel.

### B. Runtime & Framework

As already mentioned, PRACTIONIST suite provides the framework and the runtime environment, respectively supporting the coding and the execution of BDI agents *(i)* endowed with a symbolic representation about their beliefs, *(ii)* able to proactively deliberate about their intentions, *(iii)* capable of performing reactive behaviours, and *(iv)* endowed with the ability to plan their activities in order to meet some objectives [16].

PRACTIONIST framework supplies the required built-in services that define the computational model of PRACTIONIST agents. This includes the *belief logic*, the *deliberation* mechanisms that produce agent intentions, the way the agent makes *means-ends reasoning* to figure out the means (i.e. plans) to achieve its intentions [17], and the support for the actual execution of such plans. Thus PRACTIONIST agents present a double-layered structure: the bottom layer represents the framework, which defines the execution logic and provides some built-in services implementing such a logic, while the top layer includes the specific agent components to be defined in order to satisfy specific application requirements [5].

More concretely, in order to design a PRACTIONIST agent developers shall specify the following components: *(i) Goal Model*, that is the set of *Goals* the agent could pursue and the relations among them; *(ii) Plan Library*, that is a set of

means, to pursue such goals or to react to the stimuli coming from the environment; *(iii) Perceptors* to receive such stimuli; *(iv) Actions* the agent could perform and the corresponding *Effectors*, and *(v) Belief Base*, that is a set of beliefs and rules on them to model the information about both its internal state and the external world.

Moreover, agents are endowed with the ability to dynamically build plans (i.e. *Planning*). Finally the management of perceptors and effectors is part of the agent *core services* infrastructure.

The framework also includes the PRACTIONIST Agent Introspection Tool (PAIT), a visual integrated monitoring and debugging tool, which supports the analysis of the agent's state during its execution. In particular, the PAIT can be suitable to display, test and debug the agents' mental attitudes (i.e. beliefs, desires, and intentions) and their execution flow, in terms of active behaviours. Each of these components can be observed at run-time through a set of specific tabs.

Furthermore, the runtime and framework supplies facilities and built-in components for autonomically manage PRACTIONIST-based applications and external resources.

Finally, it is worth mentioning that PRACTIONIST runtime and framework has been designed on top of JADE[2], a widespread platform compliant to the FIPA[3] specifications, that provides some core services, such as a communication support, interaction protocols, life-cycle management, and so forth.

## IV. PRACTIONIST *Studio*

PRACTIONIST *Studio* is a modelling, design and development tool for BDI agent systems according to the PRACTIONIST mdoel. It includes a set of visual modelling editors, some of which are based on UML 2.0 metamodel, whereas others are based on PRACTIONIST Agent Modelling Language (PAML). More accurately, a brief description of such visual modelling editors follows:

- *i* [15] based editors*:
  - *Strategic Dependency (SD) editor*: to describe the dependency relationships among various actors in an organizational context;
  - *Strategic Rational (SR) editor*: to describe stakeholder interests and concerns and how they might be addressed by various configurations of systems and environments;
- *UML2.0 based editors*:
  - *Use Case editor*: to model use cases and system funcionalities from the actor's point of view;
  - *Class editor*: to model static structures of a system or of its parts;
- *PRACTIONIST specific editors*:
  - *Agent editor*: to model agents and specify their components;
  - *Domain editor*: to model facts about the world the agent can believe or not;
  - *Goal editor*: to model agent goals and the relationships among them;
  - *Effector/Action - Perceptor/Perception editor*: to model the means agents use to interact with their environment;
  - *Plan editor*: to model the features of plans agent can adopt to pursue their intention;
  - *Plan Body editor*: to model the body of plans, in terms of (simple or complex) flow of acts.

PRACTIONIST *Studio* aims at bridging the gap between the increasing need of development of multi-agent systems and the availability of tools for their design. Indeed, it can be used in the same way as other software modelling tools to develop multi-agent systems as it supports developers from the requirements analysis to the code generation of agents.

As many well-known CASE tools, PRACTIONIST *Studio* provide all the features that support the development of complete and consistent visual models, such as:

- *Unified model*: all diagrams created inside a PRACTIONIST project share the same model (i.e. an instance of the meta-model), whereas each generic GMF diagram file has usually its own model file.
  Sharing the same model file means sharing the same command stack, allowing us to execute cross-checks among elements and consequently model more complex and greater systems as a whole;
- *Drag and Drop* support: a PRACTIONIST project has its own model view, where the developed model is displayed as a tree. From this view it is possible to *drag and drop* the elements into diagrams, enabling us to use the same elements in different diagrams as well. Thus, if an element is modified in a diagram, it will be updated in all the other diagrams.
- *Delete from diagram* and *delete from model* actions: in a GMF diagram the *delete from model* action is enabled by default, so when an element is deleted in the diagram it is also automatically deleted from the model. Such a behaviour was modified in order to get the *delete from view* action as well, and thus have a more flexible model management.

For the development of PRACTIONIST *Studio*, the support provided by the Eclipse environment has been fully exploited. As a consequence:

- a PRACTIONIST project, which is a custom Eclipse Java project, provides several sections where developers can create their own diagrams and the source folder that will contain the generated source code;
- the model view of a PRACTIONIST project is a custom Eclipse view that displays the unified model underlying the project;
- the PRACTIONIST Java code can be generated starting from diagrams in a simple way.

The hierarchical representation of a PRACTIONIST *Studio* project is composed by several sections where developers can create their own diagrams and manage the source code generated; besides, the model view of the project is a custom Eclipse view that presents the unified model underlying the

[2]http:jade.tilab.com
[3]http://www.fipa.org

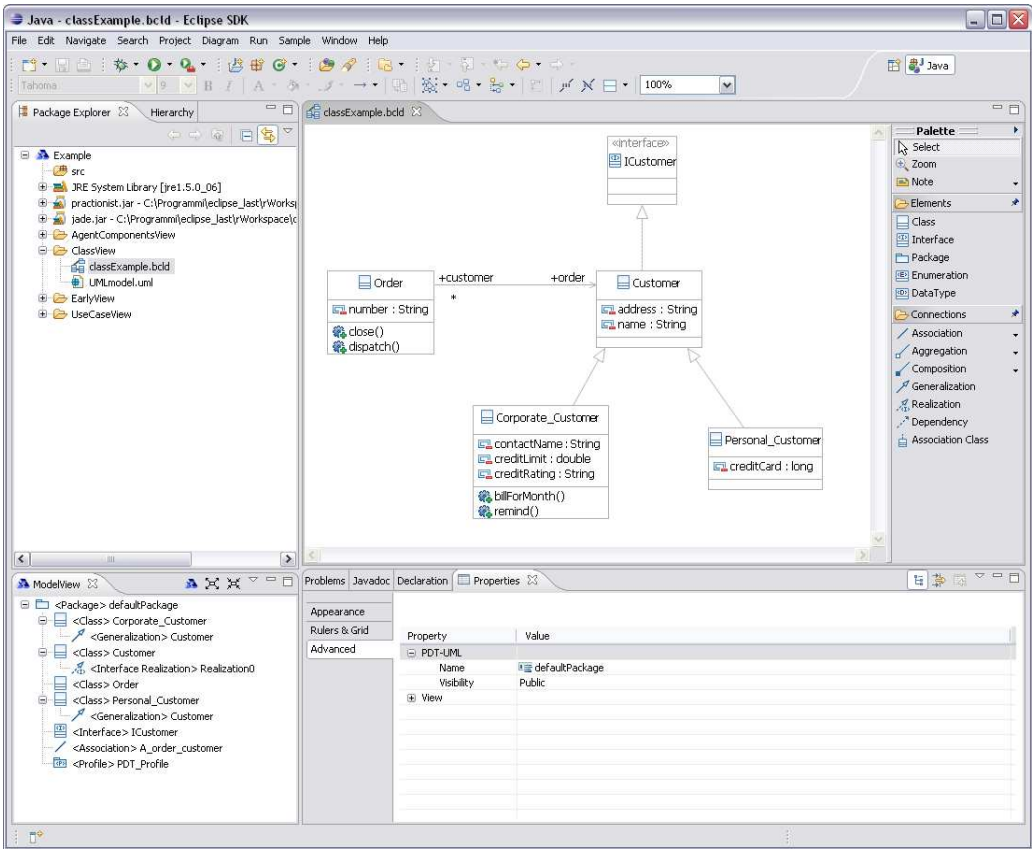


Fig. 3. A snapshot of the PRACTIONIST *Studio*.

project; finally, the Java code generation in a very simple process.

## V. Modelling with PRACTIONIST *Studio*

Throughout this section we present an overview on how to model a PRACTIONIST system by using the facilities and functionalities offered by PRACTIONIST *Studio*. This is done by describing the design of some components of the PSTS and showing some snapshots of models developed with PRACTIONIST *Studio*.

As a complex system should be able to select at runtime the best behaviour on the basis of the current situation, we believe that in the requirements analysis phase, goals can be used as an abstraction to model the functions around which the systems can autonomously select the proper behaviour [17].

The requirements state that the goals of the PSTS (PRACTIONIST Stock Trading System) must be the monitoring of investors' stock portfolio in terms of risk and profit management and supplying a decision support for the on line stock trading, by considering investors' trading rules (i.e. stop loss, stop profit, profit target, tolerance, maximum budget to be inevested a week) and their degree of willingness to risk. Besides, if users so wish, the PSTS has to be able to replace orders (*system orders*) which are too risky or profitable, asking a broker to execute them. Moreover, through the PSTS users have to be allowed to place *market orders* (a market order is a buy or sell order to be executed by the broker immediately at current market prices) and *limit orders* (a limit order is an order to buy a stock at no more - or sell at no less - than a specific price).

### A. Modelling the organizational environment of the PSTS

In order to provide a deeper level of understanding about how the PSTS can be embedded in the organizational environment, the relevant stakeholders of the application domain were modelled, where also the system-to-be (the PSTS) was introduced as another actor, along with the dependencies among them in terms of goals, tasks or resoures. In other words, it was created a Strategic Dependency (SD) diagram. Indeed the SD model focuses on the intentional relationships among organizational actors.

Referring to the PSTS case study, the SD diagram was modelled by using the SD editor of PRACTIONIST *Studio*, which is shown in the Figure 4, where actors, depicted as circles, are the `Investor`, the `PSTS`, `Yahoo` and the `Bank`; the dependencies among the actors are depicted as arrowed lines connected by a graphical symbol varying according to the dependum: a rectangle if the dependum is a resource, a rectangle with rounded corners if the dependum is a hard goal

More in detail, the `Investor` would like to have a system (the `PSTS`) that is able to provide information about stock market (`Get Stock Information`), to provide updated and detailed data about his/her Stock Portfolio (`Get Portfolio Information`), to manage risky and profitable stocks of the portfolio(`Default Manage Risky`

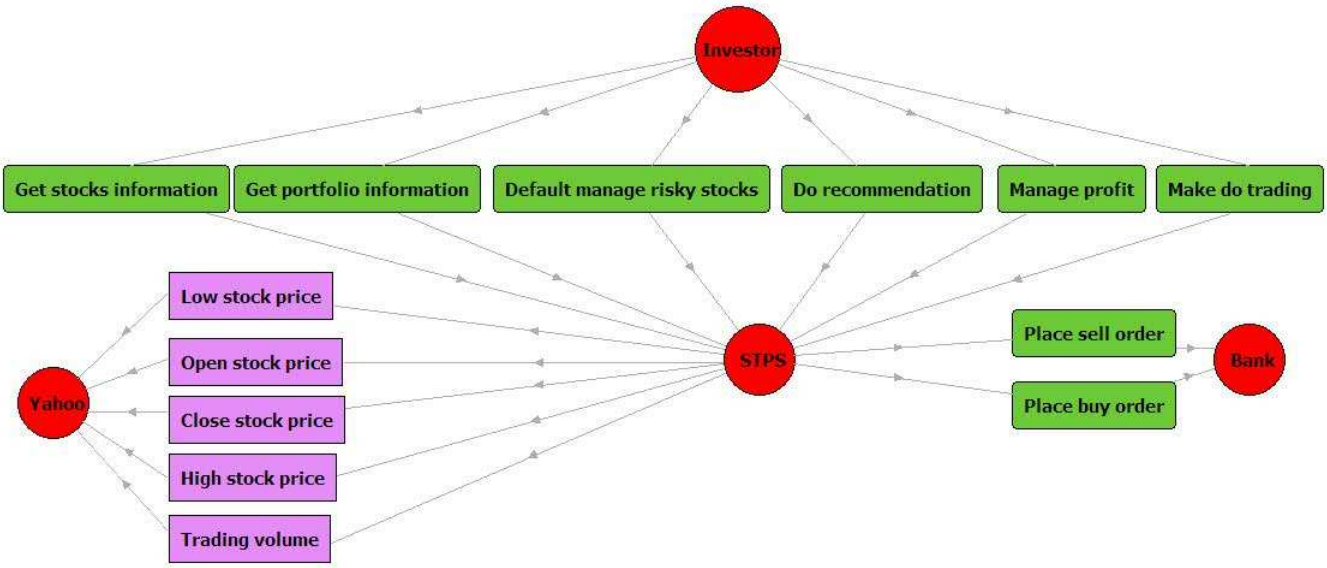


Fig. 4. Strategic Dependency model.

`Stocks` and `Manage Profit`), to give adivice on stocks to be bought or sold (`Do recommendation`) and allow to do trading autonomously (`Make do trading`). The `PSTS` depends on Internet (i.e. in particular `Yahoo`) to obtain every day the current data about stocks (opening and closing stock prices, highest and lowest stock prices and volume of trading). Finally, the PSTS depends on a broker (i.e. the `Bank`), to `Place sell order`s and `Place buy order`).

### *B. Modelling agents' goals*

Architectural analysis of the PSTS produced the entities classified as agents. In the resulting design they are

- the `Trader`, the agent in charge of managing all kinds of order (i.e. market, limit and system orders) by asking the broker (the `Bank`) to place them;
- the `Analyst`, the agent liable for executing the market analysis;
- the `Advisor`, the agent which interprets the `Analyst`'s signals and does recommendations to investors;
- the `HoldingStockManager`, the agent which monitors investor's stock current prices and places sell order for that stocks resulting too profitable (if they have reached the profit target indicated by investor or the profit has descended below the stop profit of investors) or too risky (if their value has descended below the stop loss of investors).

It is worth noting that the goal turns out to be an interesting abstraction related to autonomous entities for the development of software systems whose requirements are not entirely known at design time. Thus, the explicit representation of goals and the ability to reason about them from agents, plays an important role in the modelling phase. By using PRACTIONIST *Studio*, a designer can specify for each agent, goals it could pursue and their properties, and all relations among such goals.

More in detail, for each goal it is possible to define the success condition, the applicability condition stating whether it is possible (given current conditions) to achieve that goal and the cancel condition stating in which situations the agent should give up to pursue a goal.

Regarding the relationships among goals, PRACTIONIST *Studio* allows to model *(i)* the inconsistency between two goals (if the designer want to declare that if a goal succeds, the other one fails), *(ii)* the entailment (if the designer want to declare that if a goal succeeds, then also the other one succeeds), *(iii)* the fact that a goal is a precondition of another goal (that is the fact that a goal must succeed in order to be possible to pursue another goal), *(iv)* the dependence (if the designer want to declare that a goal is precondition of another goal and must be successful while pursuing this last one). A formal definition of the goal relationships in PRACTIONIST can be found in [17].

For example, referring to the `HoldingStockMaganer`, the properties of goals and their relationships were modelled in PRACTIONIST `Studio` as in Fig. 5

The main objective of the `HoldingStockMaganer` is `ManageHoldingStock`: it is *applicable* if the agent believes that a new current price is available for one of the holding stocks (`NewStockPriceReceived` predicate), or that the investor has bought a new stock (`NewStockBought` predicate); indeed in the last case, the agent has manage such a new stock.

The agent monitors both the profit and the risk of a stock, so the `ManageHoldingStock` depends on the `ManageRiskyStock` and the `ManageProfitableStock` goals; the agent continues to pursue these goals until the price and the amount of a stock do not change, otherwise both goals have to be cancelled.

To manage the risk and the profit of a stock, the agent has to analyze its risk and profit on the basis of investors' rules, so the goals `ComputeRisk` and `ComputeProfit`


Fig. 5. Goal Diagram related to the *HoldingStocksManager* agent.

entail respectively the goals `ManageRiskyStock` and `ManageProfitableStock`.

Finally, the dependency relationships of the `ComputeProfit` and `ComputeRisk` goals were modelled; the `dependee` goals have to be achieved to compute the value of some investor's trading rules, that is the `profit target`, the `stop profit`, and the `stop loss`.

### C. Modelling agents' plans

In the BDI agent model, another key element is the library of plans, as it represents the set of *recipe* to meet agent's intentions.

In PRACTIONIST *Studio* it is possible to declare a set of plans an agent has to own (the *plan library*), to specify the activities it should undertake in order to achieve its intentions, or handle incoming perceptions, or react to changes of its beliefs.

Each plan presents five slots: (i) *practical*, which defines the kind of events the plan is able to manage; (ii) *applicable*, to define the formula that has to be believed as true by the agent in order to actually adopt a practical plan; (iii) *invariant*, to define the condition to hold during the execution of the plan; (iv) *cancel*, to define when the plan has to be stopped with failure; (v) *success*, to define the formula has to believed as true by the agent then the plan ends with success.

But, the way a certain event is handled has to be specified in the body, which is an activity that can contain a set of *acts* ( [17]), such as desiring to pursue some goal, adding or removing beliefs, sending ACL messages, doing an action and so forth.

Thus, in order to model the plan's body, a designer can use the Plan Body editor of PRACTIONIST *Studio*. In Fig. 6 it is shown the Plan Diagram where the plan library of the *HoldingStocksManager* was modelled.

The *HoldingStockManager* agent has to manage the profit and the risk of a holding stock every time a new price is available or the investor has placed a new buy order for a stock already held, or a new stock is bought. Therefore it was equipped with the *ManageProfitForNewOrder*, *ManageProfitForNewPrice* plans, regarding the profit management, and *ManageRiskForNewOrder* and *ManageRiskForNewPrice* plans, regarding the risk management; as shown in Fig. 6. Success and cancel conditions of these plans refer to the goal success and cancel conditions, whereas they have a proper applicable condition.

Finally, other plans were also modelled, for example to handle the stimuli received from the environment (i.e. a stock price updating or a stock placed order) and to compute the investor's trading rules (that is, to manage the *ComputeStopLoss* goal, etc.).

## VI. Coding with PRACTIONIST *Studio*

As stated, PRACTIONIST *Studio* supports the actual implementation of BDI agent systems by providing an automatic code generation facility, which produces template or partially filled parts of source code according to the developed models and relying on the PRACTIONIST Framework.

In this section the source code related to the *ManageHoldingStock* goal generated by PRACTIONIST *Studio* is shown. A snippet of the goal follows:

```
/**
 * @generated
 */
public class ManageHoldingStock implements Goal
{
   ....

   /**
    * @generated
    * @see org.practionist.core.GoalProfile#applicable())
    */
   public boolean applicable()
   {
        // TODO: Insert the right variables value
       return
       beliefBase.bel(AbsPredicateFactory.create
           ("newStockPriceReceived(arrived: X)"))
        || beliefBase.bel(AbsPredicateFactory.create
```

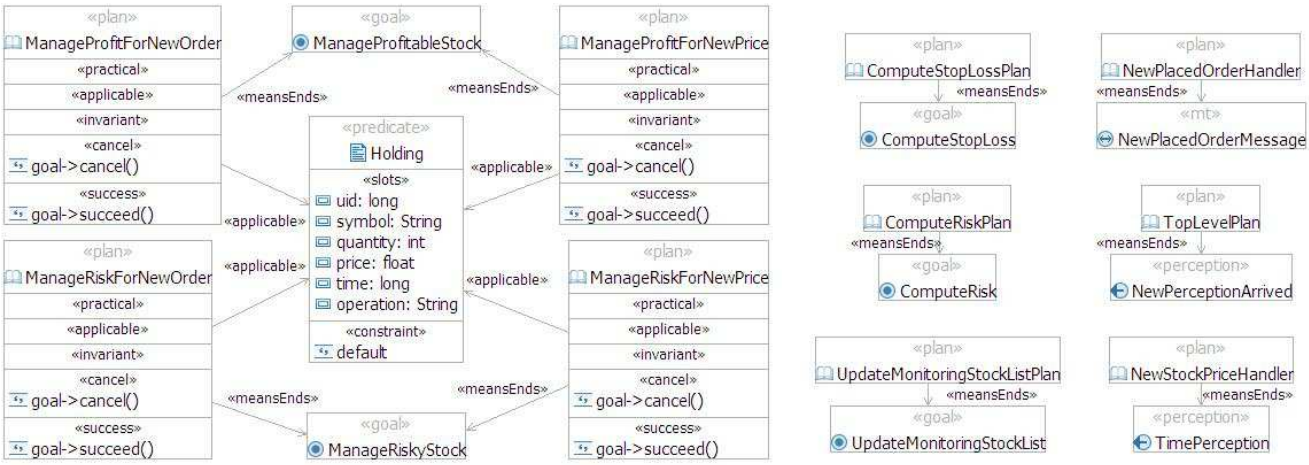


Fig. 6. Plan Diagram related to the *HoldingStocksManager* agent.

```
        ("newPlacedOrderReceived(uid: X, stockSymbol: X,
            operation: X, quantity: X, price: X)"));
   }

   /**
    * @generated
    *
    * @see org.practionist.core.GoalProfile#succeed())
    */
   public boolean succeed()
   {
      // TODO: Insert the right variables value
      return beliefBase.bel(AbsPredicateFactory
         .create("managed(investor: X, stockName: X)"));
   }
   ....
}
```

This general implementation produced by the code generator of PRACTIONIST *Studio* should be customized according to the specific requirements for this goal. An example follows:

```
/**
 * @generated
 */
public class ManageHoldingStock implements Goal
{
   .....

   /**
    * @generated
    * @see org.practionist.core.GoalProfile#applicable())
    */
    public boolean applicable()
    {
       return beliefBase.bel(AbsPredicateFactory
          .create("newStockPriceReceived(arrived: true)"))
          || beliefBase.bel(AbsPredicateFactory
             .create("newPlacedOrderReceived
                (uid: %, stockSymbol: %,
                operation: %, quantity: %, price: %)",
                uid, stockSymbol,
                operation, quantity, price));
   }

   /**
    * @generated
    *
    * @see org.practionist.core.GoalProfile#succeed())
    */
    public boolean succeed()
    {
       return beliefBase.bel(AbsPredicateFactory
          .create("managed(investor: %, stockName: %)",
            investorID, symbol));
    }
    ....
}
```

In this snippet, the designer just needs to detail the right variables of the predicates (in this example a parametrized form of the predicates has been adopted, using the symbol % and then adding values). That is, in order to express the applicability and success conditions of the goal, the corresponding beliefs were customized by replacing the aforementioned variables with the values that characterise the goal.

A code snippet of the dependency relation between the `ManageHoldingStock` and the `ManageProfitableStock` goals follows:

```
/**
 * @generated
 */
public class GR_ManageHoldingStock_ManageProfitableStock
          implements DependencyRel
{
   public Goal verifiesRel(SerializableGoal goal1,
        SerializableGoal goal2)
   {
      if (goal1 instanceof ManageHoldingStock
         && goal2 instanceof ManageProfitableStock)
            return new ManageProfitableStock();
      return null;
   }
}
```

In this example, every `ManageHoldingStock` goal depends on the `ManageProfitableStock` goal, without specifying any information about stocks. Thus, this code should be now customized according to the designer's needs:

```
/**
 * @generated NOT
 */
public class GR_ManageHoldingStock_ManageProfitableStock
      implements DependencyRel
{
   public Goal verifiesRel(SerializableGoal goal1,
      SerializableGoal goal2)
      {
         if (goal1 instanceof ManageHoldingStock
            && goal2 instanceof ManageProfitableStock)
            {
               ManageHoldingStock g =
                   (ManageHoldingStock) goal1;
               return new
                 ManageProfitableStock(g.getUID(),
                             g.getSymbol());
            }
            return null;
      }
}
```

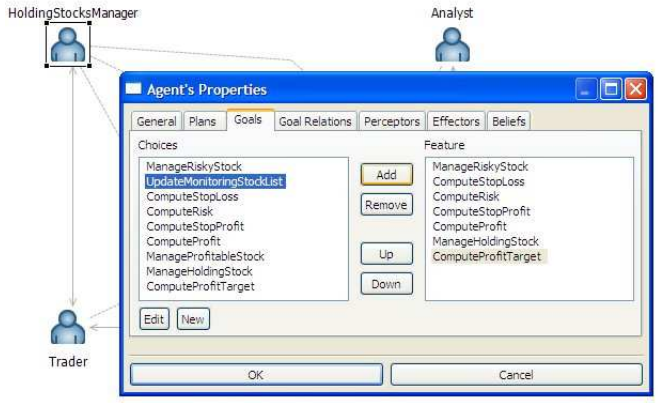


Fig. 7. Agent Diagram related to the *HoldingStocksManager* agent.

In this snippet, the designer has just to detail some properties of the dependee goal; here the dependee goal refers to the same user and symbol of stock of the dependent goal.

Finally, a code snippet of the HoldingStockManager agent class is shown:

```
/**
 * @generated
 */
protected void initialize()
{
   addBeliefSet("/home/pl/holdingstockmanager.pl");

   /****************Goals****************/
   // TODO:Remember to put the goal's parameters here
   registerGoal(new ManageHoldingStock(), "");
   registerGoal(new ManageProfitableStock(), "");
   registerGoal(new ComputeProfitTarget(), "");

   /***********Goals**Relations***********/
   // TODO:Remember to put the relation's parameters here
   registerRelation(new
       GR_ManageHoldingStock_ManageProfitableStock(), "");
   registerRelation(new
       GR_ManageRiskyStocks_ComputeRisk(), "");

   /****************Plans****************/
   // TODO:Remember to put the plan's parameters here
   addPlan(ManageRiskyStock.class, "ManageRiskyStock");
   addPlan(NewStockPriceHandler.class,
       "NewStockPriceHandler");
}
```

The Fig. 7 shows a snapshot of the Agent Diagram. More in detail, by means of such a diagram it is possible to look at the list of all the intentional elements were modelled and to choose that ones that the designer wants associate to an agent.

It is worth noting that the code generator of PRACTIONIST *Studio* is able to produce the agent source code by putting the entities which were designed in the previous modelling phases together; obviously, this code should be customized according to the designer's needs.

## VII. Conclusions and Future Work

In this paper, through a running example, i.e. the PSTS, we presented PRACTIONIST *Studio*, the visual modelling, design and development environment of the PRACTIONIST suite, which is the set of tools we have been developing to implement agent systems according to the BDI model.

PRACTIONIST *Studio* supports the development of agents endowed with a lot of useful built-in capabilities and with a computational model which is more flexible and adaptive than the agent models underlying several commercial and non-commercial frameworks.

Our tool allows the design and development of BDI agent systems from several perspectives, including the representation of intentional attitudes and relationships among them, the way the agents interact with their environment, the activities within a plan. and so forth.

As part of our future work, we aim at further developing PRACTIONIST *Studio* by adding editors for other diagrams (i.e. dynamic views, such as interactions). We also intend to improve service features of the tools, such as reverse engineering and documentation management and automatic generation.

Finally, we have been developing some other real-world applications by using the PRACTIONIST framework, methodology and *Studio*.

## References


[1] J. McCarthy, "Ascribing mental qualities to machines," Stanford University, Tech. Rep. STAN-CS-79-725, 1979.

[2] D. Dennett, *The Intetional Stance*. MIT Press, 1989.

[3] M. P. Georgeff, B. Pell, M. E. Pollack, M. Tambe, and M. Wooldridge, "The belief-desire-intention model of agency," in *ATAL '98: Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*. London, UK: Springer-Verlag, 1999, pp. 1–10.

[4] A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," in *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991, pp. 473–484. [Online]. Available: http://citeseer.nj.nec.com/rao91modeling.html

[5] V. Morreale, S. Bonura, G. Francaviglia, F. Centineo, M. Puccio, and M. Cossentino, "Developing intentional systems with the practionist framework," in *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN07)*, July 2007.

[6] O. J., "Objects and agents: how do they differ?" *Journal of Object-Oriented Programming*, pp. 50–53, 2000.

[7] J. Pavon, C. Sansores, and J. J. Gomez-Sanz, "Modelling and simulation of social systems with ingenias," *Int. J. Agent-Oriented Softw. Eng.*, vol. 2, no. 2, pp. 196–221, 2008.

[8] D. Bertolini, A. Perini, A. Susi, , and H. Mouratidis, "The tropos visual modeling language. a mof 1.4 compliant meta-model." *Contribution for the AOSE TFG meeting.*, 2005.

[9] H. Wang, J. Mylopoulos, and S. Liao, "Intelligent agents and financial risk monitoring systems," *Commun. ACM*, vol. 45, no. 3, pp. 83–88, 2002.

[10] X. Feng and C.-H. Jo, "Agent-based stock trader," in *Computers and Their Applications*, 2003, pp. 275–278.

[11] D. N. Davis, Y. Luo, and K. Liu, "Combining kads with zeus to develop a multi-agent e-commerce application," *Electronic Commerce Research*, vol. 3, no. 3-4, pp. 315–335, 2003.

[12] A. Ricci, M. Viroli, and A. Omicini, "Programming MAS with artifacts." in *PROMAS*, 2005, pp. 206–221.

[13] I. Trencansky and R. Cervenka, "Agent modelling language (aml): A comprehensive approach to modelling mas," *Informatica*, vol. 29, pp. 391–400, 2005.

[14] O. M. Group, "Unified Modeling Language, Superstructure."

[15] E. S. K. Yu, "Towards modelling and reasoning support for early-phase requirements engineering," pp. 226–235. [Online]. Available: citeseer.ist.psu.edu/article/yu97towards.html

[16] V. Morreale, S. Bonura, G. Francaviglia, M. Cossentino, and S. Gaglio, "PRACTIONIST: a new framework for BDI agents," in *Proceedings of the Third European Workshop on Multi-Agent Systems (EUMAS'05)*, Brussels, Belgium, 2005, p. 236.

[17] V. Morreale, S. Bonura, G. Francaviglia, F. Centineo, M. Cossentino, and S. Gaglio, "Goal-oriented development of BDI agents: the PRACTIONIST approach," in *Proceedings of Intelligent Agent Technology*. Hong Kong, China: IEEE Computer Society Press, 2006.