

Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA*

Mercedes Amor, Lidia Fuentes and Antonio Vallecillo

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga. Campus de Teatinos, s/n.
29071 Málaga, Spain
{pinilla, lff, av}@lcc.uma.es

Abstract. Current agent-oriented methodologies focus mainly on multi-agent systems analysis and design, but without providing straightforward connections to the implementation of such systems on any of the existing agent platforms (e.g. FIPA-OS, Jade, or Zeus), or just forcing the use of specific agent platforms. In this paper we show how the Model Driven Architecture (MDA) can be effectively used to derive agent implementations from agent-oriented designs, independently from both the methodology used and the concrete agent platform selected. Furthermore, this transformation process can be defined in a scalable way, and partly automated thanks to the use of a platform-neutral agent model, called *Malaca*.

1 Introduction

Software agents are becoming a widely used alternative for building open and distributed applications, developed as Multi-Agent Systems (MAS). This recognition has led to consider agent technology as a promising paradigm for software development [1]. As a result, several agent-oriented methodologies for developing MAS have been recently proposed [2], with the aim to provide tools, practical methods, and techniques for developing MAS.

The variety of methodologies may become a problem for the software developer when it comes to select the best-suited methodology for a given application domain. Selection criteria may include aspects such as the effort required to learn and to use, completeness, documentation, and suitability. Recent works (e.g., [3, 4, 5]) provide comparison studies between the different agent-oriented methodologies, showing the weaknesses and strengths of each one, with the aim to help the software engineer select the most suitable methodology in each case. The results clearly show that there is not a single unified and unique general-purpose methodology. FIPA [6] and OMG [7] have also created some technical committees focused on the identification of a general methodology for the analysis and design of agent-oriented systems, embracing current agent-oriented methodologies such as GAIA [8], MaSE [9], Tropos [10, 11] or MESSAGE [12]. The idea is to identify the best development process for

*This research was funded in part by the Spanish MCYT under grant TIC: 2002-04309-C02-02.

specific MAS. This work is being complemented with the definition of an agent-based unified modelling language (FIPA AUML work plan [13]).

One of the main problems of these methodologies is that they cover the analysis and design phases of MAS development, but do not address the implementation phase, i.e., they do not completely resolve how to achieve the model derivation from the system design to a concrete implementation [8, 9, 12]. Thus, the software engineer is forced to either somehow select one of the existing agent platforms for implementing the agent-oriented design, or to use a concrete agent platform because it is the only one supported by the design methodology (as in the case of Tropos) — which demands specialized skills from the developer. In the former case, the problem is that the criteria usually considered for choosing an agent platform are mainly based on the developer expertise, the programming language, available tools, or its recognition in the agent community. However, these criteria do not take into account the methodology used to design the MAS, for instance.

Besides, this transformation process —which is not a trivial task, and could be, in some cases, quite complex to achieve— has to be defined in an ad hoc manner in each case, for each methodology and for each agent platform. Thus, every single developer has to define and implement the mappings and transformations from the design produced by the selected agent-oriented methodology, to the API's provided by the particular agent platform, without any guidance or help.

Our goal in this paper is to study how such gap can be bridged, thus covering the complete life cycle of MAS. Furthermore, we also analyze how much of this process can be automated, independently from the original methodology used to analyze and design the MAS, and from the agent platform selected to implement the system.

In order to achieve such goal we will use the concepts provided by OMG's Model Driven Architecture (MDA). MDA is a modelling initiative that tries to cover the complete life cycle of software systems, allowing the definition of machine-readable application and data models, which permit long-term flexibility of implementation, integration, maintenance, testability and simulation [14]. MDA defines platform-independent models (PIM), platform-specific models (PSM), and transformations between them.

In this paper we will show how our problem can be naturally expressed in terms of the MDA, and then how the MDA mechanisms can be used for defining (and partially automating) the mappings. By applying the MDA ideas, the design model obtained as the result of applying an agent-oriented methodology can be considered as a PIM, the target agent platform for the MAS as the PSM, and the mappings between the two can be given by the transformations defined for the particular agent platform selected. The target models needs to be expressed in terms of their corresponding UML profiles, as indicated by the MDA.

However, when we initially tried to use this approach, we saw that it could work for some individual cases, but that it did not scale well for the ever-increasing number of agent-oriented methodologies and agent platforms: somebody had to define and automate the mappings between every agent-oriented methodology and every agent platform. This is not affordable at all. However, we then discovered that the use of a platform-neutral agent architecture could greatly simplify this task, since the numbers of mappings was significantly reduced. Thus, we propose the use of the agent model [15, 16] *Malaca*). One of the most outstanding features of this agent architecture is

that it is possible to execute a Malaca agent on top of any FIPA-compliant agent platforms, i.e., it was designed to be independent from the underlying agent platform. Therefore we define mappings from agent-oriented methodologies to Malaca, and from Malaca to the different agent platforms. Consequently, with just one transformation between an agent-oriented methodology and the Malaca agent model, the resulting MAS could run in any agent platform. The developer could then deploy the MAS in any agent platform, depending on the availability, price, tools provided, etc.

Moreover, as we will show later, the mappings and transformations from agent-oriented methodologies such as Tropos to Malaca are very simple, and most of them can be easily automated due to the use of UML and AUML, a big step towards bridging the gap between agent-oriented design and MAS implementation. A significant feature of this architecture is that any Malaca agent can be “programmed” simply by editing XML documents. Since most of the UML diagrams can be expressed easily in XML, the transformations between agent-oriented methodologies and Malaca are direct. This architecture reduces the development time, cost and effort, and simplifies the implementation of multi-agent systems.

In order to validate our proposal, mappings from one of the most representative agent-oriented methodologies to Malaca have been defined, namely from Tropos. They will be used throughout the paper for illustrating our approach.

The structure of this paper is as follows. Section 2 provides a brief overview of Tropos methodology. Section 3 introduces the Malaca agent architecture, and its underlying agent model. Section 4 describes our main contribution, by showing how to use the MDA approach for mapping MAS designs into the Malaca model, which can be then run into any agent platform. Section 5 outlines some of the problems and limitations of our approach, as well as further research work that could help address such limitations. Finally, Section 6 draws some conclusions.

2 Agent Methodologies Overview

Agent-Oriented methodologies provide a set of mechanisms and models for developing agent-based systems. Most agent-oriented methodologies follow the approach of extending existing software engineering methodologies to include abstractions related to agents. Agent methodologies capture concepts like conversations, goals, believes, plans or autonomous behaviour. Most of them take advantage of software engineering approaches to design MAS, and benefit from UML and/or AUML diagrams to represent these agent abstractions.

There are many methodologies with different strengths and weakness and different specialized features to support different applications domains. Clearly there is not a widely used or general-purpose methodology, but we took into account some issues like the modelling diagrams used, the kind of application domain it is appropriated for, and above all, the level of detail provided at the design phase and the available documentation. Some methodologies were not considered in this first approach because of their lack of public documentation or the level of detail achieve at the design phase. After examining current research in this area, and despite there were

other good candidates, such as Mase [9], we only use Tropos [10,11] for illustrating our proposal.

2.2 Tropos

Tropos [10] is an agent-oriented methodology created by a group of authors from various universities in Canada, Italy, Belgium and Brazil. Tropos is founded on the concepts of actor and goal and strongly focus on early requirements. The development process in Tropos consists in five phases: *Early Requirements*, *Late Requirements*, *Architectural Design*, *Detailed Design* and *Implementation*.

The first phase identifies actors and goals represented by two different models. The actor diagram depicts involved roles and their relationships, called dependencies. These dependencies show how actors depend on each other to accomplish their goals, to execute their plans, and to supply their resources. The goal diagram shows the analysis of goals and plans regarding a specific actor in charge of achieving them. Goals and plans are analysed based upon reasoning techniques such as means-end analysis, AND/OR decomposition, and contribution analysis. These models will be extended in the next phase, which models the systems within its environment. The goals can be decomposed into sub-goals.

The third phase is divided in three steps. In the first one, new actors, which are derived from the chosen architectural style, are included and described by an extended actor diagram. These actors fulfil non-functional requirements or support sub-goals identified in the previous phase. The second and third steps identify the capabilities, and group them to form agent types, respectively. The last step defines a set of agent types and assigns each of them a set of capabilities. This assignment, which is not unique and depends on the designer, is captured in a table.

The *Detailed Design* phase deals with the detailed specification of the agents' goals, belief and capabilities. Also communication among agents is specified in detail. This phase is usually strictly related to implementation choices since it is proposed within specific development platforms, and depends on the features of the adopted agent programming language. This step takes as input the specification resulting from the architectural design and generates a set of UML activity diagrams for representing capabilities and plans, and AUML sequence diagrams for characterizing agent interaction protocols. AUML is an extension of UML to accommodate the distinctive requirements of agent, which results from the cooperation established by FIPA and the OMG. This is achieved by introducing new classes of diagrams into UML such as *interaction protocol diagrams* and *agent class diagrams*.

Finally, the implementation phase follows the detailed design specification given in the previous phase. Tropos chooses a BDI platform for the implementation of agents, namely JACK Intelligent Agent [17], an agent-oriented development environment built on top of Java. The main language constructs provided by this platform (agents, capabilities, database relations, events and plans) have a direct correspondence with the notions used in the design phase. In addition, Tropos provides guidelines and heuristics for mapping Tropos concepts into BDI concepts, and BDI concepts into JACK constructs.

3 The Malaca Agent Model

Most existing agent architectures focus on the type of agent (BDI, reactive), but do not provide direct support for handling and reusing properties and functionality separately. This approach results in agent design and implementations being quite complex, brittle, and difficult to understand, maintain, and reuse in practice.

The Malaca agent architecture is based on the definition and reuse of software components, let them be either in-house or commercial-off-the-shelf (COTS) components. In addition, by applying the separation of concerns principle promoted by aspect-oriented software development [18], we separate into different and decoupled entities the distribution of messages through FIPA-compliant platforms, the codification of exchanged messages in FIPA ACL formats, and also the agents' functionality from their coordination. By "componentizing" agents in such way, they can be reused and replaced independently to build specific agent architectures independently from the underlying agent platform(s) used. This also enables dynamic composition of agent at runtime, allowing the dynamic reconfiguration and adaptation of the agents' behaviour to support new interaction protocols and functionality, to access to different agent platforms, or to use different ACL formats. Besides, when treated as components, Malaca agents are simply configured using XML documents that contain the agents' descriptions.

Figure 1 shows the meta-model of the Malaca agent model (a part of its UML profile), where all these entities are explicitly represented. This diagram also represents the basic structure of the XML agent description that is used to create an agent since there is a direct correspondence between the UML model shown there and the XML representation of the agent: meta-model classes and association represent the XML elements, and class attributes represent XML element attributes.

In order to produce agents able to be executed in any FIPA-compliant agent platform, we have separated everything related with the use of Message Transport Service (MTS), bundling it into a "distribution aspect". This distribution aspect will be later bound to the particular adaptors (*plug-ins*) of the corresponding agent platforms on which the agent instance will be run. Then, the actual distribution of messages using a particular message transport service offered by a FIPA-compliant agent platform is performed by an independent entity, the *adaptor*. Such adaptor defines a common interface, which will be realized by each concrete adaptor instance of the target agent platform(s), which will deal with the specific services of such platform(s). Since agent platform dependencies are encapsulated as an external plug-ins, our agents can be adapted to engage in any FIPA-compliant agent platform, and even be used in more than one agent platform simultaneously (for additional details see [16]).

As stated before, the encoding format of messages exchanged by the agent within an interaction is also bundled in a separated entity. Thus, the codification of ACL messages in a concrete FIPA format is merged neither with the agent platform access, nor with the behavior of the agent. In our agent architecture, parsers deal with different ACL representations. Each parser has to realize a common interface to code and decode output and input messages. In the model, for each different ACL format supported, we provide an *ACLParser* plug-in that parses ACL messages formatted

according to the value of the *format* attribute. Once again, the agent could support more than one ACL format at the same time.

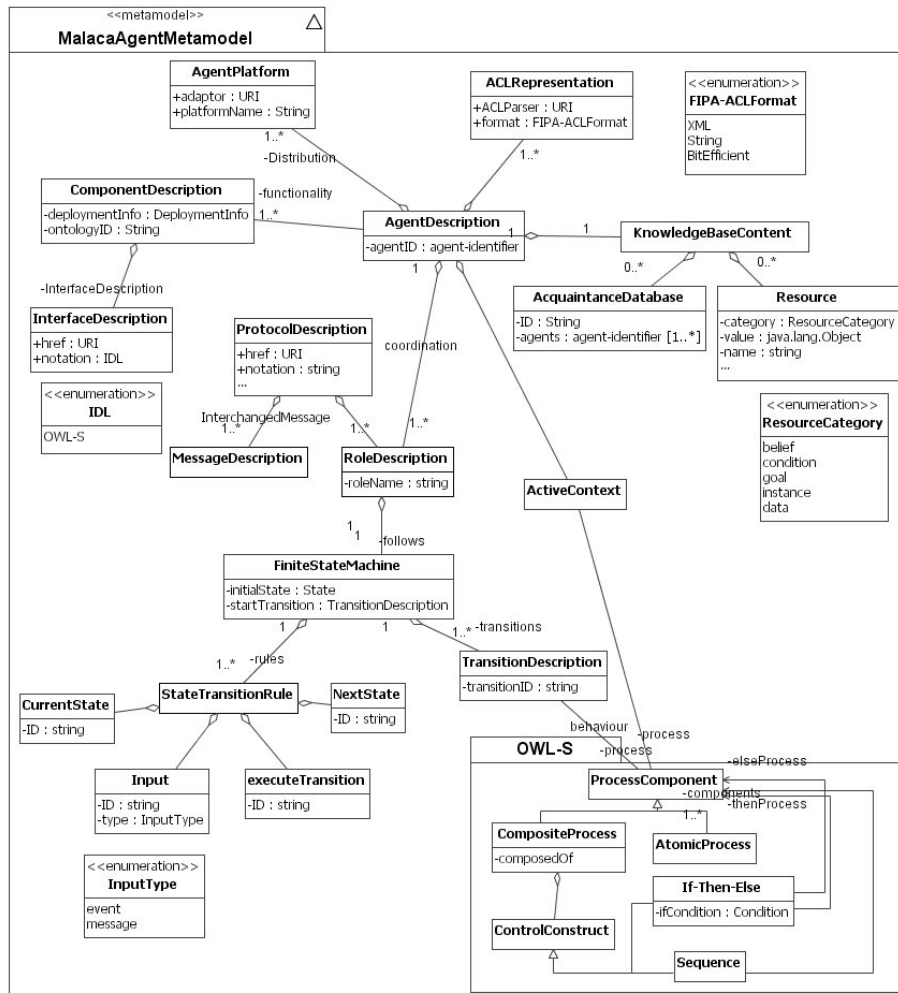


Figure 1. UML class diagram with a Malaca agent description

The behaviour of a Malaca agent is given by its functionality and by the way it interacts with other agents (i.e., its *coordination* aspect). In our model, agent functionality is provided by reusable software components, which offer the set of core services, and also the application-dependent functionality. Components that are initially plugged into the agent architecture are packaged into the functionality element of the agent description. Each component is described in a *ComponentDescription* class, which provides information about its provided interface and its implementation. The component interface describes the set of offered services in an XML document in the format specified by the *notation* element (by default,

OWL-S [19]). The *deploymentInfo* attribute points to a XML document with the description, using the *notation* format (by default, the CCM *softpackage* format [20]), of the component implementation. This information includes the kind of implementation (e.g. Java, CORBA, Web service), how to locate and deploy the component, etc.

Coordination is also modelled by using an independent entity called *connector*, which decouples the agent functionality from its interactions. Every time a new conversation starts, a new connector is created to control it. For this task, the connector uses a description of the interaction protocol followed. The set of roles of interaction protocols supported is given by the *coordination* element. Each role description is part of an interaction protocol is described by a XML document using the *notation* format (by default, the *ProtocolDescription* XML schema [15]).

The UML class diagram in Figure 1 also depicts the structure of a protocol description. Agent interaction protocols are described by the set of message description, interchanged during the interaction and by a set of finite state machines for representing the behavior of each participant role. The description of a message, in a *MessageDescription* element, should include the performative and should contain at least some description of the message content. The value of any other message field can also be specified. A separate finite state machine within the *RoleDescription* class, identified by an attribute *roleName*, describes each side of a conversation (at least the initiator and the responder). Each finite state machine is represented by a set of state transition rules enclosed by the *FiniteStateMachine* class and each rule is defined in a *StateTransitionRule* class.

The transition from a state to another carries out the execution of the agent functionality (defined in the *StateTransitionRule* by the attribute *executeTransition*). The *TransitionDescription* class encloses the set of agent actions that are invoked during protocol execution. Instead of a simple sequence of invocations to the agent internal functionality, it is possible to use more complex control structures to coordinate the execution of the agent functionality. OWL-S provides the basis for the definition of agent functionality as services. As part of the DARPA Agent Markup Language [21] program and within the OWL-based framework, OWL-S is an ontology for describing Web services that gives a detailed description of a service's operation, and provides details on how to interoperate. The control structures defined in the *Process Model* of OWL-S are used to encompass a set of agent actions in a transition description.

Finally, the agent description also contains the initial content of the agent Knowledge Base, expressed in terms of beliefs, goals, and conditions; the acquaintance database, defined as a set of the identifiers of the agents with which the agent will interact; and an active context of the agent upon start up. Within an *ActiveContext* class, it is possible to specify the initial behaviour the agent will execute (expressed in OWL-S by a sequence of actions, a set of protocols executed in parallel, etc.).

The Malaca UML Profile, which is derived from its metamodel, defines *stereotypes* for each metamodel element and also defines *constraints*, associated to stereotypes, which impose restrictions on the corresponding metamodel elements. Constraints can be used, for instance, to state that the attribute *transitionID* has a unique value for all the elements in the *transitions* collection of a *FiniteStateMachine*.

The abovementioned restriction can be expressed by the following OCL [22] constraint:

Context MalacaAgentMetamodel::FiniteStateMachine
Inv: self.transitions -> isUnique(transitionID)

The Malaca agent model is implemented in Java, and currently provides adaptors for Jade [23], Zeus [24], and FIPA-OS [25] agent platforms. It also supports String and XML ACL encodings. One of the benefits of this model is that the only artifacts that should be provided by the developer to define a MAS in Malaca are the XML documents with the agent descriptions, the components provided interfaces, and the protocol descriptions. We shall see in the next section how these artifacts can be even automatically generated from the MAS designs produced by Tropos. This will allow a direct connection between the MAS design and its implementation in any of the agent platforms.

4 Applying MDA to MAS design to produce implementations

The problem of transforming the design diagrams produced by a given agent-oriented methodology to a set of implementation classes of an agent platform API, such as the ones provided by FIPA-OS, Zeus, or Jade, can be addressed by expressing such designs and agent platforms as *models*, and then expressing the transformations between them in terms of *mappings* between models. The OMG Model Driven Architecture (MDA) provides the right kind of mechanisms for expressing such kind of models, the entities of each one, and for defining transformation between them.

MDA is an approach to system development based on the use of models, which are descriptions of a system and its environment for some certain purpose. A model is often presented as a combination of drawings and text (the text may be in a modelling language or in natural language). Regarding a set of models, MDA sets down how those models are prepared, and the relationships between them. In MDA, a platform is a set of subsystems and technologies that provides a set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented. MDA distinguishes between platform-independent models (PIM) and platform-specific models (PSM).

The general MDA model transformation is depicted by the MDA pattern, shown in Figure 2. The PIM and some other information are combined by the transformation to produce a PSM. MDA defines many ways in which such transformations can be done. A MDA mapping provides specifications for transformation of a PIM into a PSM for a particular platform. A model type mapping specifies a mapping from any model built using types. Another approach to mapping models is to identify model elements in the PIM, which should be transformed in a particular way, given the choice of a specific platform for the PSM. However, most mappings will consist in some combination of type and instance mappings. A mapping may also include templates to specify particular kinds of transformations. In order to apply these concepts to agent technologies, we need to define agent-oriented PIMs and PSMs, and mappings

between them. Here, the design model of a MAS produced by an agent-oriented methodology will constitute the PIM, that needs to be marked using the UML profile of the target agent platform to produce a PSM.

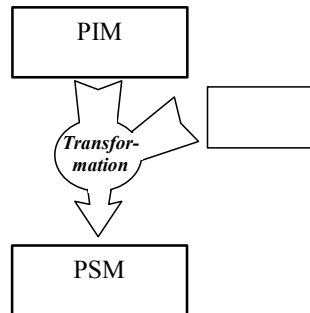


Figure 2. The MDA pattern for model transformation

To illustrate this approach, in this paper we will apply MDA to transform Tropos design model to the Malaca agent model. An important benefit of our agent model is that it does not depend on the target agent platform on which it will be executed, and therefore there is no need to develop different implementations for each FIPA-compliant agent platform. This fact greatly simplifies the process of providing an implementation for each different agent platform. Figure 3 graphically shows the process that will be described in detail in the next sections. The transformations are illustrated using UML, AUML produced by the agent-oriented methodology used here.

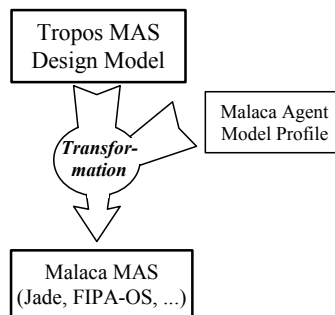


Figure 3. The MDA model transformation from Tropos design model to the Malaca MAS specification using the Malaca Agent Model UML Profile

4.2 Applying MDA: From Tropos to Malaca

Now we will show through an example how the set of models resulting from applying Tropos can be transformed in a set of Malaca agents using the MDA mechanisms. In the design phase, Tropos details agent interactions, capabilities and plans. The designer assigns one or more capabilities to each agent. The Malaca agent model

deals with XML agent descriptions that contain references to software components interfaces, protocols descriptions in XML, agent platform adapters and so on.

To illustrate the transformation for the Tropos design model, we will use the diagrams supplied in [10], which provides a case study. Unfortunately, available literature of Tropos does not provide a complete example, and also the process that explains each phase varies from paper to paper.

Marking Tropos Detailed Design Model. As stated before, in Tropos, the design phase deals with the detailed specification of the agents, capabilities and communications. More precisely, the design phase in Tropos produces:

- Agent assignments, expressed as a table resulting from the architectural design phase, that defines the agent types and the capabilities assigned to each agent. An agent can have assigned capabilities that are associated to different actors.
- Agent Interaction Diagrams. AUML sequence diagrams are used to model basic interactions between agents. In [11], interactions are described by introducing additional interactions, together with constraints on the exchanged messages.
- Capability Diagrams. One UML activity diagram models each capability. External events, such as input messages, define the starting state of a capability diagram; activity nodes model plans, transition arcs model events, and beliefs are modelled as objects. UML activity diagrams can further specify each plan node of a capability at the lowest level. In this case, the activity nodes correspond to simple or complex actions.

Now we will show how the Tropos detailed design given in [10], can be transformed into a Malaca Model. In order to define this transformation we will “mark” the Tropos design model using the classes defined in the Malaca metamodel showed in Figure 1. A *mark* represents a concept stereotyped in the Malaca profile and is applied to an element of the Tropos design model to indicate how it should be transformed. Thus, we will mark Tropos design elements (agents, interactions, messages, capabilities, plans and so on), with the corresponding Malaca entities that will implement them (*AgentDescription*, *ProtocolDescription*, *TransitionDescription*, OWL-S processes, etc). Agent types are marked as *AgentDescription*. Each capability is marked as a *TransitionDescription*, and each interaction (represented in UML sequence diagram) is marked as a *ProtocolDescription*.

The elements of each UML diagram can also be marked. Figure 4 shows the marked agent interaction diagram given in [10]. Every agent (object) in the agent interaction diagram (UML interaction diagram) is marked as a *RoleDescription*, and every communicative act between agents is marked as a *MessageDescription*, but also as an *Input* (an element of a *StateTransitionRule*). Also, we specify the value of the tagged value (that corresponds to an attribute of the metamodel class) associated to the stereotype element including notes that show the corresponding stereotype, the name of the tagged value, and the value assigned to it.

Figure 5 displays the marked diagram of the capability *Present Query results*. The external event is marked as a *MessageDescription* and every plan —activity node— is marked as an *Atomic* or a *Composite* OWL-S process. If the plan node is not further specified in another UML activity diagram it is marked as an Atomic process (see *Present Empty Result* and *Present Query Results* node s). Otherwise, it is marked as a

Composite process, as occurs with the *Evaluate Query Result* plan, since it is further specified in another UML activity diagram. Plan diagrams are also marked.

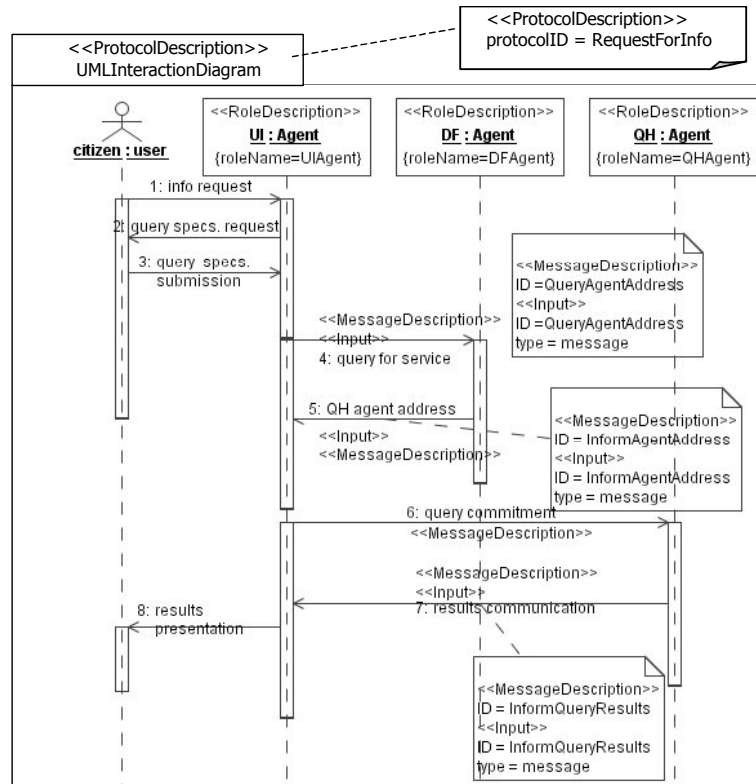


Figure 4. Marked Agent Interaction Diagram in Tropos (extracted from [10]).

Tropos to Malaca Mappings and Transformations. After applying the marks defined in the Malaca profile, we obtain a set of marked UML and AUML diagrams. The transformation process applies mapping rules for the same mark depending on the marked element. The result of the application of such mapping rules is, in this case, a set of XML documents that specify the PSM of the system for the Malaca platform.

In order to illustrate the mapping rules, we will describe here, as example, some rules that have been applied to a few elements marked in the diagram of Figure 5.

- When a mark <<TransitionDescription>> is applied to a UML activity diagram the transformation produces an XML instance of the complex type TransitionDescription (defined in a XML schema). The value of the XML attribute is taken from the value assigned to the tagged value ID in the note (in the example is *PresentQueryResultToTheUser*).

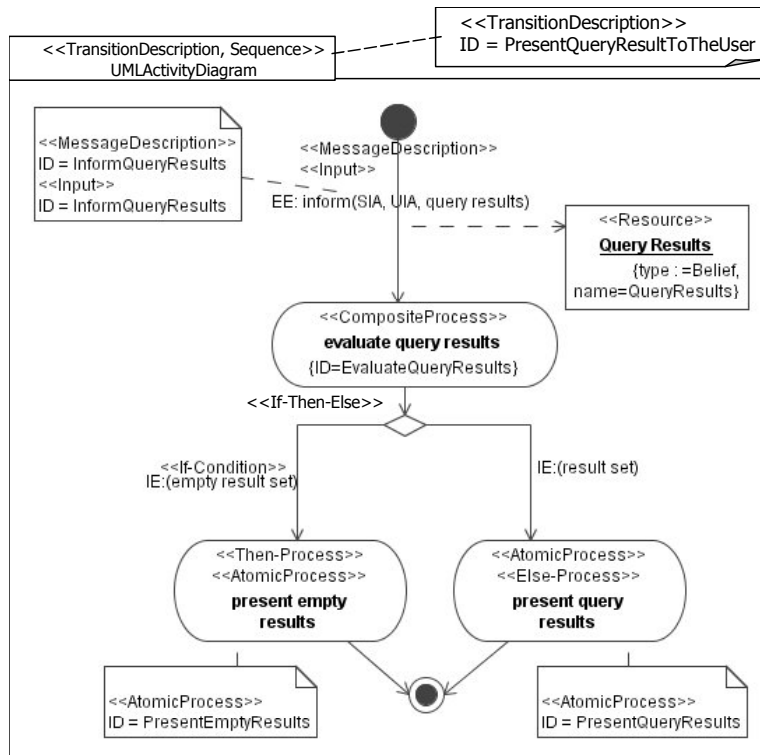


Figure 5. Marked Capability Diagram (extracted from [10]).

- When a mark `<<Sequence>>` is applied to a UML activity diagram the transformation produces the definition of a *CompositeProcess* XML description attached to the *TransitionDescription* element produced by the application of the previous rule. The components of the *Sequence* are derived from the application of the following transformation rules.
- When a mark `<<AtomicProcess>>` is applied to an action state element within a UML activity diagram the transformation produces the XML description of an *AtomicProcess*, which is included as a *component* of the *ControlConstruct* element produced by the application of the previous rule.
- When a mark `<<CompositeProcess>>` is applied to an action state element within a UML activity diagram the transformation produces the XML description of an *CompositeProcess*, which is included as a *component* of the *ControlConstruct* element produced by the application of the previous rule.
- When a mark `<<If-Then-Else>>` is applied to a junction element within a UML activity diagram the transformation produces the XML description of an *If-Then-Else* control construct which is included as a *component* of the *ControlConstruct* element produced by the application of the previous rule.
- When a mark `<<IfCondition>>` is applied to a transition element within a UML activity diagram that depart from a junction marked as `<<If-Then-Else>>`, the transformation produces the XML description of a condition, which is included

as a *ifCondition* element of the *If-Then-Else* element produced by the application of the previous rule.

- When a mark <<thenProcess>> or <<elseProcess>> is applied to a action state element, marked also as a <<AtomicProcess>>, within a UML activity, the transformation produces the XML description of an atomic process, which is included as a *then* (or *else*) element of the *If-Then-Else* element produced by the application of a previous rule.

The application of these transformation rules to the diagram of Figure 5 produces the XML description of a transition identified as *PresentQueryResults* as depicted in Figure 6.

```
-<TransitionDescription ID="PresentQueryResultsToTheUser">
  -<CompositeProcess>
    -<composedOf>
      -<Sequence>
        -<components>
          +<CompositeProcess ID="EvaluateQueryResult">
            -<If-Then-Else>
              -<ifCondition>
                <IsTrue resource="emptyResultSet" />
              </ifCondition>
              -<then>
                <AtomicProcess ID="presentEmptyResults" />
              </then>
              -<else>
                -<AtomicProcess ID="presentQueryResults"/>
              </else>
            </If-Then-Else>
          </components>
        </Sequence>
      </composedOf>
    </CompositeProcess>
  </TransitionDescription>
```

Figure 6. Present Query Result XML description (complete).

Also, we can apply the constraints expressed in OCL to ensure that the identifier of the transition is unique within the collection of transitions identifiers.

This is only a brief example of how MDA can be applied to transform elements of the Tropos design model into a Malaca agent description. Again, once we count with a Malaca description of the MAS, it can be implemented in any FIPA-compliant agent platform. Then, we can obtain a straightforward implementation from that design by applying MDA mappings and transformations again.

5 Limitations and Further Extensions to Our Work

One of the problems found when trying to implement multi-agent systems directly from their high-level designs and descriptions appear when the designer describes *what* needs to be done, but gives no indication on *how* (for instance, using heuristics, guidelines and examples rather than algorithms). In such cases, the transformations cannot be automated, since such algorithms have to be provided. In general, the level

of detail provided in the design phase determines the accuracy of the implementation, and therefore its potential automated implementation. Thus, an important aspect in selecting a methodology is the level of detail provided.

Second, the diagrams and texts used in the design phase have to be interpreted and mapped during the transformation process. Since one of our goals is to automate such mappings, it is very important for diagrams and texts used in agent-oriented methodologies to follow standard notations (such as UML), allowing its automatic processing. Besides, AUML diagrams should also allow for some kind of automated support, currently inexistent —although the advent of UML 2.0 and its new extensions mechanisms can alleviate this if AUML gets aligned with UML 2.0.

Finally, MDA seems a very attractive and powerful approach for automated development. However, it also has many unresolved issues, and still lacks tool support. The use of UML 2.0, the advent of QVT [26] and the emergence of new modelling tools that support the MDA principles can also help MDA get more mature and consolidate its ideas.

Apart from these shortcomings, there are further extensions to our work. First, we plan to develop some tools for automating the transformations described in this paper. Apart from providing a proof-of-concept to our work, we think they can be of great value to any MAS developer that follows MaSE or Tropos methodologies. They will also assist uncover some more issues of our proposal, helping us make it more robust.

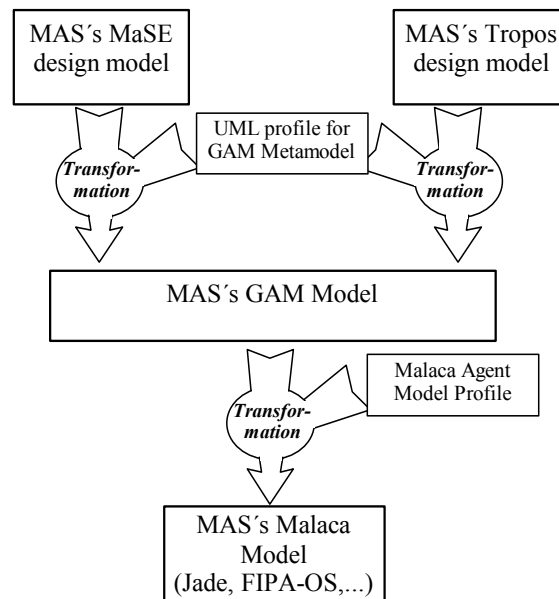


Figure 7. Derivation of MAS implementations using a middle GAM model.

A second line of work is related to the interesting idea by FIPA and the OMG to define a common agent model to most agent-oriented methodologies. Here we have seen the benefits of using a neutral model (Malaca) for MAS implementation purposes, which provides the common mechanisms provided by FIPA-compliant

agent platforms. In this way, any design model of the MAS produced using an agent-oriented methodology can be implemented by mapping it (using the MDA mechanisms) into its corresponding Malaca model. But this means that a different transformation is needed for every methodology into the Malaca model. Instead, a better solution is to identify and standardize the common elements of the existing agent-oriented methodologies at the design phase, as pursued by FIPA and the OMG. The common elements could form a generic agent model (GAM) on which specialized features of every agent-oriented methodology might be based. Thus, we could introduce an intermediate model that semantically can cope with the concepts managed at design time by agent-oriented methodologies. With this, MAS design models could be naturally mapped to a new design model conforming the element defined in the GAM. From there, we will count with a general and common model for MAS designing purposes, a common model for implementation purposes, and the only thing that needs to be done is to define (just) one MDA transformation from the GAM to Malaca, as shown in Figure 7.

6 Conclusions

In this paper we have presented how MDA can be effectively applied to agent technologies, providing a partially automated support for the derivation of MAS implementations right from their designs, independently from the methodology used to realize the design, and the target agent platform selected.

Our main contributions have been the definition of a common and neutral agent model that implements all the concepts required by FIPA-compliant agent platforms, and the use of the MDA mechanisms for defining the transformations between the design models produced by existing agent-oriented methodologies and the Malaca model. From there, the MAS implementation is quite straightforward. We have presented our experience in deriving and applying these transformations to a well-known methodology, Tropos.

References

1. M. Wooldridge, P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art", in *First Int. Workshop on Agent-Oriented Software Engineering*, LNAI 1957, 2000. pp. 1-28.
2. C.A. Iglesias, M. Garijo, J.C. Gonzalez, "A Survey of Agent-Oriented Methodologies", in *Intelligent Agents V – Proceedings of the Fifth International Workshop ATAL 98*, Springer-Verlag, 1998.
3. S. A. O'Malley, S.A. DeLoach, "Determining When to Use an Agent-Oriented Software Engineering Paradigm", in *Second International Workshop On Agent-Oriented Software Engineering*, 2001.
4. Sturn, O. Shehory, "A Framework for Evaluating Agent-Oriented Methodologies", in *International Workshop On Agent-Oriented Information Systems*, 2003.
5. K.H. Dam, M. Winikoff, "Comparing Agent-Oriented Methodologies", in *International Workshop On Agent-Oriented Information Systems*, 2003.

6. FIPA, “FIPA Methodology Technical Committee”, Foundation for Intelligent Physical Agents <http://www.fipa.org/activities/methodology>.
7. OMG, “OMG Agent Working Group”, in Object Management Group <http://www.objs.com/agent/>
8. M. Wooldridge, N. R. Jennings, D. Kinny, “The Gaia Methodology for Agent-Oriented Analysis and Design”, in *Journal of Autonomous Agents and Multi-Agent Systems*, vol.3, n.3, 2000. pp. 285—312.
9. S. A. DeLoach, M. F. Wood, C. H. Sparkman, “Multiagent System Engineering”, in *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, n. 3, 2001. pp. 231-258.
10. P. Bresciani, et al. “Tropos: An Agent-Oriented Software Development Methodology”, in *International Journal of Autonomous Agents and Multi-Agent Systems*, 2003.
11. J. Castro, M. Kolp, J. Mylopoulos, “Towards Requirements-Driven Information Systems Engineering: The Tropos Project”, in *Information Systems*, vol. 27, Issue 6, 2002. pp. 365-389.
12. MESSAGE: Methodology for Engineering Systems of Software Agents. Deliverable 1. Initial Methodology. July 2000. EURESCOM Project P907-GI.
13. B. Bauer, J. P. Muller, J. Odell, “Agent UML: A Formalism for Specifying Multiagent Software Systems”, in *International Journal of Software Engineering and Knowledge Engineering*, vol.11, n.3, 2001. pp 207—230.
14. OMG, “Model Driven Architecture. A technical Perspective”, Object Management Group, OMG Document `ab/2001-01-01`, 2001. Available from www.omr.org.
15. M. Amor, L.Fuentes, J.M. Troya, “Training Compositional Agents in Negotiation Protocols”, next publication in *Integrated Computer-Aided Engineering International Journal*, 2004.
16. M. Amor, L.Fuentes, J.M. Troya, “A Component-Based Approach for Interoperability Across FIPA-Compliant Platforms”, in *Cooperative Information Agents VII*, LNAI 2782, 2003. pp. 266—288.
17. The Agent Oriented Software Group, “Jack Development Environment”, <http://www.agent-software.com>
18. Aspect-Oriented Software Development, in <http://www.aosd.net>
19. The DAML Services Coalition, “OWL-S: Semantic Mark-up for Web Services” available at <http://www.daml.org/services/>
20. OMG, “CORBA Components. Packaging and Deployment”, in Object Management Group, OMG Document `formal/02-06-74`, June 2002. Available from www.omg.org.
21. The DARPA Agent Markup Language Homepage, <http://www.daml.org/>
22. Object Management Group. Object Constraint Language Specification, OMG document `ad/02-05-09`, 2002. Available from www.omg.org.
23. F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, “Jade Programmer’s Guide”, 2003, available at <http://sharon.cselt.it/projects/jade/>
24. J. Collis, D. Ndumu, C. van Buskirk “The Zeus Technical Manual”, Intelligent Systems Research Group, BT Labs. July 2000.
25. Emorphia, “FIPA-OS Developers Guide”, Nortel Networks' Agent Technology Group, 2002, available at <http://sourceforge.net/projects/fipa-os/>
26. OMG, “MOF 2.0 Query/View/Transformation RFP”, in Object Management Group, OMG Document `ad/03-08-03`. 2003. Available from www.omg.org.