From Stakeholder Intentions to Software Agent Implementations

Loris Penserini, Anna Perini, Angelo Susi, and John Mylopoulos

ITC-IRST, Via Sommarive 18, I-38050, Trento, Italy
{penserini, perini, susi}@itc.it, jm@cs.toronto.edu

Abstract. Multi-Agent Systems have been proposed as a suitable conceptual and technological framework for building information systems which operate in open, evolving, heterogeneous environments. Our research aims at proposing design techniques and support tools for developing such complex systems. In this paper we address the problem of better linking requirements analysis to detailed design and implementation in the *Tropos* agent-oriented methodology with the aim to address adaptability issues. In particular, we revisit the definition of agent capability in *Tropos* and refine the development process in order to point out how capability specification can result from the integration of various analysis strategies. We also show how fragments of an implementation can be generated automatically from an agent capability specification.

1 Introduction

Nowadays, distributed information systems need to operate in open, evolving, heterogeneous environments. Trust in these systems by their owners and users entails ever-increasing expectations for robustness, fault tolerance, security, flexibility and adaptability. Multi-Agent Systems (MAS) have been proposed as a suitable conceptual framework for building such information systems [3, 5, 10, 15, 12]. In this framework, an information system is conceived as an open network of software agents who interact with each other and human/organizational agents in their operational environment in order to fulfill stakeholder objectives. Agent-oriented software engineering projects have been developing novel design techniques and support tools for complex information systems [10]. In particular, the *Tropos* methodology [3, 5] captures early requirements through an analysis of stakeholder goals and strategic dependencies among them. System requirements and design is then derived in a systematic way. System design includes both architectural and detailed design, and is followed by system implementation. Our research is conducted within the context of the *Tropos* project.

In this work, we refine the *Tropos* software development methodology proposed elsewhere [3, 5] by focusing on the concept of agent capability. Agent capability has been defined in agent-oriented programming [20] as the ability of an agent to achieve a goal. This definition has been revised in recent work [15] into a refined notion based on the philosophical idea that 'can' implies both *ability* and *opportunity*. This suggest that the lack of either ability or opportunity implies 'cannot'. More specifically, [15] uses the concept of "capability for a given goal" meaning that the agent has at least one plan —the ability— that can fulfill a given goal. This plan constitutes a necessary condition for achieving the goal, while the sufficient condition (i.e. the opportunity) is defined in terms of the pre-conditions or the context that can trigger the plan.

In this paper, we illustrate how the above capability definition can be naturally accommodated with the *goal* and *plan* concepts. More precisely, as detailed in [17], an actor capability is always related to a leaf-goal after goal analysis has been completed. Moreover, we revise the *Tropos* design process in order to make capability modelling and analysis more explicit and systematic. This extension allows us to better exploit information on the environment captured during early and system requirements analysis, while conducting design and implementation of a MAS. Our ultimate objective is to define a systematic process for designing software agents able to adapt and extend their capability at run time, through composition mechanisms analogous to those used in web services [21].

In this work, we adopt Model-Driven Architecture (MDA) guidelines and standards proposed by OMG's [14]. Along with an extended Tropos development process, we are developing specific tools that support the proposed methodology by facilitating the construction, analysis, and transformation of models.

The rest of the paper is structured as follows. Section 2 recalls background notions of *Tropos* and of MDA guidelines and standards. Section 3 introduces an example we use to illustrate our approach. Section 4 introduces our definition of capability and proposes a systematic process for capability design. Section 5 presents a toolset for implementing capability in JADE (Java Agent Development Framework [2]) through an automatic transformation of a platform independent model to a platform specific one, while Section 6 describes capability implementation. Section 7 presents related work and Section 8 offers concluding remarks.

2 Background

We adopt the *Tropos* agent-oriented methodology [3, 5] which rests on a model-driven software development process, i.e. it guides the software engineer in building a conceptual model, which is incrementally refined and extended, from an early requirements model to system design artifacts and then to code. The methodology uses a modelling language based on a multi-agent paradigm named *i** [22], which provides concepts of *actor*, *goal*, *plan*, *softgoal*, *resource* and *capability*. The *i** modelling framework also includes relationships between actors and goals. In addition, the framework provides a graphical notation to depict views of a model, such as *actor diagrams*, which point out dependencies between a set of actors and *goal diagrams*, which depict how actor goals can be decomposed into subgoals[1].

The *Tropos* methodology also includes various analysis techniques which are tool supported[2] and a structured software development process which has been specified in terms of a non-deterministic concurrent algorithm [3]. This process starts with the identification of critical actors ("stakeholders") in a domain along with their goals, and

[1] The *Tropos* modelling activities are supported by the TAOM4E tool [18] (see http://sra.itc.it/tools/taom).

[2] The *Tropos* formal analysis, goal-reasoning and security analysis techniques are supported by the T-Tool, the GR- Tool and ST-Tool respectively (see http://www.troposproject.org).

proceeds with the analysis of goals from the perspective of each actor. In particular, given a goal, the software engineer may decide to *delegate* it to an actor already existing in the domain or to a *new actor*. Such delegations result in a network of dependency relationships among actors. Moreover the software engineer may decide to *analyze* a goal producing a set of subgoals. Goal analysis generates a goal hierarchy where the leaves in various combinations represent concrete solutions to the root goal. Finally the software engineer may decide that a certain actor is able to *satisfy* the goal via a plan the actor is able to execute; in this case the goal is assigned to that actor (with no further delegations). The process is complete when all goals have been dealt with.

This iterative process is organized in four main requirements analysis and design phases, each characterized by specific objectives. In particular, during *Early Requirements* the environment (i.e. the organizational setting) is modelled and analyzed; during *Late Requirements*, the system-to-be is introduced and its role within the environment is modelled; during *Architectural Design* the system architecture is specified in terms of a set of interacting software agents; *Detailed Design* is concerned with the specification of software agent capabilities and interactions and provides the input to code generation.

In refining the modelling process algorithm and building tools which can support it [19], we adopted ideas and standards from MDA. MDA conceives system development in terms of a chain of model transformations, namely, from a domain model (Computationally Independent Model — CIM) to a Platform Independent Model (PIM), and from a PIM to a Platform Specific Model (PSM), from which code and other development artifacts can then be straightforwardly derived. In this paper we focus on how to build a PIM for a MAS generic platform in *Tropos* and on how to automatically transform it into a PSM; we consider here the JADE programming platform [2] metamodel. Our framework is compliant with the MDA metamodelling standard called Meta Object Facility (MOF) [13], which defines a set of modelling constructs supporting metamodelling. Moreover, we exploit a Frame Logic-based approach described in [6] to deal with metamodel to metamodel transformations.

3 Example

An example is taken from the on-line selling shop application[3] domain. According to *Tropos*, the early requirements analysis has to define the social actors and their intentions in terms of social dependencies, commitments, and responsibilities among stakeholders. While, the late requirements phase introduces the system-to-be —e.g. the actor Retailer System— relating it to the other stakeholders of the domain. In this case, as depicted by the Fig. 1, some of the Customer needs are delegated to the Retailer System —i.e. softgoals search for the desired product automatically, flexible and automatic payment, and fast delivery— in order to correctly design the system functionalities. For the sake of simplicity, the figure does not depict all the elements a *why* dependency is composed of. Each time a Customer asks for product details, Retailer System can fulfill such a request by achieving the goal provide product info. As illustrated by the goal diagram of Fig. 1, there are two possible (means-ends) alternatives

[3] The scenario used is an idealization, intended solely for illustration purposes.

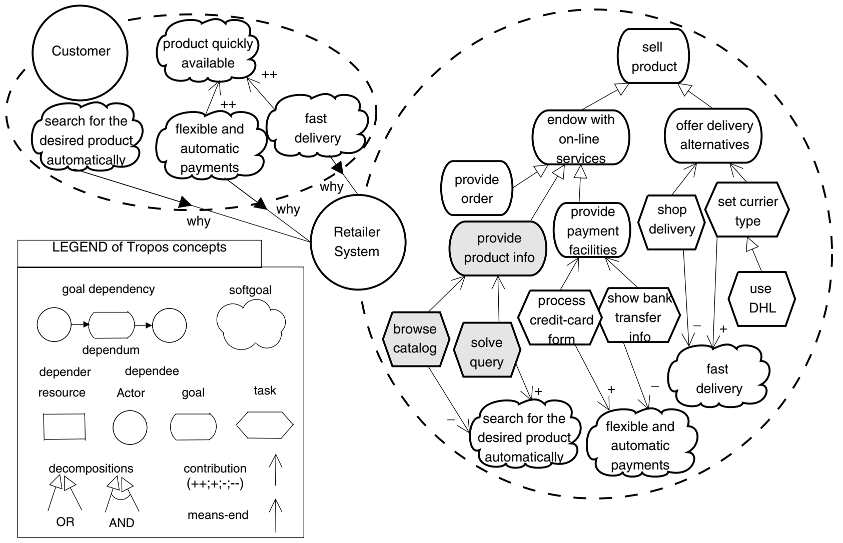


Fig. 1. Late Requirements: Fragment of Retailer System Goal Analysis

to deals with such a goal: browse catalog and solve query. Moreover, in our case, the goal diagram models a class of Customers that need to search for product details as much as possible automatically, i.e. by the softgoal search for the desired product automatically. This non-functional requirement can be used by the Retailer System to drive the plan selection according to the contribution link analysis. Specifically, as illustrated by Fig. 1, the plan solve query gives a positive contribution (+), while browse catalog gives a negative contribution (-) to this softgoal.

Given a characterization of the system-to-be inside its operating environment, e.g. Fig. 1, the methodology allows the designers soft attention to Architectural and Detailed Design. In our case, Fig. 2.(A) depicts a fragment for the Architectural Design phase where the main system components have been identified, i.e., Web Server, Order Manager, and Search Manager. The fulfillment of some of the previous functional and non-functional requirements have been delegated to such actors (hereafter agents). For example, as depicted in Fig. 2.(A), the goal provide product info has to be fulfilled by the agent Search Manager. Again, the plan process credit-card form has been delegated to the external actor Credit Authority. Therefore, architectural design results in a multi-agent system consisting of agents, dependencies among them, as well as environmental constraints these agents have to cope with.

The Detailed Design deals with specification details for each agent, showing and describing how an agent concretely behaves in order to execute a plan or to satisfy a goal. For example, as illustrated by Fig. 2.(B), in order to fulfill the plan solve query, the agent Search Manager relies on three sub-plans interpret ACL performatives, deal with cooperation, and provide results. For the sake of simplicity, only the plan deal with cooperation has been further detailed in three atomic sub-plans search for new acquaintances, get the query, and deal with matching. In our case, to get the query, this agent depends on the actor Web Server responsible for interfacing the

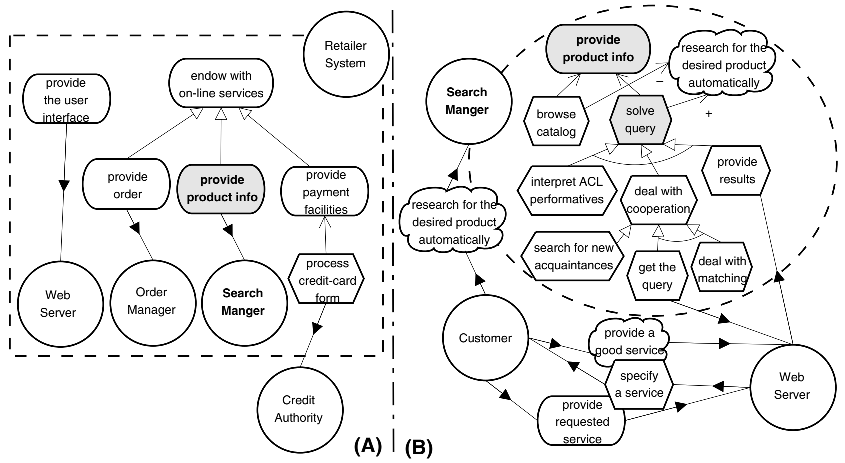


Fig. 2. (A) Architectural Design: fragment of the Retailer System sub-actors definition; **(B)** Detailed Design: fragment of the Goal Analysis for the agent Search Manager

system-to-be with external users or other agents (Customer). Therefore, Customer, who models both human users and software agents, depends on the system through the component Web Server for the goal provide requested service. Moreover, softgoals may model stakeholders needs that may be only achieved by means of specific system actor capabilities. Hence, also softgoals (Fig. 1) have to be delegated to specific system components, e.g. Customer depends on Search Manager to satisfy search for the desired product automatically.

To effectively deal with the agent behavior at run-time, the methodology has also to address the dynamic aspects that affect agent activities. This important phase is discussed in detail in the next sections.

4 Capability Design

While we adopt the *Tropos* definition of capability, we propose to extend the way to specify it during design by explicitly describing not only the dynamic part, but also its descriptive and context part. For this, we revise the *Tropos* definition of capability to include both *ability* and *opportunity*, as detailed in [17]. In particular, the *ability* part is described via the *Tropos* means_end relationship between a goal and a plan, while the *opportunity* is described in *Tropos* via plan/softgoal contributions, $\langle plan, softgoal, metric\rangle$ ($metric \in \{+, -, ++, --\}$) and environmental constraints (e.g. temporal constraints between sub-plans) that are specified by model annotations. More formally, the definition of capability is given in terms of a set of basic building blocks that a designer can use to represent its several aspects, namely:

$$Cap = \langle means\_end(goal, plan), \cup_i contribution(plan, softgoal_i, metric),$$
$$\{A_1, \ldots, A_n\}\rangle$$

where $contribution(plan, softgoal_i, metric)$ is the set of contribution relationships of the plan $plan$ to the softgoals $softgoal_i$ —according to a specific metric $metric$— and $\{A_1, \ldots, A_n\}$ is a set of model annotations that describe domain constraints. Our work adopts the Formal *Tropos* language [8], a first order temporal logic language, to specify constraints on the model elements[4]. The annotations contain also information that concern dynamic aspects of a capability. In our approach, these dynamic aspects are modelled by AUML activity and interaction diagrams, see Fig. 3. In this approach, each root-level plan may be described by an activity diagram, where leaf-level plans are modelled as activities. Indeed, the Tropos AND/OR decomposition comes out with leaf-level plans suitable to model atomic agent actions.

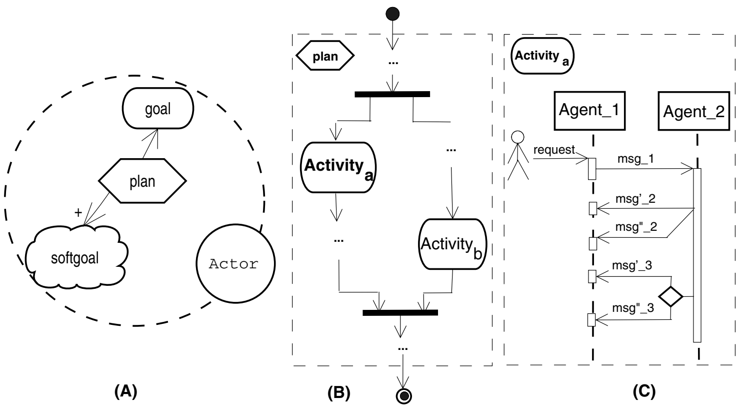


Fig. 3. Focusing on the dynamic dimensions of a capability: (A) actor's view in *Tropos* notation, (B) control-flow structure of the ability $plan$, and (C) agent-interaction for a part of the ability fulfillment, i.e. activity $Activity_a$

Fig. 3.(A) shows the elements and the relationships that characterize a capability from early requirements to detailed design. Fig. 3.(B) describes the activity diagram for the *plan*, while Fig. 3.(C) gives the representation of agent interactions for just one of the activities of the *plan* —$Activity_a$. We can find this pattern in the example shown in Fig. 1; focusing on the goal provide product info the *ability* part of the capability is given by the means_ends relationship with the plan solve query. The ability part only gives a partial description of the capability, that is, it does not provide any information on the influence of the environment on the behavior of the system at run-time. The second part of the definition describes the *opportunity* for the capability, can be given via "softgoal contributions", e.g related to the softgoal search for the desired product automatically that our customer requires. As an example, in Table 1, for Cap_2,

[4] We are exploring the possibility to integrate different approaches dealing with the agent intelligence representation, such as, declarative annotations in OWL-S [21, 16], or belief-desire-intention concepts for agent behavior characterization [4, 15]. Therefore, the annotations $A_1, ..., A_n$, could represent further details of a capability expressed in one of several specification languages.

we specify an annotation named A_1 that is associated with a BDI based semantics as detailed in the next Section.

In order to support the capability design activity, we added a new step to the *Tropos* algorithm described in [3]. The details of a revised version of the proposed algorithm for the Tropos design process can be found in [17]. In particular, during the initialization of the design process, the set of stakeholders and goals is added to the model; goals are then assigned to actors, and therefore become the root goals for those actors. During the analysis, for every goal in the model, a *goal_Analysis* step is carried out, in order to delegate the goal, expand it into subgoals or operationalize each goal, associating it one or more plans, thereby discovering a required capability for the system. According to this strategy, given a certain goal, the *capability_modelling* procedure proposes plans that can fulfill the goal and adds to the current model a means_ends relationship for every discovered goal/plan pair. This pair constitutes the first part of the definition of a capability. For every discovered means_ends links the algorithm collects the set of "softgoals contribution" relationships related to the plan involved in the contribution and discovered during the modelling process. These contributions represent conditions from the domain for that capability. The set of annotations —such as softgoals that model environmental constraints, related to the goals/plans involved in the capability— feeds the "annotation" part of the capability definition, e.g. A_1 for Cap_2. The capability discovery and specification process can be iterated during the whole modelling activity in order to capture new capabilities or new components of capabilities that have been already specified and that gradually emerge during the analysis. The output of the analysis process is a set of capabilities related to a given goal that in our case is partially illustrated in Table 1.

Table 1. Capabilities at Architectural Design phase

Agent	Capabilities	Means_End(goal,plan)	List of Contributions	Annotations
$SearchManager$	Cap_1	provide product Info, browse catalogue	{search the desired prod. autom. -}	...
	Cap_2	provide product Info, solve query	{search the desired prod. autom. +}	A_1
$CreditAuthority$	Cap_3	provide payment facilities, process credit card	{flexib. and autom. pay. +}	...
	Cap_4	provide payment facilities, show bank transfer info	{flexib. and autom. pay. -}	...
$OrderManager$	Cap_5	provide order, manage order form	{null}	...
...	...	...,...	{...}	...

Focusing on Ability. Taking advantage from the previous capability modelling phase, here we illustrate how the methodology effectively deals with ability aspects. Fig. 4 addresses the dynamic aspects of the capability Cap_2. To effectively deal with such aspects, we consider two dimensions: *(i)* the control-flow structure of the activities that the capability is composed of, and *(ii)* for each activity and for each agent interaction required by its execution, the required interaction protocols. We propose to use AUML activity diagrams for *(i)* —e.g. as illustred in Fig. 4.(A)— and AUML interaction diagrams for *(ii)* —e.g. as illustrated in Fig. 4.(B). In the example, the control-flow for the ability part of Cap_2 is composed of 4 activities —i.e. Fig. 4.(A)— with the following labels and meanings:

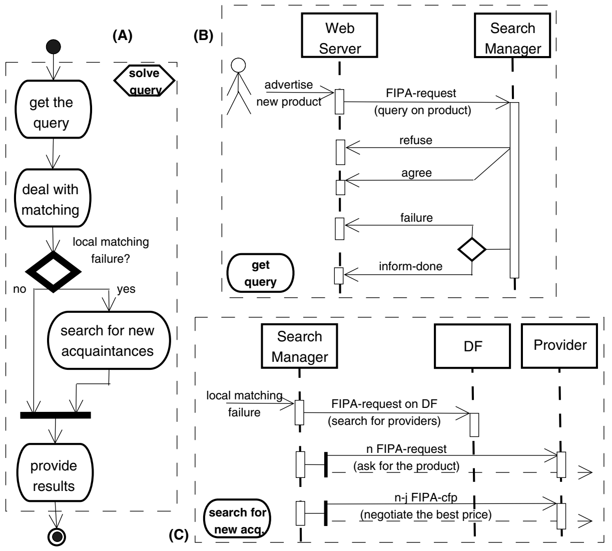


Fig. 4. A fragment of the two dimensions for the capability Cap_2 (Table 1) detailed-design: (A) the control-flow structure and (B) the agent interactions

– get the query. As further detailed by the interaction diagram of Fig. 4.(B), Search Manager waits for the web-user's request that carries out partial information on the product it is interested in, i.e. such an agent plays the *responder* role within a FIPA-request. The message content carries out the information on the product to look for, structured as follows: <product-category=..>, <product-name=..>, <product-quantity=..>, <product-price-range=(*min*,*max*)>.
– deal with matching. Once the product specifications have been correctly interpreted the Search Manager checks such a product in its local repository. Notice that, this phase does not require any external interaction, but the repository is inquired by Search Manager using a self FIPA-request IP.
– search for new acquaintances. This activity is performed when a failure occurs during the local matching phase (deal with matching), e.g., the local stock quantity is not sufficient for the required quantity, the current price does not fit the range, the product does not exist, etc. Therefore, in order to overcome such failures, the Search Manager cooperates with other providers, i.e. distributed warehouses. This capability activity is the most complicated as depicted by Fig. 4.(C), indeed, it is composed of three IPs: i) a FIPA-request in order to ask the Directory Facilitator (DF) for the providers; ii) n FIPA-requests targeted to all the providers returned by the DF in order to check the product availability in their warehouses; iii) a FIPA-cfp in order to negotiate the best price with the n-j providers figured out at the previous step.
– provide results. At the end, the agent communicates the obtained results by a simple inform message that can be of two types: a failure description (no results) or a list of retrieved products along with their detailed descriptions.

5 From Platform-Independent to Platform-Specific Model

In this section we present a technique to automatically derive the description of the capability of a JADE agent, i.e. a PSM, from the PIM model. In particular, we show how it is possible to map AUML Activity and Interaction Diagrams to JADE structures, using transformation techniques compliant with MDA's Query/View/Transformation requirements, that have been introduced in [9]. We exploit a Frame Logics based approach to model transformation described in [6] and implemented in the Tefkat tool[5]. The language consists of three major concepts: *pattern definitions*, *transformation rules*, *tracking relationships*. *Pattern definitions* are generated in order to identify structures

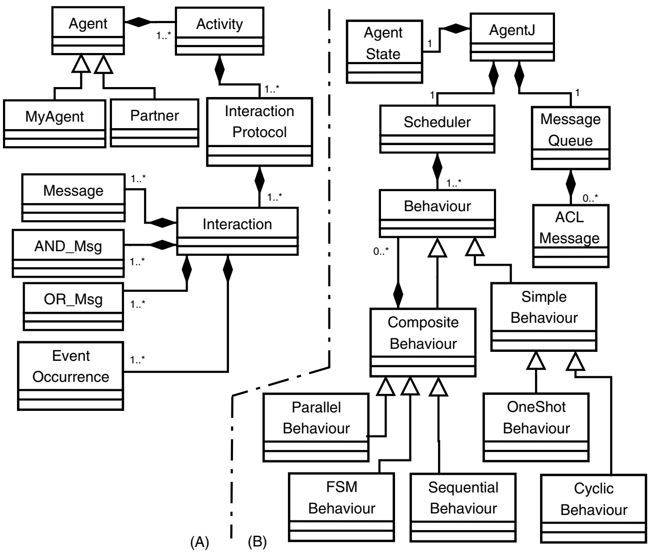


Fig. 5. **(A)** Fragment of the Interaction Diagram metamodel (part of the PIM model); **(B)** Fragment of the JADE metamodel (the PSM)

that are frequently used in a given transformation. *Transformation rules* map source to target metamodels constructs. *Tracking relationships* allow to maintain the traceability between entities in source and target model instances. The syntax of rules uses clauses such as the *Forall* and *Where* to recognize elements of the instance of the source model, and *Make* and *Set* for building the instance of the target model.

In the following we give an example of the mapping of AUML Interaction Diagrams to a subset of the JADE platform concepts based to the example described in the previous sections. The transformation is based on the metamodels of the two languages. Fig. 5(A) shows a subset of the AUML Interaction Diagram metamodel as described in [1]. From one side the agents involved in the interaction represented by the class Agent, on the other the interaction protocol constituted by a set of interactions (represented by the class Interaction and Interaction Protocols) made of simple messages or more complex structures like the And, Or and Xor composition of messages.

[5] More details are available in http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

A subset of the target metamodel is shown in Fig. 5(B). Here an agent is described as an aggregation of Message Queue, Agent States and Scheduler behaviors that is an aggregation of behaviors. A behavior can be simple or composite, allowing to specify a composition of several behaviors. Fig. 6 illustrates an excerpt of the mapping rules

```
TRANSFORMATION Interaction2JADE: auml → Jade
RULE Agent2Agent()
  FORALL MyAgent mya
  MAKE AgentJ a
  SET a.name = mya.name, a.role = mya.role;
  LINKING AgentForAgent WITH agent = a, myagent = mya
RULE InteractionProtocol2Behaviour()
  FORALL MyAgent a1, MyAgent a2, InteractionProtocol ip
  WHERE ip.send = a1.name
     AND ip.rec = a2.name
     OR ip.rec = a1.name
     AND AgentForAgent LINKS myagent = a1, agent=ag1
     AND AgentForAgent LINKS myagent = a2, agent=ag2
  MAKE Behaviour b1
  SET b1.type = ip.type, b1.sender = ip.send, b1.receiver = ip.rec, b1.counter = ip.counter, b1.AgentJ = ag1
     ......
```

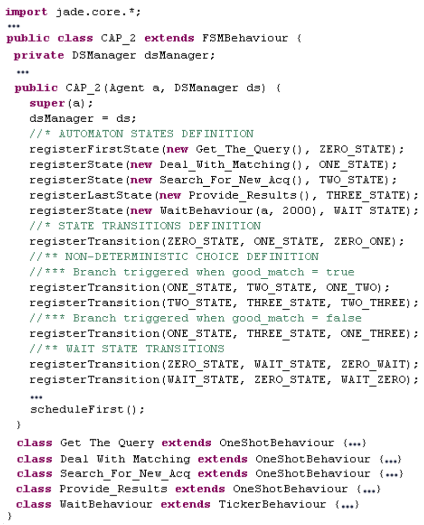
Fig. 6. An excerpt of the transformation from Interaction Diagram metamodel to JADE metamodel defined in the grammar described in [6]

from the Interaction Diagram metamodel to JADE platform metamodel. They are specified in terms of a subset of the grammar described in [6]. The RULE *Agent2Agent* allows for the transformation of the set of agents of the Interaction diagrams into a set of agents (AgentJ) in the target JADE model. The rule is composed by clauses: the clauses FORALL and WHERE retrieve the set of agents in the source metamodel; the clauses MAKE and SET are in charge to build the set of Agents in the target platform, simply creating a new agent for every retrieved agent in the source model. The RULE *InteractionProtocol2Behaviour* refers to the mapping of the Interaction Diagrams elements, and in particular of the messages exchanged by agents in a given protocol, into the definition of a JADE agent behavior. The target structure is created via the MAKE directive and instantiated via the SET directive in the rule; part the resulting XMI file is shown below.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
     xmlns:jade_conc="http:///jade_conc.ecore">
  <jade_conc:AgentJ xmi:id="26990772" name="Search_Manager" role="Serch_Manager"/>
  <jade_conc:AgentJ xmi:id="11552137" name="Provider" role="Provider"/>
  <jade_conc:Behaviour xmi:id="10834914" type="req"
     sender="Search_Manager" receiver="Provider" counter="1"/>
  <jade_conc:Behaviour xmi:id="2511137" type="cfp"
     sender="Search_Manager" receiver="Provider" counter="2"/>
...
</xmi:XMI>
```

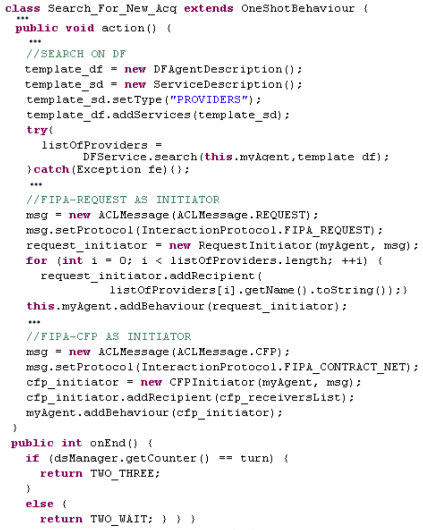
6 Ability and Opportunity Implementation

Let's consider one of the examples of Table 1. Cap_2 is composed of an ability part — i.e. the plan solve query— and an opportunity part —i.e. the softgoal research for the desired product automatically. Since the opportunity aspect is related to the intelligent part of an agent behavior, it naturally fits into a Belief-Desire-Intention (BDI) agent architecture, having adopted the Jadex plug-in for JADE. Although our current prototype only supports the automatic transformation (from detailed design to implementation) for the ability part, we have manually generated the opportunity part as Jadex-based precondition. In this case, the goal provide product info has been assigned specific trigger-messages (FIPA-request) that the agent can satisfy adopting one of the abilities already specified at design-time in Fig. 2.(B) —i.e. plans browse catalog and solve query. Notice that, these trigger-messages may also carry out information about the user's profile —i.e. as a precondition— that enables or disables the achievement of specific softgoals. Specifically, such a precondition for Cap_2 has been annotated in A_1 at design-time, as depicted in Table 1. Therefore, each time a user logs in, the system classifies her/him into a predefined category that also enables or disables ($true$ or $false$) the activation of specific preconditions. For example, if the softgoal research for the desired product automatically is enabled, at the time the agent has to achieve the goal provide product info, the ability solve query will receive a higher selection-priority in respect to browse catalog.



```
import jade.core.*;
...
public class CAP_2 extends FSMBehaviour {
 private DSManager dsManager;
 ...
 public CAP_2(Agent a, DSManager ds) {
   super(a);
   dsManager = ds;
   //* AUTOMATON STATES DEFINITION
   registerFirstState(new Get_The_Query(), ZERO_STATE);
   registerState(new Deal_With_Matching(), ONE_STATE);
   registerState(new Search_For_New_Acq(), TWO_STATE);
   registerLastState(new Provide_Results(), THREE_STATE);
   registerState(new WaitBehaviour(a, 2000), WAIT STATE);
   //* STATE TRANSITIONS DEFINITION
   registerTransition(ZERO_STATE, ONE_STATE, ZERO_ONE);
   //** NON-DETERMINISTIC CHOICE DEFINITION
   //*** Branch triggered when good_match = true
   registerTransition(ONE_STATE, TWO_STATE, ONE_TWO);
   registerTransition(TWO_STATE, THREE_STATE, TWO_THREE);
   //*** Branch triggered when good_match = false
   registerTransition(ONE_STATE, THREE_STATE, ONE_THREE);
   //** WAIT STATE TRANSITIONS
   registerTransition(ZERO_STATE, WAIT_STATE, ZERO_WAIT);
   registerTransition(WAIT_STATE, ZERO_STATE, WAIT_ZERO);
   ...
   scheduleFirst();
 }
 class Get The Query extends OneShotBehaviour {...}
 class Deal With Matching extends OneShotBehaviour {...}
 class Search_For_New_Acq extends OneShotBehaviour {...}
 class Provide_Results extends OneShotBehaviour {...}
 class WaitBehaviour extends TickerBehaviour {...}
}
```

(A)



```
class Search_For_New_Acq extends OneShotBehaviour {
 ...
 public void action() {
   ...
   //SEARCH ON DF
   template_df = new DFAgentDescription();
   template_sd = new ServiceDescription();
   template_sd.setType("PROVIDERS");
   template_df.addServices(template_sd);
   try{
     listOfProviders =
           DFService.search(this.myAgent,template df);
   }catch(Exception fe){};
   ...
   //FIPA-REQUEST AS INITIATOR
   msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   request_initiator = new RequestInitiator(myAgent, msg);
   for (int i = 0; i < listOfProviders.length; ++i) {
     request_initiator.addRecipient(
               listOfProviders[i].getName().toString());}
   this.myAgent.addBehaviour(request_initiator);
   ...
   //FIPA-CFP AS INITIATOR
   msg = new ACLMessage(ACLMessage.CFP);
   msg.setProtocol(InteractionProtocol.FIPA_CONTRACT_NET);
   cfp_initiator = new CFPInitiator(myAgent, msg);
   cfp_initiator.addRecipient(cfp_receiversList);
   myAgent.addBehaviour(cfp_initiator);
 }
 public int onEnd() {
   if (dsManager.getCounter() == turn) {
     return TWO_THREE;
   }
   else {
     return TWO_WAIT; } } }
```

(B)

Fig. 7. A fragment of the Cap_2 implementation: (A) control-flow as a JADE-based automaton implementation, (B) agent interactions as JADE-based FIPA IPs

The implementation of the ability part of a capability results from a transformation process, previously explained, in terms of a set of Tefkat xmi files, one for each activity of the AUML activity diagram. Consequently, as shown in Fig. 4 Cap_2 is composed of 4 transformation output files that are read and interpreted in order to generate the real Java code, i.e. our agent template for the agent Search Manager. Iteratively the same process applied to the whole table 1 produces the MAS previously designed.

The tool component in charge of performing the last development phase —i.e. code generation— uses very simple rules[6] in order to map the semi-structured information specified by the xmi output files into a JADE-based agent framework. Such rules drive the mapping between the diagrammatic concepts and a flexible agent framework that allows an agent to play different roles along with different capabilities according to specific environmental conditions —i.e. trigger messages that represent stakeholder intentions. The principal rules that have been adopted for our agent framework are the following:

1. Each agent *extends* the class *jade.core.Agent* according to special trigger-messages —i.e. target goals. That is, an agent can sense the environment and consequently switch to a specific role, hence it plays a precise capability. To deliver on such an aim, each agent owns a table that relates each capability to a set of trigger-messages and viceversa.
2. Each capability *extends* the class *jade.core.behaviours.FSMBehaviour*, namely it represents a final states machine (automaton). Thanks to such an implementing choice, each single activity of an activity diagram —i.e. an atomic task of the capability— corresponds to a single state of the automaton.
3. Each state is monitored —in terms of messages exchanged— in order to make the agent aware about the next state-transition. Thanks to such a feature, the agent can handle non-deterministic events at the moment they occur. Moreover, each time a failure occurs, such a strong property may allow the agent to switch in a compensation state[7].

An excerpt of the ultimate Cap_2 development phase is given in Fig. 7. In particular, Fig. 7.(A) shows how is defined in JADE the automaton associated to Cap_2: a states assignment step, e.g. *registerState(**new** Deal_With_Matching(),ONE_STATE)* and a state transitions step, e.g. *registerTransition(ONE_STATE,TWO_STATE,ONE_TWO)*. Notice that, each single activity of the activity diagram (Fig. 4.(A)) has been mapped in a JADE *jade.core.behaviours.FSMBehaviour* state, while inside each state a FIPA-IP has been mapped in an equivalent JADE FIPA-IP, as detailed by Fig. 4.(B). As illustrated in Fig. 4.(B), the framework allows the agent to monitor the state termination, namely, each IP (JADE behavior) saves its information on a *jade.core.behaviours.DataStore* class that is periodically checked (i.e. by our *dsManager*). This framework property allows the agent to monitor its internal behaviors and to pro-actively react against internal failures.

7 Related Work

There are, two types of research that are relevant to our work, namely research on agent capability and on AOSE methodologies covering agent implementation issues. Along the first line we mention the proposal given in [15], which defines a possible formal

[6] Related to the target agent framework building. Moreover, we are still investigating how many of these rules can be delegated to the Tefkat engine.

[7] This issue is not within the scope of this paper. However, we are actively investigating it.

relationship between capabilities and BDI concepts —i.e. beliefs, goals and intentions. This work roots the concept of capability into the philosophical idea that ‘can’ implies both *ability* and *opportunity*.

As detailed in this paper, our approach adopts the concept of capability as composed of ability and opportunity. Moreover, our approach extends previous capability formalizations principally in two directions. The first one is that it considers the possibility to have an agent ability —i.e. a plan— decomposed in sub-plans that can also be delegated to other agents. While, the second extension takes into account the possibility to have an opportunity, related to a given agent ability, composed of different opportunities that come from other agent perspectives. Moreover by means of our methodology the designer can trace the capability environmental constraints arisen in the early phases down to detailed design and implementation.

An approach which attempts to link the operative part of the capability —i.e. a set of actions embedded in the *behavior* concept— with the intelligence of an agent — i.e. in terms of beliefs and intentions, is proposed in [4]. More precisely, the authors propose an agent-oriented approach to software engineering called Behavior Oriented Design (BOD). By means of their BOD methodology, a complex problem can be decomposed in simple and independently modules that contain the agent actions. In particular, they consider an agent characterized principally with: *goals* —i.e. conditions to be achieved—, *intentions* —i.e. goals and subgoals that are currently chasing—, *beliefs* —i.e. the knowledge basis as partial view of the world—, and the *behaviors* —i.e. set of actions it can take. Thanks to such an approach, each agent can be characterized by behavioral modules. Even if our capability definition seems similar to this behavior concept, the proposed framework is less flexible than our in detailing the single components. In particular, the above mentioned approach, considers the modules as predefined rigid blocks of actions related to specific agent goals and beliefs. On the contrary, our approach also describes how atomic actions (sub-plans) contribute to the stakeholders intentions and beliefs achievements, i.e. by the softgoal contribution link analysis technique.

Along the second line, [11] goes in the direction of adding flexibility to agents and proposes a component based framework that facilitates the domain experts themselves making modification of deployed multi-agent systems with the aim of increasing the capacity of the systems to fit the evolving needs identified in the domain. In particular the framework is based on agent systems composed by well defined components and gives a structured support to the user for modifying or composing existing components, or adding new components in well defined ways; this mechanism, for example, intends to help the experts in specifying new goals and plans for the agents starting from the adaptation of the components that describe the existing one.

Among AOSE methodologies that describe and cover the agent implementation phase, Passi seems to be one of the most flexible and documented [7]. In the Passi methodology the process that guides the agent-based code generation is quite similar to our approach. For example, such methodology adopts activity diagrams to specify agent behaviors (i.e. *Multi-Agent Behaviour Description*), and it characterizes an agent role in terms of its tasks, e.g. see Chapter IV of [10] for details. The main differences with our approach are the followings. While Passi aims to model an agent role in terms of its tasks (i.e. the behavior), we model capabilities in terms of interaction protocols and internal tasks. In this way, the role is only a logical concept that arises when the agent plays a specific set of capabilities. Hence, our agent may play several roles, namely an agent behavior may be composed of several capabilities (composition). Passi does not consider stakeholder intentions and social dependencies —e.g. as illustrated in this paper by the *Tropos* softgoals— as strategic knowledge elements that the agent requires to effectively deal with capability selection. On the contrary, by means of the opportunity concept modelling, we are able to embed in the agent knowledge also environmental constraints figured out at the early phases of the requirements analysis. Notice that, such requirements cannot (easily) emerge by only considering the MAS architectural level.

8 Conclusions and Future Work

This paper focuses on design issues for agent oriented software development, such as requirements traceability and automated code generation. In particular, we revise the *Tropos* capability definition to better trace early and late requirements —e.g. stakeholder intentions and domain constraints— till down the MAS detailed design and implementation phases. Specifically, we have illustrated through examples —supported by prototype tools— that a MDA approach can cope with the automatic mapping between a platform-independent agent-based conceptual model (Tropos) and a platform-specific agent-based model (JADE). Whenever possible, our approach is based on current standards, namely, OMG's MDA for model transformations, IEEE's FIPA for an agent architecture and interaction protocols, and AUML for activity and interaction diagrams. As future work, we propose to deal with monitoring and compensation during capabilities execution to validate the system behavior with respect to design-time requirements. Further validation on real case studies will also be performed.

Acknowledgments

We would like to thank Barbara Tomasi, Loris Delpero, and Alessandro Orler for their precious contribution to accomplish the experiments.

References

1. B. Bauer, J. P. Muller, and J. Odell. Agent uml: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):1–24, 2001.
2. F. Bellifemine, A. Poggi, and G. Rimassa. JADE: A FIPA Compliant agent framework. In *Practical Applications of Intelligent Agents and Multi-Agents*, pages 97–108, April, 1999.
3. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, July 2004.
4. J. Bryson and S. McIlraith. Toward behavioral intelligence in the semantic web. *IEEE Computer - Web Intelligence*, 35(11):48–54, 2002.

5. J. Castro, M. Kolp, and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*. Elsevier, Amsterdam, the Netherlands.
6. CBOP, DSTC, and IBM. MOF Query/Views/Transformations, 2nd Revised Submission. Technical report, 2004.
7. M. Cossentino. *From requirements to code with the PASSI methodology*. Chapter 4, In [10], 2005.
8. A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *IEEE Int. Symposium on Requirements Engineering*, pages 174–181, Toronto (CA), Aug. 2001. IEEE Computer Society.
9. T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review of omg mof 2.0 query / views / transformations submissions and recommendations towards the final standard. In *MetaModelling for MDA Workshop*, York, England, 2003.
10. B. Handerson-Seller and P. Giorgini. *Agent-Oriented Metodologies*. Idea Group, 2005.
11. G. Jayatilleke, L. Padgham, and M. Winikoff. A Model Driven Component-Based Development Framework for Agents. *Computer Systems Science & Engineering*, 4(20), 2005.
12. N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
13. S. R. Judson, R. B. France, and D. L. Carver. Specifying Model Transformations at the Metamodel Level, 2004. http://www.omg.org.
14. S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled*. Addison-Wesley, 2004.
15. L. Padgham and P. Lambrix. Formalizations of Capabilities for Bdi-Agents. *Autonomous Agents and Multi-Agent Systems*, 10:249–271, 2005.
16. L. Penserini and J. Mylopoulos. Design Matters for Semantic Web Services. Technical Report T05-04-03, ITC-irst, April 2005.
17. L. Penserini, A. Perini, A. Susi, and J. Mylopoulos. From Stakeholder Intentions to Agent Capabilities. Technical report, ITC-irst, Trento, Italy, October 2005.
18. A. Perini and A. Susi. Developing Tools for Agent-Oriented Visual Modeling. In G. Lindemann, J. Denzinger, I. Timm, and R. Unland, editors, *Multiagent System Technologies, Proc. of the Second German Conference, MATES 2004*, number 3187 in LNAI, pages 169–182. Springer-Verlag, 2004.
19. A. Perini and A. Susi. Agent-Oriented Visual Modeling and Model Validation for Engineering Distributed Systems. *Computer Systems Science & Engineering*, 20(4):319–329, 2005.
20. Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60:51 – 92, 1993.
21. K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, pages 27–46, 2003.
22. E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, University of Toronto, 1995.