Engineering Applications of Artificial Intelligence 25 (2012) 159–173

Contents lists available at SciVerse ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Model-driven engineering techniques for the development of multi-agent systems

José M. Gascueña [a], Elena Navarro [a,b], Antonio Fernández-Caballero [a,b,*]

[a] Instituto de Investigación en Informática de Albacete (I3A), 02071 Albacete, Spain
[b] Departamento de Sistemas Informáticos, Universidad de Castilla-La Mancha, 02071 Albacete, Spain

ARTICLE INFO

Article history:
Received 8 June 2010
Received in revised form
4 July 2011
Accepted 22 August 2011
Available online 9 September 2011

Keywords:
Agent-based method
Model-driven development
Meta-modeling
MDE-MAS method and tool
Agent-oriented software development
Multi-agent systems
Surveillance systems
Eclipse-Modelling Framework
Graphical Modelling Framework

ABSTRACT

Model-driven engineering (MDE), implicitly based upon meta-model principles, is gaining more and more attention in software systems due to its inherent benefits. Its use normally improves the quality of the developed systems in terms of productivity, portability, inter-operability and maintenance. Therefore, its exploitation for the development of multi-agent systems (MAS) emerges in a natural way. In this paper, agent-oriented software development (AOSD) and MDE paradigms are fully integrated for the development of MAS. Meta-modeling techniques are explicitly used to speed up several phases of the process. The Prometheus methodology is used for the purpose of validating the proposal. The meta-object facility (MOF) architecture is used as a guideline for developing a MAS editor according to the language provided by Prometheus methodology. Firstly, an Ecore meta-model for Prometheus language is developed. Ecore is a powerful tool for designing model-driven architectures (MDA). Next, facilities provided by the Graphical Modeling Framework (GMF) are used to generate the graphical editor. It offers support to develop agent models conform to the meta-model specified. Afterwards, it is also described how an agent code generator can be developed. In this way, code is automatically generated using as input the model specified with the graphical editor. A case of study validates the method put in practice for the development of a multi-agent surveillance system.

1. Introduction

Currently, the use of the model-driven engineering (MDE) approach throughout the software development process is gaining more and more attention (Gasevic et al., 2009). MDE concerns the exploitation of models as the cornerstone of the software development process. It allows both developers and stakeholders to use abstractions closer to the domain than to computing concepts. Thus, it reduces the complexity and improves the communication. As the main aim of MDE is to develop software, this paradigm uses software models as their expression vehicle.

Sometimes, models are constructed to a certain level of detail, and then code is written by hand in a separate step. Some other times (most often) code is automatically generated from the models, ranging from code skeletons to completely deployable products. Usually, these models are specified by instantiating *meta-models*, that is, models to describe models. The basic idea of meta-model is to identify the general concepts in a given problem domain and the relations used to describe models. This serves as a strategy that forces a clear distinction between the real problem to be solved by the system and the framework where the model lives.

The use of MDE has the following consequences for a software development process. (1) More time can be devoted to analyzing and designing models. (2) The time necessary to perform coding tasks is reduced, as code generators are usually available to carry them out in an automatic way. The programmers are responsible for completing those parts of the system that developers either have decided not to generate or cannot do. (3) The quality of the developed system is improved, as the generated code (usually) does not have bugs. And, (4) productivity is improved as the time necessary for coding is reduced. More effort is devoted to solve errors during early phases of the life cycle, avoiding in this way the "snow ball" effect (Pressman, 2010). Moreover, MDE provides inter-operability among heterogeneous systems thanks to the specification of bridges between different technologies. Portability is also improved to adopt a new technology, just developing a new code generator, as the models are independent of any technology. In summary, MDE offers important benefits in aspects as important as productivity, portability, inter-operability and maintenance (Kleppe et al., 2003).

In contrast, MDE also demonstrates some drawbacks (Mattsson et al., 2009). Although MDE automates the steps from detailed design to implementation, as described before, at present

* Corresponding author at: Departamento de Sistemas Informáticos, Universidad de Castilla-La Mancha, 02071 Albacete, Spain. Tel.: +34 967 599200; fax: +34 967 599224.
E-mail address: Antonio.Fdez@uclm.es (A. Fernández-Caballero).

0952-1976/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.engappai.2011.08.008

it is not able to automate enforcement of the architecture on the detailed design. This is due to the inability to model architectural design rules. Unfortunately, this is a bottleneck in large MDE projects, as the developers have to carry out this task manually, that is, in a similar way to more traditional approaches.

On the other hand, multi-agent systems (MAS) are appropriate to model and develop complex software applications with high need of autonomy, communication among autonomous elements and distribution (Jennings et al., 1993; Karageorgos et al., 2003; Posadas et al., 2008; Leitão, 2009). More and more, MAS are introduced in different domains (e.g. teleoperated systems, Rodríguez-Seda et al., 2010, intruder detection systems, Jha and Massan, 2002, and so on). Agent-oriented software development (AOSD) (Henderson-Sellers and Giorgini, 2005) is the paradigm described for the construction of this kind of systems. Lately, several methodologies, such as Gaia (Wooldridge et al., 2000), MaSE (Deloach et al., 2001), ADELFE (Bernon et al., 2003), Prometheus (Padgham and Winikoff, 2004), Tropos (Bresciani et al., 2004), and INGENIAS (Pavón et al., 2006) have come up following this paradigm. Every one of them exhibits the characteristics that a software methodology (Bauer and Odell, 2005) should have, that is, a *modeling language* and a *software process*. A modeling language is used for the specification of the corresponding models by using its specific syntax (notation) and its associated semantics. A software process specifies the development activities, the inter-relationships among them, and how they are performed. In the definition of the AOSD methodologies two different approaches have been followed.

In first place, some of them, such as ADELFE, extend a generic modeling language – Unified Modeling Language (UML) (Rumbaugh et al., 2004) – and a process described in the context of software engineering – Unified Software Development Process (Jacobson et al., 1999). Other approaches, such as Prometheus, have their own language and development process. However, no matter which approach is followed by a methodology, there are always compelling arguments to provide tool support for their application. This is why, the MDE approach for building supporting tools emerges naturally as a way to improve the development of agent-based software applications. This is the main argument that has conducted this work, namely, to show how the AOSD paradigm can be integrated with the MDE approach.

The rest of the paper is organized as follows. In Section 2 some previous works related to our proposal are revisited. Then, in Section 3 our new proposal of the use of model-driven engineering in Prometheus methodology are described. In Section 4 the Prometheus Model Editor is introduced in detail through describing the meta-model definition, the graphical editor construction and the code generation. A case of study related to the development of a multi-agent surveillance system is used in Section 5 to validate the method. Lastly, Section 6 offers some conclusions and hints towards future work.

2. Related works in model-driven engineering for multi-agent systems

Meta-models define general concepts of a given problem domain and their relationships. General concepts and relationships are really a language that, for instance, may be used to specify the domain's requirements (Smolík, 2006). The advantage of introducing meta-models in the development process is the higher abstraction level to work with.

According to Molesini (2008), meta-models should be used in AOSD as they describe each methodology and infrastructure in a compact and precise way. Moreover, they form the basis for analyzing and comparing methodology and infrastructure. Indeed, the elements and the relationships that describe them are present in the meta-models. Besides, they help to study the existing gap between agent-oriented methodologies and agent-oriented infrastructures, as they allow one to isolate the main concepts of the system from the underlying technology. Finally, meta-models are the starting point to define methodologies along with their corresponding agent-oriented infrastructures.

Unfortunately, there are several agent-oriented modeling languages but not a standard one. In principle UML could be considered as the standard to be used, but it is not the best tool for modeling agent-based systems (Bauer, 2001). This is basically due to two reasons: (1) compared to objects, agents are active as they take initiative and have control over external requests; and (2) agents do not only act in isolation but in cooperation or coordination with other agents. Several languages that extend UML have been proposed so far to solve this problem. For instance, Agent UML (AUML) (Bauer et al., 2001) is the first agent modeling language that follows this approach. It provides interaction protocol diagrams and agent class diagrams as extensions of UML's sequence and class diagram, respectively, as a solution to the stated problems. However, the absence of a meta-model and modeling tools are the main drawbacks that explain why this language is not widely accepted. More recently, the Agent Modeling Language (AML) (Cervenka and Trencansky, 2007), a semi-formal visual modeling language based on the UML 2.0 superstructure has been proposed. It is supported by tools like (Enterprise Architect, 2010; StarUML, 2010). However, AML does not concern about code generation, as it focuses its attention on specification tasks. Despite the main aim of AML is to offer a new and well documented unified language suitable for industrial development, unfortunately, most research groups are still using classic methodologies (Henderson-Sellers and Giorgini, 2005), such as INGENIAS or Prometheus, to carry out the modeling of agent-based applications.

Now let us focus on one of the principal agent-oriented methodologies, namely INGENIAS (Pavón et al., 2006). The basis of INGENIAS methodology is the definition of a MAS meta-model described by using GOPRR (Graph, Object, Property, Relationship, and Role) (Kelly et al., 1996). A set of agent-oriented MDE tools (model edition, verification, validation and transformation) are integrated into the INGENIAS Development Kit (IDK) (Gómez-Sanz et al., 2008). The INGENIAS meta-model describes the elements for modeling MAS from different perspectives—agent, organization, environment, goals and tasks, and interaction (Fuentes-Fernández et al., 2010). The agent perspective focuses on the elements necessary to specify the behavior of each agent. The organization perspective shows the system architecture. From a structural point of view, the organization is a set of entities with aggregation and inheritance relationships used to define a schema where agents, resources, tasks and goals exist. Under this perspective, groups may be used to decompose the organization, plans, and workflows to establish the way the resources are assigned, whose tasks are necessary to achieve a goal, and who has the responsibility for carrying them out. The environment perspective defines the agents' sensors and actuators, and identifies the system resources and applications. The goals and tasks perspective describes the relations between tasks and goals. The interaction perspective describes how the coordination among agents is performed. The IDK tool supports the INGENIAS methodology, so that each one of the previous concepts are specified using either an UML-like or INGENIAS specific notation. This facility allows users familiar with the UML notation to reduce the learning curve of INGENIAS. Moreover, the IDK tool has a module for JADE code generation, and a mechanism to define templates used to develop code generation modules for the required target platform.

Now, PIM4Agents (Hahn et al., 2009) is another meta-model that takes into account seven points of view to deal with the different aspects of the system: multi-agent, agent, organization, interaction, role, behavior, and environment. Every aspect is defined by means of a sub-meta-model that, jointly with the others, make up the PIM4Agents meta-model. Moreover, the authors propose rules to transform PIM4Agents (platform-independent) models to written code by using the agent-oriented programming languages JACK (Winikoff, 2005) and JADE (Bellifemine et al., 2007). These transformations are defined using the JackMM and JadeMM (platform-specific) meta-models of JACK and JADE, respectively.

Regarding technical aspects, all these meta-models have been described by using Ecore. Model-To-Model (M2M) transformations have been specified using Atlas Transformation Language (ATL) (Jouault et al., 2008) to map platform-independent concepts to platform-specific concepts. Also, the MOFScript language (Oldevik, 2009) is used to generate code written in JACK or JADE. Besides, PIM4Agents, similarly to INGENIAS meta-model, may evolve by (1) adding new modeling concepts in the existing aspects; (2) extending the modeling concepts of the defined aspects; and (3) incorporating new modeling concepts to describe additional aspects for MAS. The available environment DSML4MAS (Warwas and Hahn, 2009) offers support to create diagrams conforming to the PIM4Agents meta-model and to generate JACK and JADE code.

Also, the language provided by the Prometheus methodology (Padgham and Winikoff, 2004) allows us to describe a MAS by using several diagrams. In general, diagrams to specify requirements, to show the system overall architecture, and to describe the internal architecture of each agent and its interactions are available. The set of concepts, and relationships among them, used in these diagrams are specified with a meta-model (Dam et al., 2006) represented by means of UML language. The Prometheus Design Tool (PDT) supports the modeling using the Prometheus methodology language. Moreover, it provides a code generator, which is hard-code implemented, to facilitate the code generation in JACK language from the developed models.

Regarding the software process followed to develop a MAS, using the three agent modeling languages described previously, it is found that: (1) INGENIAS adopts the Unified Software Development Process (USDP) to define the steps necessary to develop the MAS entities; (2) on the contrary, Prometheus defines its own development process, providing guidelines to identify the MAS entities; and (3) there is not a process related to PIM4Agents, but it is the developer's experience which determines the completion of the system views.

Table 1 offers a brief comparison among Prometheus, INGENIAS and PIM4Agents technological features. Specifically, these are (1) the meta-model language used to represent its modeling language, (2) their modeling tool, (3) the selected mechanism to implement code generation, (4) the code generated by the tool, (5) the software development process, and (6) guidelines defined to discover agents. The three proposals use different meta-modeling languages and are supported by tools. Prometheus takes a clear advantage in front of INGENIAS and PIM4Agents: it provides a collection of guidelines that help to determine the agents that make up a MAS. However, the code generation functionality provided by Prometheus is hard-code implemented so that it is difficult to maintain, whereas INGENIAS and PIM4Agents implement it by using a template-based mechanism. Therefore, in order to solve the drawback of Prometheus in regard to INGENIAS and PIM4Agents, the exploitation of meta-modeling techniques emerge in a natural way as a powerful solution.

There are other tools that also support the visual design of agent applications. JACK Development Environment (JDE) (Agent Oriented Software Pty. Ltd., 2008) is an integrated development environment that provides graphical tools for writing plans, connecting plans to agents, managing inter-agent communication, as well as for compiling and running JACK agent applications. This tool automatically generates JACK language code for all graphically represented elements (e.g. agents, events, plans, capabilities, data). WADE (Workflows and Agents Development framework) (Caire et al., 2009) is a software platform built on top of JADE that facilitates the development of distributed multi-agent applications where agent tasks are defined according to the work flow metaphor. WOLF (WOrkflow LiFe cycle management environment) (Caire et al., 2008) is an Eclipse plug-in that provides support to the graphical definition of workflows to develop WADE-based applications. A workflow is implemented as a Java class because the language does not provide support for this aim. In short, JDE and WOLF are tools related to a specific implementation platform, whereas others such as IDK, PDT and DSML4MAS, as well as the Prometheus Model Editor (see Section 4), are related to an agent modeling language. It is worth noting that models specified by using JDE and WOLF are at a lower abstraction level than those specified by using INGENIAS or Prometheus, what means a clear difference. Moreover, JDE and WOLF do not provide functionalities to develop new code generators capable of generating code for an agent platform different from JACK and JADE, respectively.

3. Model-driven engineering in Prometheus methodology

As aforesaid, the main argument that has conducted this work is to demonstrate that MDE techniques are a perfect complement for developing multi-agent systems. So, the development of MAS can be carried out by exploiting models at different stages of the process. All necessary MDE techniques are put into practice with the objective of exemplifying the proposal in the Prometheus agent-oriented software methodology (Padgham and Winikoff, 2004). Fig. 1 summarizes the technologies used to achieve our goal from two different perspectives:

- How can MDE be used to build tools for MAS? In this work, the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) is used to specify the Prometheus Ecore meta-model (PEMM), necessary to describe the modeling elements of the Prometheus methodology. Moreover, the Graphical Modeling

Table 1
Comparing Prometheus, INGENIAS and PIM4Agents.

Feature	INGENIAS	Prometheus	PIM4Agents
Meta-model language	GOPRR	UML	Ecore
Support tool	IDK	PDT	DSML4MAS
Technology for code generation	Template based proprietary mechanism	Hard-code	MOFScript
Generated code	JADE	JACK	JACK and JADE
Development process	Unified Software Development Process	Proprietary	–
Mechanism to discover agents	No	Yes	No

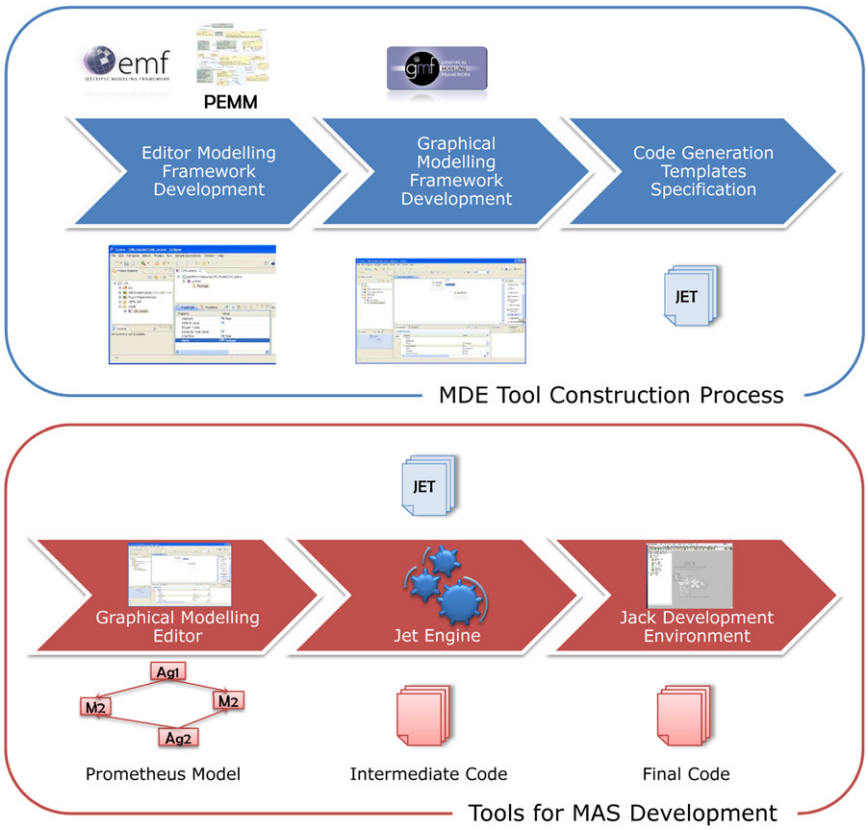


Fig. 1. Technologies used in the proposal.

Framework (GMF) (Gronback, 2009) is exploited to build a graphical editor for specifying the Prometheus models. Afterwards, Java Emitter Templates (JET) technology (Vogel, 2009) is used to create templates for automatically generating the intermediate code.

- How can MAS be specified by applying an MDE approach? Prometheus Models are specified and intermediate code is generated by exploiting the tools obtained by means of the previous process. Subsequently, the process to import the generated code into the JACK Development Environment (JDE) (Agent Oriented Software Pty. Ltd., 2008) to complete the implementation by using the agent-oriented programming language JACK (Winikoff, 2005) is also illustrated.

The functionality offered by the proposed tool (definition of models according to the Prometheus methodology and JACK skeleton code generation) is not totally new. Currently, the Prometheus Design Tool (PDT) (Padgham et al., 2008) already provides support for these tasks. However, the main difference of this new proposal is that PDT does not take advantage of all the benefits provided by MDE. So, our solution is a technological innovation in the specific context of MAS applications developed following the Prometheus methodology.

The Prometheus methodology (Padgham and Winikoff, 2004) is defined as a proper detailed process to specify, implement and test/debug agent-oriented software systems. It offers a set of detailed guidelines, including examples and heuristics, which provide a better understanding of what is required in each step of the development process. This process incorporates three phases:

- The System Specification phase identifies the basic goals and functionalities of the system, develops the use case scenarios that illustrate how it works, and specifies the inputs (*percepts*) and outputs (*actions*). Its main results are the analysis overview diagram, scenarios diagram, goal overview diagram, and system roles diagram.
- The Architectural Design phase uses the outputs of the previous phase to determine the types of agents of the system and their interactions. Its main results are the data coupling diagram, agent-role diagram, agent acquaintance diagram, and system overview diagram.
- The Detailed Design phase focuses on developing the internal structure of each agent and how each agent performs its tasks within the global system. It obtains the agent overview and capability overview diagrams.

Notice that the two first phases (System Specification and Architectural Design) do not assume a particular agent architecture. It is only in the last phase, the Detailed Design phase, where Prometheus targets a particular agent architecture named BDI (belief-desire-intention) (Georgeff et al., 1998). The key concepts in this architecture are the beliefs (what the agent knows does not know about the world), desires (what the agent wants to do) and intentions (how the agent plans to do what it wants to do) (Ronald and Sterling, 2005). One advantage of this architecture is that for reasoning it uses understandable common terms used by humans. Another advantage is that it has been implemented in several agent programming languages such as JACK (Winikoff, 2005), Jadex (Pokahr et al., 2005) and Jason (Bordini et al., 2007).

Finally, Prometheus details how the entities obtained during the design are transformed into concepts used in a specific agent-oriented programming language (JACK) (Winikoff, 2005). In this work, the Ecore language is used to develop the meta-model concepts specific to the Prometheus language used in the three phases of Prometheus method. A direct relation can be established among the previous elements (proposed Prometheus

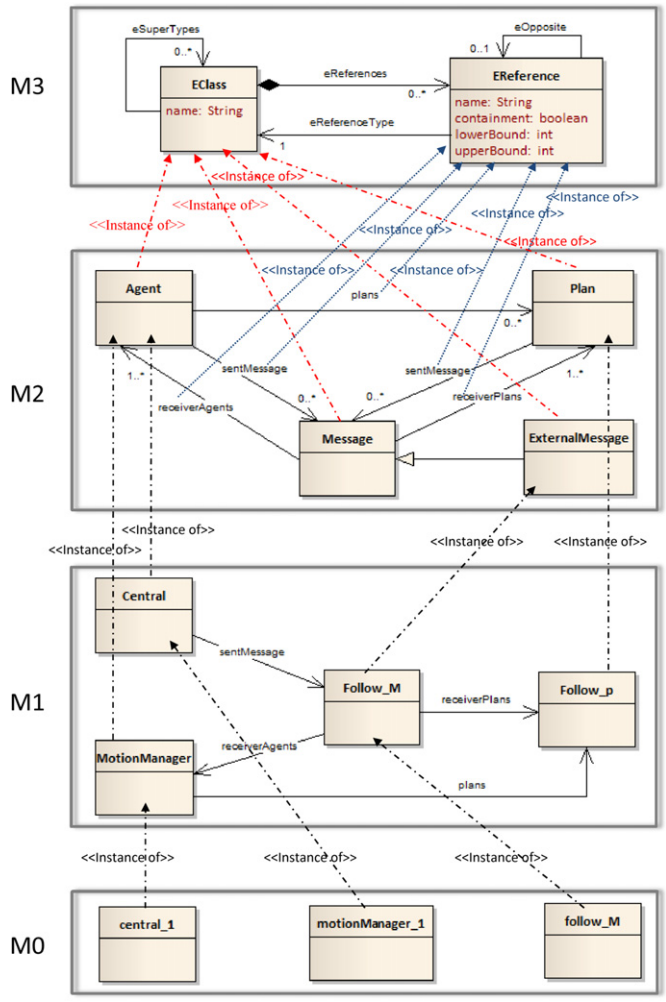


Fig. 2. MOF architecture.

Ecore meta-model, Prometheus multi-agent system model and JACK code) and the four layer architecture proposed by meta-object facility (MOF) (O.M.G., 2002). Indeed, as illustrated in Fig. 2, the meta-modeling architecture MOF, proposed by the Object Management Group, has four different layers (O.M.G., 2002). At the meta-meta-modeling layer (M3), a collection of primitives is offered to define meta-models at the M2 layer. That is, the meta-meta-model describes the properties of the meta-models. At the meta-modeling layer (M2), the defined meta-elements are used to instantiate the elements that make up models at M1 layer. At the modeling layer (M1) the application model is specified. Finally, it is at the M0 layer where instances of M1 models are described.

Now, Ecore is the meta-model used by Eclipse Metamodeling Framework (EMF) to define meta-models (Steinberg et al., 2009). Ecore is an implementation of the Essential MOF (EMOF) language, which is a subset of MOF. The EMOF model provides the minimal set of elements required to specify meta-models. The main elements of Ecore are *EClass*, *EReference* and *EAttribute*. An *EClass* instance defines an element of the EMF meta-model that describes a set of similar entities of the model. An *EClass* instance can be related to another *EClass* by means of unidirectional relationships named *EReferences* whose multiplicity is specified by means of attributes *lowerBound* and *upperBound*. Bi-directional relationships can be specified by using two EReferences and the corresponding *oppositeOf* attribute. Moreover, an *EClass* has *EAttributes* (EAttribute instances) to specify its properties.

The name of an Ecore meta-class is made up of the name of the element that it implements along with a prefix "E". There are two types of EReferences (García-Magariño et al., 2009): (a) *containments* describe composite relationship among two EClasses; and (b) *non-containments* describe that an *EClass* is related to another *EClass*.

As shown in Fig. 2, for the concrete case of Prometheus methodology, at M3 layer the Ecore meta-model definition is found. At M2 layer the Prometheus Ecore meta-model (PEMM) is defined. At M1 layer MAS models are specified according to the PEMM meta-model. Finally, at M0 layer the instances of the models at M1 layer are defined.

4. The Prometheus Model Editor

This section introduces the developed Prometheus Model Editor. This is the tool required to solve the problem described earlier. Remember that the code generation functionality provided by Prometheus is hard-code implemented. Therefore, the editor provides a better technological solution compared to Prometheus Design Tool (PDT). Again, this is thanks to the use of meta-modeling techniques. Moreover, as Prometheus is selected, the guidelines offered are helpful to experts in MAS development because they can convey their experience to other users by explaining why and how the different elements of the agent-based application have been obtained (Fernández-Caballero and Gascueña, 2010). In addition, Prometheus is also useful as it explicitly considers agent perceptions and actions as modeling elements.

Overall, the developed Model Editor is carried out following the three activities of the process shown at the top of Fig. 1. Firstly, a meta-model that specifies the modeling concepts of the Prometheus methodology is created. Afterwards, the Eclipse Graphical Modeling Framework (GMF) (Gronback, 2009) is used to develop the graphical editor that supports the creation of graphical models, in accordance with the Prometheus meta-model. Finally, the last step carries out the implementation of the functionality into code written in JACK agent programming language. Each one of these steps is detailed in the following sections. In Section 4.1, the defined PEMM meta-model is presented. Next, Section 4.2 introduces the graphical editor developed to create Prometheus models. Finally, in Section 4.3 it is illustrated how code generation mechanisms are integrated into the proposed tool.

4.1. Defining the meta-model

The starting point to describe the proposed Prometheus Ecore meta-model (PEMM) is its representation by means of the Prometheus UML meta-model (PUMM) (Dam et al., 2006) and the expertise acquired by modeling agent-based applications using Prometheus supporting tool PDT. The meta-model is created using directly the EMF's simple tree-based Ecore editor.

The following rules are used to name each PEMM element: (i) every EClass starts with an upper-case letter; (ii) EAttributes and EReferences start with a lowercase letter that denotes the referenced EClass. Regarding the graphical notation, PEMM elements represent the following images: (a) a rectangle split by means of two horizontal lines depicts an EClass; (b) an arrow illustrates EReference non-containment relationships; (c) a line with a diamond at one end represents a containment EReference; (d) a dashed line and a solid line denote two opposite EReferences; (e) the multiplicity of an EReference, established by means of lower-bound and upper-bound attributes, is always shown next to the line they are related to; and (f) an UML hierarchy symbol – a triangle that joins supertype and subtype – is used to illustrate the property ESuperType.

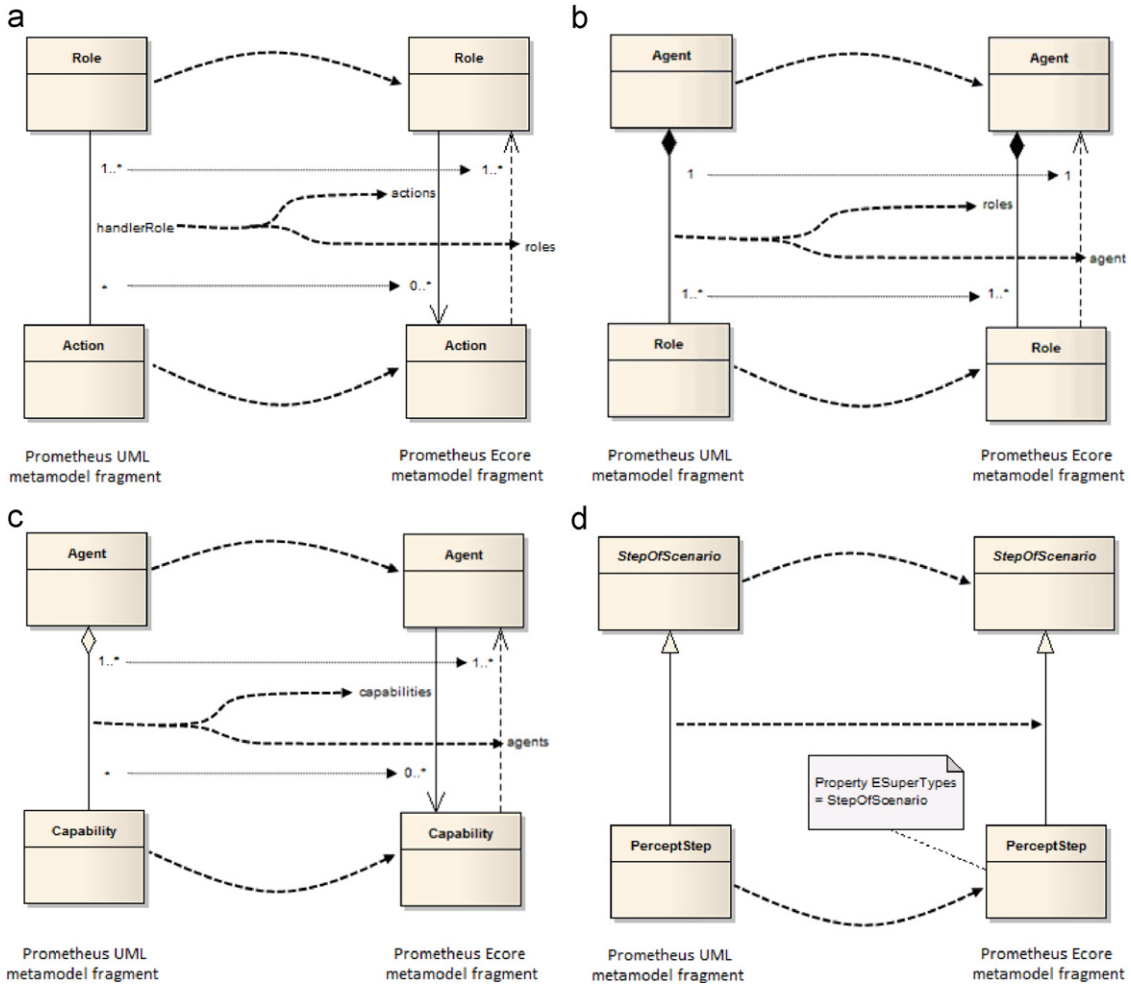


Fig. 3. Criteria to transform PUMM into PEMM.

The criteria adopted to translate PUMM into PEMM are explained next. Notice that fragments of the example shown in Fig. 3 are included in brackets for a better comprehension.

- An EClass is created for every meta-entity of the PUMM (*Role* and *Action* EClasses).
- With regard to meta-relationships, our steps – as described next – can be distinguished, depending on whether they are related to an association, aggregation, composition or generalization, respectively. As a rule, a pair of opposite EReferences is created for every meta-relationship between two meta-entities of the PUMM meta-model; in this way, the UML navigability is simulated in Ecore. Considering this fact:
  - Whenever there is an association (see Fig. 3a), the "source" EClass (*Role*) has a non-containment EReference (*actions*) to the "target" EClass (*Action*). Similarly, the "target" EClass (*Action*) has a non-containment EReference (*roles*) to the "source" EClass (*Role*).
  - If there is an aggregation (see Fig. 3b), the "source" (*Agent*) has an EReference containment (*roles*) to the "target" EClass (*Role*). Besides, the "target" EClass (*Role*) has a non-containment EReference (*agent*) to the "source" EClass (*Agent*).
  - Composition (see Fig. 3c) is dealt with in a similar way to association.
  - If there is a generalization (see Fig. 3d) the property ESuperTypes denoting the specialized class is used. For instance, in Fig. 3d the EClass *PerceptStep* has its property ESuperTypes set to *StepOfScenario*.

When a developer creates a Prometheus model in the tool, he/she does not create all the potential EReferences in M1, but only those identified as likely in PDT (see Table 2). The EMF framework automatically creates the corresponding opposite EReferences in M1. Although not absolutely necessary, the opposite relationships are included to facilitate the management of the models. After considering the above-mentioned facts, the final PEMM structure is the following (see Fig. 4):

- A "Prometheus" package contains all the elements of the meta-model.
- A model is made up of a collection of basic elements. This is represented by adding an EReference named "elements" to the EClass "Model". It has (1) its property EType set to the target EClass (BasicElement), (2) its lowerBound and upperBound properties set to 1 and −1, respectively, to denote that "Model" instances are related to one or more instances of "BasicElement", and (3) its property containment set to true to specify that it is an aggregation relationship. "Model" is not an abstract EClass; that is, it can be instantiated.
- BasicElement is an abstract EClass that has two EString attributes to specify the name and a brief description, respectively, of an entity of the model. Every concept used by the

Table 2
Connections between entities in each diagram used in prometheus.

Diagrams	Connections
Analysis overview	Actor ⟶ Percept; Actor ⟵ Action; Scenario ⟵ Percept; Scenario ⟶ Action
Scenario	Scenario ⟶ Action—it means first scenario has the second scenario in some step, it is to say, the second scenario is a sub-scenario of the first scenario
Goal overview	Goal ⟶ Goal; where the second goal is a subgoal of the first goal
System roles	Role ⟶ Goal; Role ⟶ Action; Percept ⟶ Role
Data coupling	Data ⟶ Role—it means Data is read by the Role; Role ⟶ Data—it means Data is written by the Role
Agent-role grouping	Agent ⟶ Role
System overview	Agent ⟶ Action; Percept ⟶ Agent; Agent ⟶ Data—it means Data is written by the Agent; Data ⟶ Agent—it means Data is read by the Agent; Agent ⟶ Message—it means Message is sent by the Agent; Message ⟶ Agent—it means Message is received by the Agent. Interaction protocols are able to include relations Actor ⟶ Percept; Actor ⟵ Action; and messages between agents
Agent overview	Capability ⟶ Action; Capability ⟶ Data; Capability ⟶ Message; Percept ⟶ Capability; Percept ⟶ Plan; Plan ⟶ Action; Plan ⟶ Message; Plan ⟶ Data; Message ⟶ Plan; Message ⟶ Capability; Data ⟶ Capability; Data ⟶ Plan
Capability overview	Capability ⟶ Action; Capability ⟶ Data; Capability ⟶ Message; Percept ⟶ Capability; Percept ⟶ Plan; Plan ⟶ Action; Plan ⟶ Message; Plan ⟶ Data; Message ⟶ Plan; Message ⟶ Capability; Data ⟶ Capability; Data ⟶ Plan

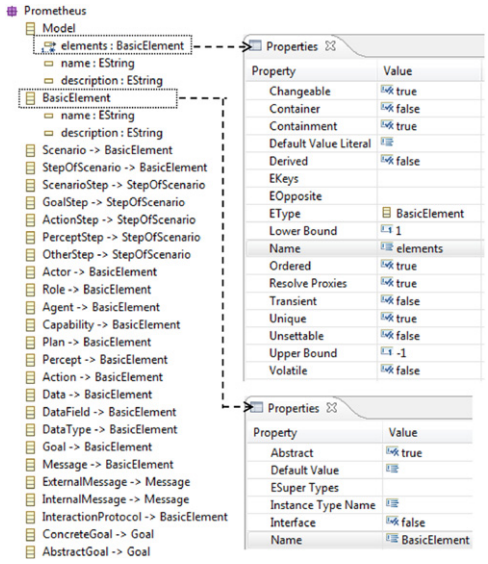


Fig. 4. Prometheus Ecore meta-model (PEMM).

Prometheus methodology (e.g. actor, agent, role, perception, etc.) is described as an EClass that directly or indirectly inherits from "BasicElement". The attributes defined in the root EClass (BasicElement) are also available to all the other EClasses included in the hierarchy.

4.2. Building the graphical editor

The Prometheus Graphical Editor is created through the guided process offered by the Graphical Modeling Framework (GMF) (Gronback, 2009), which allows us to create an editor from a Domain Model. Fig. 5 summarizes the models and steps of this process by means of an activity diagram. The models generated are:

(1) A Generator Model (.genmodel) is obtained by using the Prometheus Ecore meta-model (PEMM) (see dm:ecore in Fig. 5) described in Section 4.1. Specifically, a tree-based editor is created to specify models by using this Generator Model (.genmodel) (see gmd:genmodel in Fig. 5).
(2) Afterwards, a Graphical Model (.gmfgraph) (see gm:gmgraph in Fig. 5) that defines the figures, nodes and links in the diagrams is derived. This model specifies the elements of the model that have a graphical representation, as node, relation or attribute. Fig. 6a shows the wizard used to automatically create the gmfgraph file (partly shown in Fig. 6b) from the PEMM meta-model. For instance, Fig. 6a shows that it is only necessary to check the corresponding option to graphically describe *Agent* elements as nodes. Also, the name and the description of the *Agent* elements are represented in diagrams with label attributes. Finally, the relationships between *Agent* and *Goal* elements are described by means of graphical relations.
(3) Next, the Tooling Model (.gmftool) (see tm:gmftool in Fig. 5) that defines the palette of the Prometheus Graphical Editor is derived. This model specifies which elements are in the palette and how they are shown (as relationships or nodes). A wizard (see Fig. 7a) is also available for generating the Tooling Model (see Fig. 7b), also using the PEMM meta-model as input. For instance, the image (Agent.png) that is displayed when an agent instance is created in a Prometheus model is manually customized. On the contrary, images for links are displayed using a default image.
(4) Afterwards, links between the Domain Model (.ecore) (see dm:ecore in Fig. 5), the Graphical Model (.gmfgraph) and the Tooling Model (.gmftool) are recorded in the Mapping Model (.gmfmap) (see mm:gmfmap in Figs. 5 and 9). The process is supported by means of a new wizard that allows us to select each one of the previous models, and to customize which elements are nodes or links, or to change their properties by means of the dialog shown in Fig. 8.
(5) Finally, the Mapping Model (.gmfmap), along with the Generator Model (.genmodel), is used by another wizard to obtain a GMF Graphical Generator Model (.gmfgen, see Fig. 10).
(6) The Graphical Generator Model (.gmfgen) (see gen:gmfgen in Fig. 5) is employed by the Graphical Modeling Framework of Eclipse to automatically generate code (.java) for the Prometheus graphical editor.

Once the generated code is compiled as a new plug-in (Prometheus .diagram), it can be run to create new Prometheus models.

4.3. Generating code

If the implementation is carried out in a manual way, starting from the design, divergences between design and implementation can emerge. This makes the design less useful for maintenance and for the comprehension of the system (Bordini et al., 2007). In this case, there is a gap between the design models and the existing implementation languages. In order to bridge this gap,

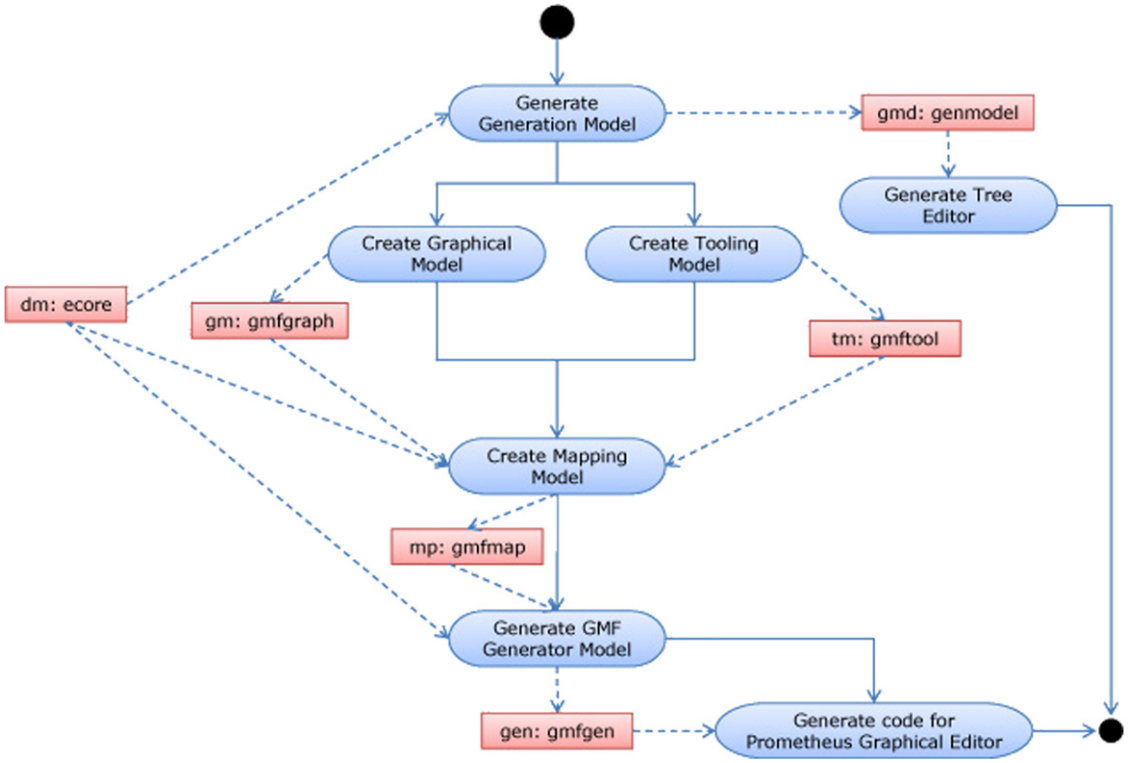


Fig. 5. Steps for building the graphical editor.

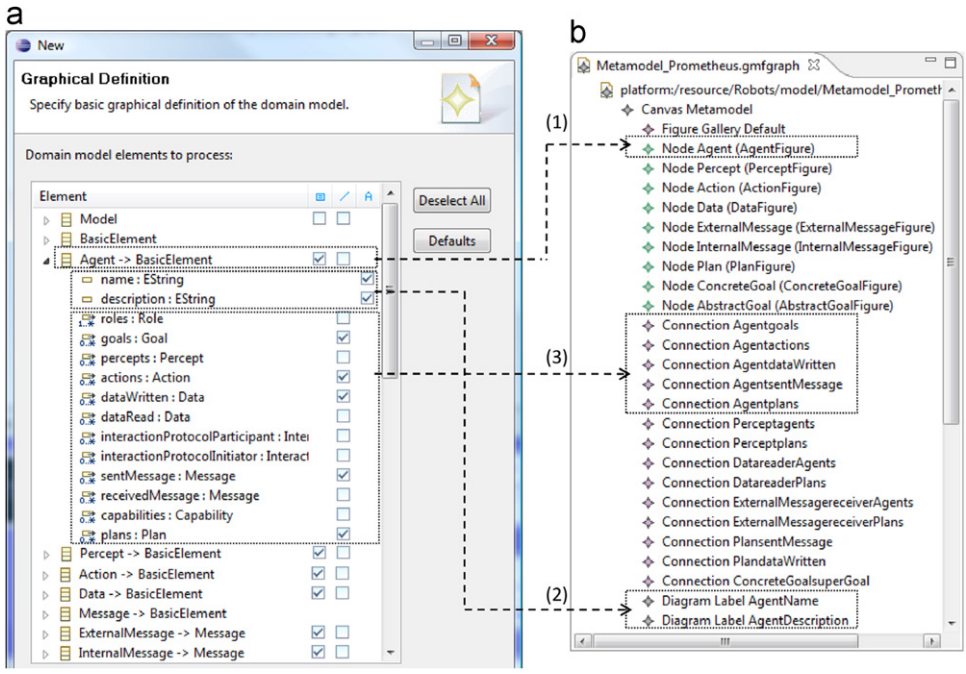


Fig. 6. Establishing mappings among the available options of (a) the graphical definition wizard and (b) the generated gmfgraph file.

techniques can be used that introduce refined design models directly implementable in a programming language. Alternatively, a dedicated agent-oriented programming language that provides constructs to implement the high-level design concepts (Dastani et al., 2004) could be used.

The Prometheus methodology follows the first approach. During the Detailed Design phase it offers models close to the concepts used in a specific agent-oriented programming language named JACK (Winikoff, 2005). Hence, the entities obtained during the design can directly be transformed into concepts used in JACK. Table 3 shows which Prometheus entities are translated into their equivalent JACK concepts. Note that some entities (*Actor*, *Goal*, *Protocol*, *Role*, *Scenario*) are not transformed into JACK concepts because they are only used during the design. In addition, the *Action* concept is not transformed into a JACK specific concept, but it can be implemented in the associated agent as a method.

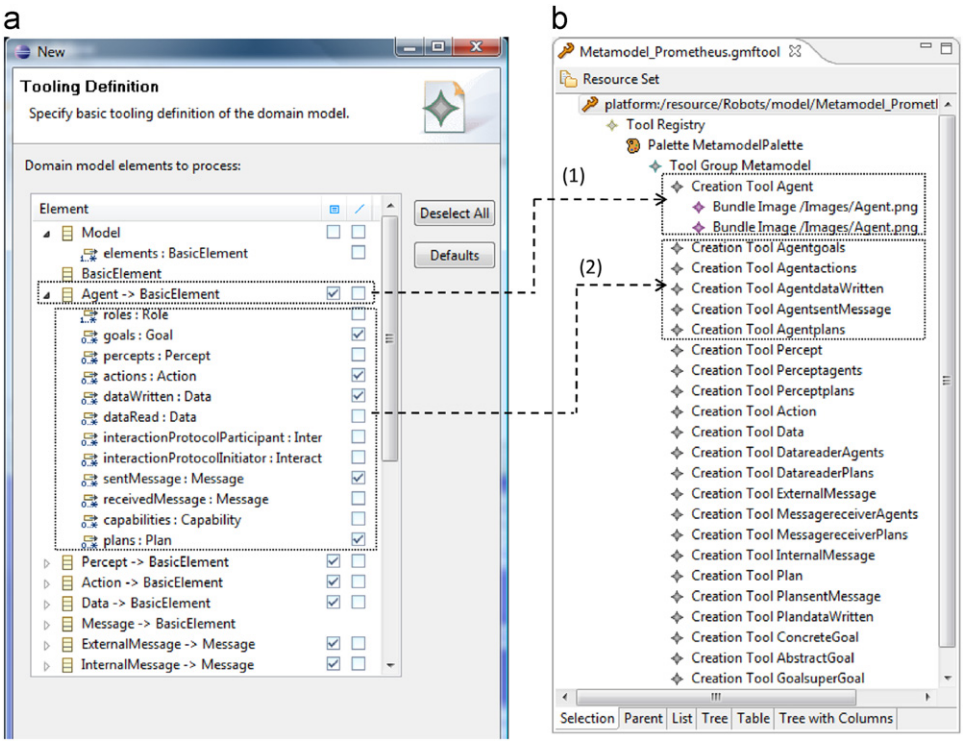


Fig. 7. (a) Tooling definition wizard; (b) gmftool file.

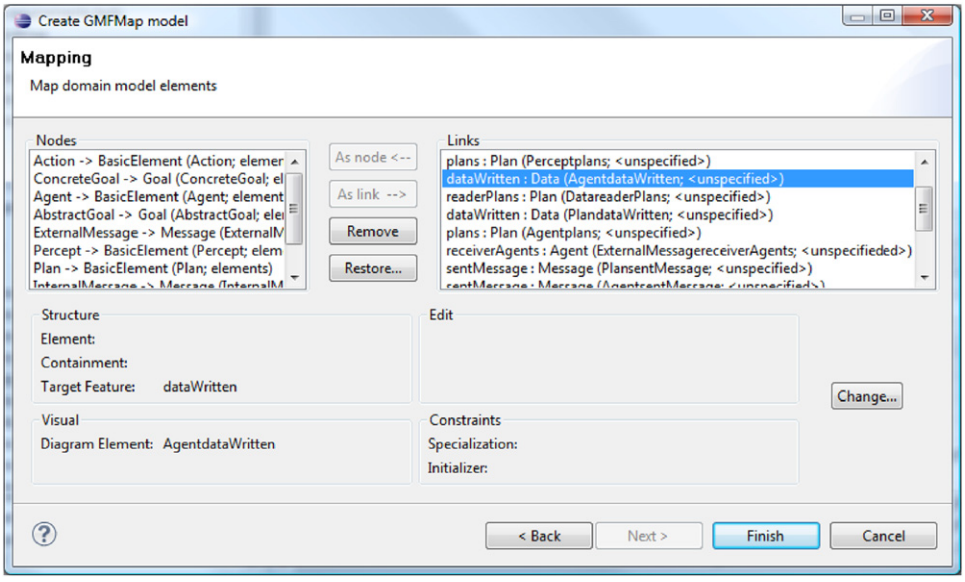


Fig. 8. Map domain model elements.

Besides, JET (Java Emitter Templates) (Vogel, 2009) is used in the editor to provide code generation abilities. JET is an engine for generating code through templates. It has been developed as an Eclipse plug-in that belongs to EMF. One advantage of JET is that it uses a subset of the syntax of Java Server Pages (JSP) to define templates for code generation. This way, developers who are used to this technology can learn it very easily. In addition, XPath may be used to access and navigate throughout the nodes of the source model.

Basically, the process followed to develop the proposed code generator is described in Fig. 11. First, a template is created by hand for each type of entity (*AgentTemplate*, *DataTemplate*, *EventTemplate*, *PerceptTemplate*, *PlanTemplate* and *CapabilityTemplate*), obtaining the related java classes in an automatic way. Table 4 shows an *AgentTemplate* fragment included in AgentTemplate.javajet file. Next, the *JackCodeGenerator* class is developed. It is in charge of instantiating the templates by using their related classes and of generating the code of the MAS using the Prometheus model.

5. Case of study: a multi-robot multisensory surveillance system

The authors of this paper have a long experience in the development of surveillance systems (Fernández-Caballero et al., 2010; Pavón et al., 2007). Therefore, the proposal has been validated in

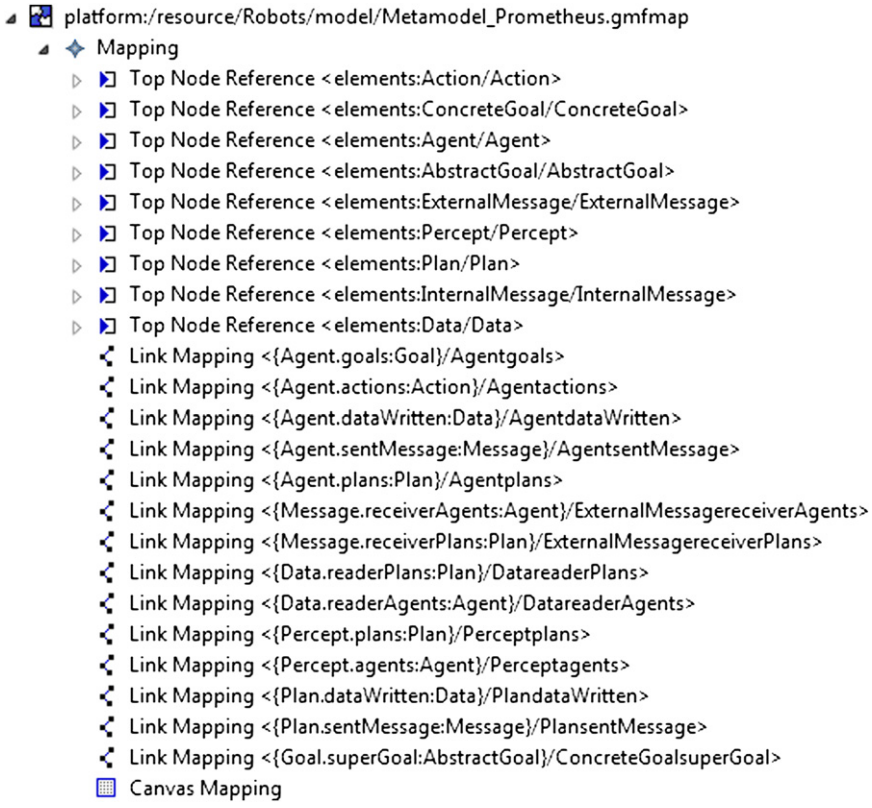


Fig. 9. gmfmap file.

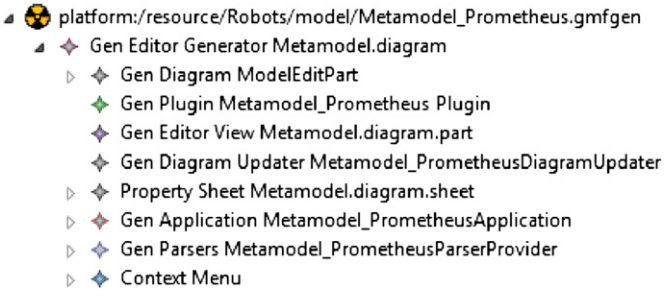


Fig. 10. gmfgen file.

Table 3
Mapping Prometheus and JACK concepts.

Prometheus entity	JACK concept
Agent	Agent
Capability	Capability
Percept	Event
Message	Event
Plan	Plan
Data	Beliefset
Action	Method

real-world projects related to the development of agent-based surveillance systems (Gascueña and Fernández-Caballero, 2011). Indeed, one of the developed systems focused on the detection and tracking of objects by means of a variable number of mobile robots that are under the control of software agents (Gascueña and Fernández-Caballero, 2011). The case of study presented in this paper introduces the mentioned multi-robot surveillance system developed by means of the Prometheus Model Editor.

The developed system introduces a MAS at several levels. Firstly, a single mobile robot is composed of different kinds of sensors. Such a multisensory system can be developed as a MAS where there is an agent for each physical sensor of the robot (Vinyals, 2011). There are reactive agents that gather information provided by the devices. Other agents collaborate to satisfy the global goal of the robot. Moreover, the problem offers timing functionality that allows us to consider that the system inherits autonomy and proactivity—which are properties of the agents (Jennings et al., 1993). At a second level, the case of study considers the existence of a human guard and a number of patrolling mobile robots.

Due to limitation in space, the case of study is only explained at the multisensory MAS level. Thus, the process that the proposed multisensory system applies is depicted in Fig. 12. Each robot is moving randomly around the environment (state *wandering*) while the collected images are shown to the human guard. After some time has elapsed (*Timer_P*), a robot stops to analyze the images captured until that moment (state *detecting*). There are two likely alternatives for each robot in this state:

- If the robot detects a movement (Moreno-Garcia et al., 2010; Delgado et al., 2010) then (1) information about the detected blob is obtained, and (2) the guard is warned to decide whether the robot should track the blob (*Follow_P*) or not (*Timer_P*).
- Alternatively, the tracking process (*Follow_P*) is started by the guard if he/she perceives that something is moving in the environment thanks to the images displayed on his/her interface. In that case, the guard orders (*Detect_P*) to analyze the images to check whether there is or not movement. If the image analysis does not detect movement, then the robot goes on moving randomly. Otherwise, the guard orders to start the tracking process of the blob (*Follow_P*).

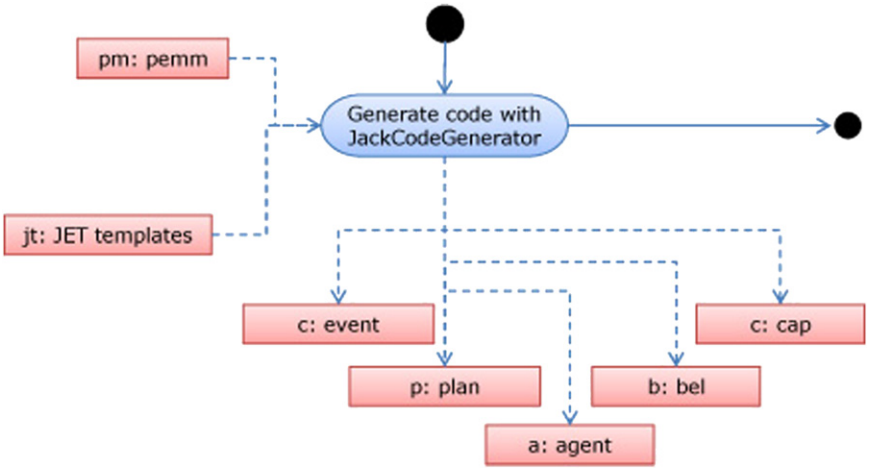


Fig. 11. Jack code generator based on JET.

Table 4
AgentTemplate fragment.

```
(1)  <      %@    jet
(2)   package="CodeGenerator"
(3)     imports="java.util.*
(4)     Metamodel.*
(5)         java.util.ListIterator
(6)         java.util.Iterator
(7)         java.util.ArrayList"
(8)    class="AgentImplementator"
(9)  %>
(10) <%  MetamodelFactory MF=MetamodelFactory.eINSTANCE;
(12)       MetamodelPackage MP=MetamodelPackage.eINSTANCE;
(13)       Model M=MF.createModel();
(14)       Agent A=MF.createAgent();
(15)       ArrayList<Agent> Ags=new ArrayList();
(16)       A=(Agent)argument;
(17) %>
(18) package agents;
(19) import capabilities.*;
(20) import plans.*;
(21) import events.*;
(22) import data.*;
(23) public agent <%=((Agent)argument).getName()%>extendsAgent
```

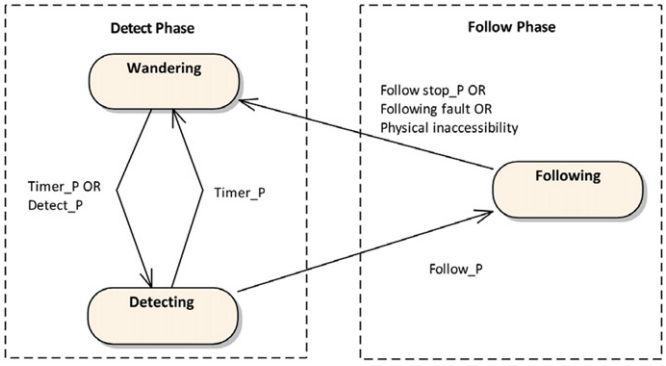


Fig. 12. The robot's states.

In order to achieve a successful tracking of an object (state *following*), the images are captured, displayed, and analyzed continuously so that blob information is obtained. The object is followed until this tracking phase finishes. The condition is satisfied by three different reasons: (1) the guard has decided not to continue following the target (*Follow stop_P*); (2) the target is out of the field of view (*Following fault*); or (3) it is impossible to follow the target (*Physical inaccessibility*) because there is some physical obstacle in the environment (for example, the object goes upstairs). In these cases, the robot starts wandering again.

The next three subsections are structured according to the three steps shown at the bottom of Fig. 1. Firstly, a Prometheus model is specified by using the developed graphical Prometheus Editor (see Section 5.1). Afterwards, intermediate code is automatically generated (see Section 5.2). Finally, the implementation is completed (see Section 5.3).

5.1. Step 1: Prometheus model

Using as input the above problem description, a MAS is defined using the developed Prometheus Editor. Firstly, the agents and goals of the proposed surveillance system are specified. As Fig. 13 shows, *Camera*, *Wheels*, *Bumper* and *Sonar* agents pursue to satisfy the goal of capturing an image, moving a robot, controlling collisions and avoiding obstacles, respectively. All these agents are identified to control each one of the physical sensors in the surveillance system. In addition, other agents and their corresponding goals are specified to control the previous ones. Specifically, as illustrated in Fig. 13, *ImageManager* is responsible for goals related to managing the images sent by the *Camera* agent (analyze images, get blob information and show field of view),

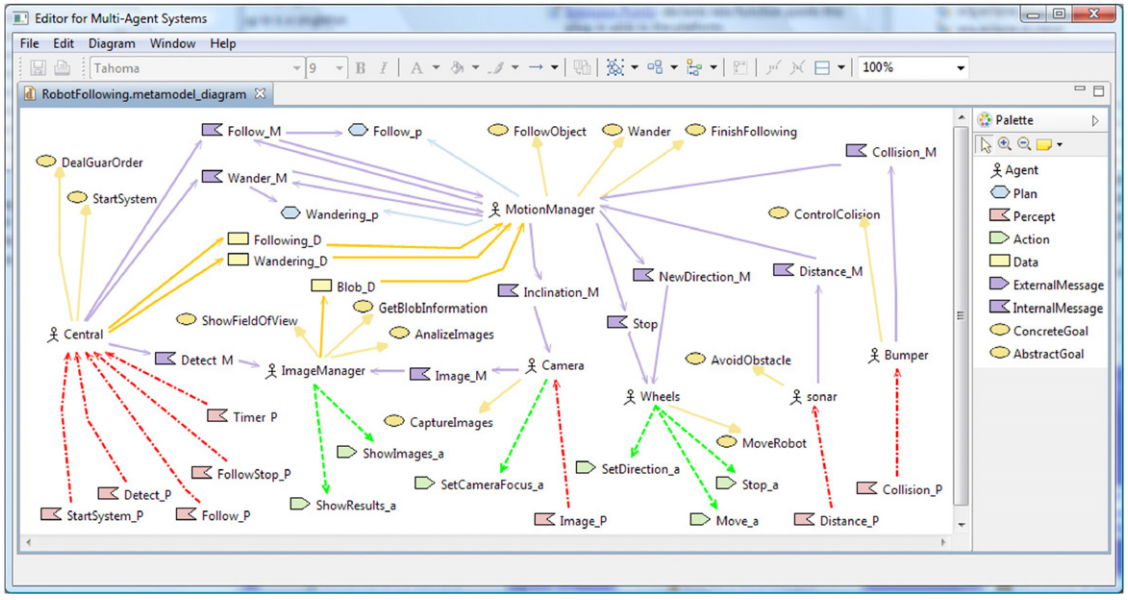


Fig. 13. What the Prometheus Graphical Editor looks like while it shows the (part of) Prometheus model for the surveillance system.

whereas *MotionManager* is responsible for those goals related to control the robot movement according to its actual state (wander, follow object and finish following). Finally, a *Central* agent pursues to control the interaction with the user of the application. Therefore, it is responsible for satisfying goals of starting the system and managing the commands issued by the guard. In short, there are seven agent instances for each robot that patrols the environment.

In addition, interactions between the system and the environment are described. On the one hand, the information that comes into the system from the environment is identified (percepts). It corresponds to impacts detected by the bumper device (*Collision_P*), images captured by the camera (*Image_P*), distance to obstacles/targets perceived by the sonar (*Distance_P*), time percept issued by a timer (*Timer_P*), and orders issued by the guard to start the system (*StartSystem_P*) and to control the change of the system state (*Detect_P*, *Follow_P*, *FollowStop_P*). On the other hand, every output produced by the system and observable by an external observer is also identified (actions). It corresponds to the camera movements carried out based on the tilt, pan and zoom parameters provided (*SetCameraFocus_a*), commands to control wheel motion (*SetDirection_a*, *Stop_a*, *Move_a*), and an action *ShowImages_a* to show the images captured. *ShowRsults_a* also highlights with a square the image regions where movement has been detected.

Once the agents, goals and interaction with the environment are specified, the interaction among these agents are defined. In short, the proposed MAS (see Fig. 13) presents a hierarchical communication among agents that allows achieving the global goal: the surveillance of the environment. As can be observed, *Central* sends messages to *Motion Manager* and *Image Manager* depending on the robot's state. The *Motion Manager* sends messages to the *Camera* and *Wheels* agents to move the robot's mobile components. Moreover, it receives messages from the *Bumper* and *Sonar* agents containing the collected information. Finally, the *Image Manager* receives messages from the *Camera* agent with the captured images to show them or to detect motion.

To complete the specification of the Prometheus Model of the surveillance system, the planning of the agents is specified. For instance, the *Follow_p* and *Wandering_p* plans are specified for the *Motion Manager* agent (see Fig. 13) in order to establish the procedures used by the robot when it follows an object and wanders around the environment, respectively. The *Follow_p* plan is triggered due to three different reasons:

(1) The *Motion Manager* agent sends to itself a *Follow_M* message that contains "continue follow" to continue the *following* process.
(2) The *Central* agent sends to the *Motion Manager* agent a *Follow_M* message that contains "start follow" to start the *following* process.
(3) The *Central* sends to the *Motion Manager* agent a message *Follow_M* with information "stop follow" to stop the *following* process.

5.2. Step 2: intermediate code

Once the Prometheus model of the surveillance system is defined, the tool proposed in this paper is used to generate code, according to the process depicted in Fig. 14. The process can be summarized as follows. The JackCodeGenerator class is executed to generate a JACK skeleton code. The Prometheus model and the JET templates (see Fig. 11) are used as input. This automatically generates a JACK folder that includes several subfolders (agents, capabilities, data, events, plans). The agent sub-folder contains a file ".agent" for each agent entity included in the model. The filename is the name of the agent in the Prometheus model. The same is applicable to capabilities, data, messages and plan entities created from the model.

5.3. Step 3: final code

As shown in Fig. 14, the JACK Development Environment (JDE) tool is used to import the intermediate code generated by the proposed tool. Then, the imported skeleton code is completed and some new java classes are created (e.g. to set up graphical user interfaces, to create instances of agents, and so on). For example, Fig. 15(a) shows the JACK skeleton code related to the *Follow_M* message entity. Also, the final code for *Follow_M* is depicted in Fig. 15(b). Notice that a more descriptive name (methodName is always set by default) is provided for the posting method. Moreover, a variable is declared to put the contained information into the message. Finally, all this code is compiled so that the application can be run.

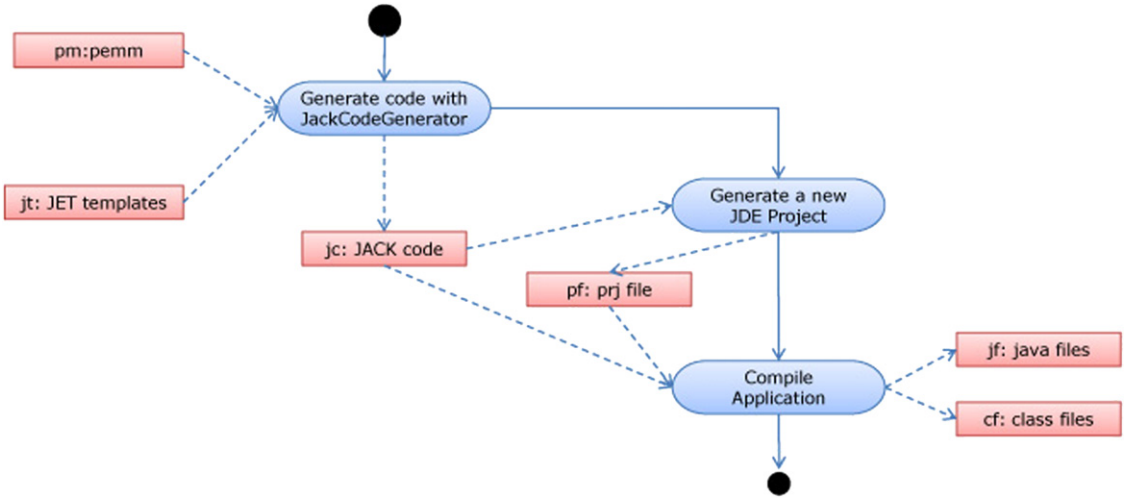
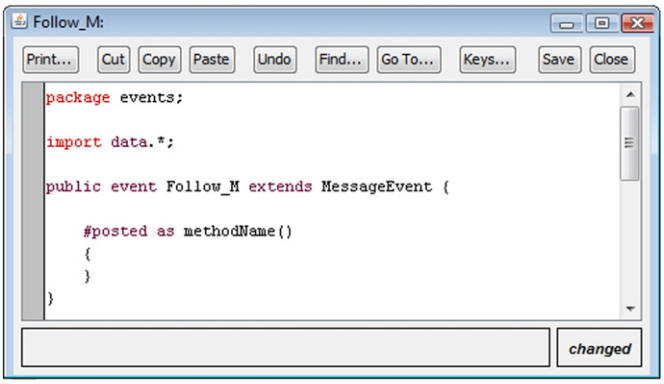


Fig. 14. Code generation process.

a



b

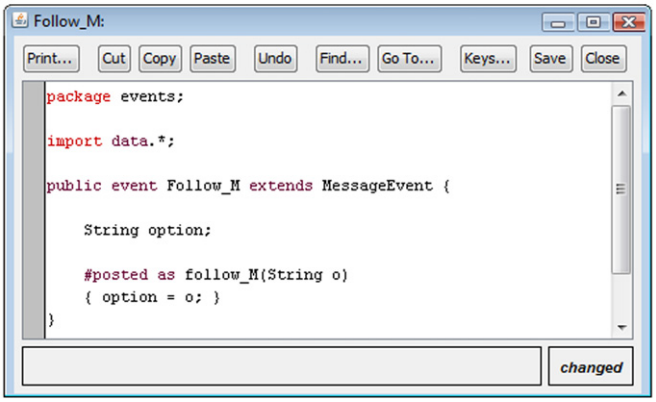


Fig. 15. Follow_M message. (a) JACK skeleton code. (b) Final code.

6. Conclusions and future work

The introduction of MDE in the software development process is being more and more embraced by both academics and practitioners. The main reason is the benefit provided in terms of productivity, portability, inter-operability and maintenance. Taking account of the importance of these quality requirements for any developed system, we believe that MAS can also take advantage of MDE to improve the quality of their development. The concept of model is already present in most of the agent methodologies. Therefore, the use of MDE techniques for agent-oriented software development (AOSD) emerges in a natural way. Indeed, we are not introducing new concepts, but we offer new technological solutions for supporting the existing concepts. Moreover, although the learning curve of MDE tools for AOSD can be steep at first, it is offset by the benefits that they provide in terms of quality of the developed products.

With this idea in mind, this paper has shown how to build a tool for developing MAS through exploiting MDE techniques. Specifically, a Model Editor that provides support to an AOSD methodology named Prometheus has been developed. The Model Editor is fully integrated in Eclipse; this way, it uses the facilities provided by model-based development technologies. As stated, the functionality offered by the proposed tool is not totally new. Indeed, the definition of models according to the Prometheus methodology and the generation of code in JACK are also provided by PDT. However, the tool exploits important meta-modeling techniques. For instance, it is able to build the complete modeling environment. It allows us to modify the modeling environment in a very easy manner, just by changing the meta-model. Also, one of the main functionalities of the proposed Model Editor is the generation of code by means of templates. This facility helps to generate applications in an automatic way.

In addition, the Model Editor has been validated in the context of several real-world projects which focus on multi-robot multi-sensory surveillance systems. Only a partial description of the system has been sketched in this paper due to space limitations. However, the work has shown that the use of MDE improves the development process of multi-agent surveillance systems. For example, the specification of the model created by using the developed editor is not tied to any specific agent implementation language. In this work, templates have been created to automatically generate code in JACK language, which uses the BDI model to represent the internal structure of its agents. However, it is feasible to create new templates for generating code in other languages (e.g. Jadex and Jason) that also support the BDI agent architecture. Thus, code may be generated in different languages with a same specification. In short, the use of MDE techniques allow us to face up technological changes more easily.

One of our future works is related to the improvement of the proper Model Editor. As stated in Section 4, Prometheus uses several kinds of sub-meta-models to provide complementary views of the system. These sub-meta-models make up the

Prometheus meta-model. Currently, the Model Editor provides only one view to specify the Prometheus meta-model. This way its usability is reduced. That is why we are working on the implementation of several kinds of views according to the different perspectives of the Prometheus meta-model.

Acknowledgments

This work is partially supported by the Spanish Ministerio de Ciencia e Innovación through the TIN2010-20845-C03-01 and CENIT A-78423480 grants, and by the Junta de Comunidades de Castilla-La Mancha through the PII2I09-0069-0994 and PEII09-0054-9581 grants.

References

Agent Oriented Software Pty. Ltd., 2008. JACK Intelligent Agents Development Environment Manual. Available at 〈http://www.aosgrp.com/documentation/jack/JDE_Manual_WEB〉.

Bauer, B., Müller, J.P., Odell, J., 2001. Agent UML: a formalism for specifying multiagent software systems. Int. J. Software Eng. Knowl. Eng. (IJSEKE) 11 (3), 207–230.

Bauer, B., 2001. UML class diagrams revisited in the context of agent-based systems. Agent-Oriented Software Engineering II. Lecture Notes in Computer Science, vol. 2222; 2001, pp. 101–118.

Bauer, B., Odell, J., 2005. UML 2.0 and agents: how to build agent-based systems with the new UML standard. Eng. Appl. Artif. Intell. 18, 141–157.

Bellifemine, F., Caire, G., Greenwood, D., 2007. Developing Multi-Agent Systems with JADE. John Wiley and Sons.

Bernon, C., Greizes, M.P., Peyruqueou, S., Picard, G., 2003. ADELFE, a methodology for adaptive multi-agent systems engineering. In: Third International Workshop Engineering Societies in the Agents World (ESAW), Lecture Notes in Artificial Intelligence, vol. 2577, pp. 156–169.

Bordini, R.H., Dastani, M., Winikoff, M., 2007. Current issues in multi-agent systems development. In: 7th Annual International Workshop on Engineering Societies in the Agents World. Lecture Notes in Artificial Intelligence, vol. 4457, pp. 38–61.

Bordini, R.H., Hübner, J.F., Wooldridge, M., 2007. Programming Multi-agent Systems in AgentSpeak using Jason. John Wiley and Sons.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A., 2004. Tropos: an agent-oriented software development methodology. Autonomous Agents Multi-Agent Syst. 8, 203–236.

Caire, G., Porta, M., Quarantotto, E., Sacchi, G., 2008. Wolf—an Eclipse plug-in for WADE. In: Proceedings of 17th IEEE Workshops on Enabling Technologies Infrastructure for Collaborative Enterprises (WETICE), pp. 26–32.

Caire, G., Quarantotto, E., Sacchi, G., 2009. WADE: an open source platform for workflows and agents. In: Proceedings of the Second Multi-Agent Logics, Languages, and Organisations Federated Workshops, pp. 69–72.

Cervenka, R., Trencansky, I., 2007. The Agent Modeling Language—AML: A Comprehensive Approach to Modeling Multi-Agent Systems. Whitestein Series in Software Agent Technologies and Autonomic Computing.

Dam, K.H., Winikoff, M., Padgham, L., 2006. An agent-oriented approach to change propagation in software evolution. In: Proceedings of the Australian Software Engineering Conference (ASWEC)IEEE Computer Society, pp. 309–318.

Dastani, M., Hulstijn, J., Dignum, F., Meyer, J., 2004. Issues in multiagent system development. In: Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), pp. 922–929.

Delgado, A.E., López, M.T., Fernández-Caballero, A., 2010. Real-time motion detection by lateral inhibition in accumulative computation. Eng. Appl. Artif. Intell. 23 (1), 129–139.

Deloach, S., Wood, M., Sparkman, C., 2001. Multiagent system engineering. Int. J. Software Eng. Knowl. Eng. 11 (3), 231–258.

Enterprise Architect. 〈http://www.sparxsystems.com.au/〉. Last visited October 2010.

Fernández-Caballero, A., Castillo, J.C., Martínez-Cantos, J., Martínez-Tomás, R., 2010. Optical flow or image subtraction in human detection from infrared camera on mobile robot. Robot. Autonomous Syst. 58 (12), 1273–1281.

Fernández-Caballero, A., Gascueña, J.M., 2010. Developing multi-agent systems through integrating Prometheus, INGENIAS and ICARO-T. Agents and Artificial Intelligence. International Conference, ICAART 2009 67, 219–232 (Revised Selected Papers. Communications in Computer and Information Science).

Fuentes-Fernández, R., García-Magariño, I., Gómez-Rodríguez, A.M., González-Moreno, J.C., 2010. A technique for defining agent-oriented engineering processes with tool support. Eng. Appl. Artif. Intell. 23 (3), 432–444.

García-Magariño, I., Fuentes-Fernández, R., Gómez-Sanz, J.J., 2009. Guideline for the definition of EMF meta-models using an entity-relationship approach. Inf. Software Technol. 51 (8), 1217–1230.

Gascueña, J.M., Fernández-Caballero, A., 2011. On the use of agent technology in intelligent, multi-sensory and distributed surveillance. Knowledge Eng. Rev. 26 (2), 191–208.

Gascueña, J.M., Fernández-Caballero, A., 2011. Agent-oriented modeling and development of a person-following mobile robot. Expert Syst. Appl. 38 (4), 4280–4290.

Gasevic, D., Djuric, D., Devedzic, V., 2009. Model Driven Engineering and Ontology Development, 2nd ed. Springer-Verlag.

Georgeff, M.P., Pell, B., Pollack, M.E., Tambe, M., Wooldridge, M., 1998. The belief-desire-intention model of agency. In: Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th International Workshop, pp. 1–10.

Gómez-Sanz, J.J., Fuentes, R., Pavón, J., García-Magariño, I., 2008. INGENIAS development kit: a visual multi-agent system development environment. In: Proceedings of 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1675–1676.

Gronback, R.C., 2009. Eclipse Modeling Project, A Domain-Specific Language (DSL) Toolkit. Addison-Wesley.

Hahn, C., Madrigal-Mora, C., Fischer, K., 2009. A platform-independent meta-model for multiagent systems. Autonomous Agents Multi-Agent Syst. 18 (2), 239–266.

Henderson-Sellers, B., Giorgini, P., 2005. Agent-Oriented Methodologies. Idea Group.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. The Unified Software Development Process. Addison-Wesley.

Jennings, N.R., Varga, L.Z., Aarnts, R.P., Fuchs, J., Skarekc, P., 1993. Transforming standalone expert systems into a community of cooperating agents. Eng. Appl. Artif. Intell. 6 (4), 317–331.

Jha, S., Massan, M., 2002. Building agents for rule-based intrusion detection system. Comput. Commun. 25, 1366–1373.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: a model transformation tool. Sci. Comput. Programming 72, 31–39.

Karageorgos, A., Mehandjiev, N., Weichhart, G., Hämmerle, A., 2003. Agent-based optimisation of logistics and production planning. Eng. Appl. Artif. Intell. 16 (4), 335–348.

Kelly, S., Lyytinen, K., Rossi, M., 1996. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In: Advanced Information Systems Engineering. Lecture Notes in Computer Science, vol. 1080. pp. 1–21.

Kleppe, A., Warmer, J., Bast, W., 2003. MDA Explained: The Model Driven Architecture ™: Practice and Promise. Addison-Wesley.

Leitão, P., 2009. Agent-based distributed manufacturing control: a state-of-the-art survey. Eng. Appl. Artif. Intell. 22 (7), 979–991.

Mattsson, A., Lundell, B., Lings, B., Fitzgerald, B., 2009. Linking model-driven development and software architecture: a case study. IEEE Trans. Software Eng. 35 (1), 83–93.

Molesini, A., 2008. Meta-models Environment and Layers: Agent-oriented Engineering of Complex Systems. Ph.D. Thesis, Alma Mater Studiorum-Università di Bologna.

Moreno-Garcia, J., Rodriguez-Benitez, L., Fernández-Caballero, A., López, M.T., 2010. Video sequence motion tracking by fuzzification techniques. Appl. Soft Comput. 10 (1), 318–331.

Oldevik, J., 2009. MOFScript User Guide. Available in 〈http://www.eclipse.org/gmt/mofscript/doc/〉.

Object Management Group, 2002. Meta object facility (MOF) specification—version 1.4, April 2002. Available in 〈http://www.omg.org/spec/MOF/1.4/〉.

Padgham, L., Winikoff, M., 2004. Developing Intelligent Agents Systems: A Practical Guide. John Wiley and Sons.

Padgham, L., Thangarajah, J., Winikoff, M., 2008. Prometheus design tool. In: Proceedings of the 23rd AAAI Conference on Artificial Intelligence, pp. 1882–1883.

Pavón, J., Gómez-Sanz, J.J., Fuentes, R., 2006. Model driven development of multi-agent systems. In: Model Driven Architecture—Foundations and Applications, Lecture Notes in Computer Science, vol. 4066, pp. 284–298.

Pavón, J., Gómez-Sanz, J.J., Fernández-Caballero, A., Valencia-Jiménez, J.J., 2007. Development of intelligent multi-sensor surveillance systems with agents. Robotics Autonomous Syst. 55 (12), 892–903.

Pokahr, A., Braubach, L., Lamersdorf, W., 2005. Jadex: a BDI reasoning engine. In: Multi-Agent Programming Languages, Platforms Applications, pp. 149–174.

Posadas, J.L., Poza, J.L., Simó, J.E., Benet, G., Blanes, F., 2008. Agent-based distributed architecture for mobile robot control. Eng. Appl. Artif. Intell. 21 (6), 805–823.

Pressman, R.S., 2010. Software Engineering: A Practitioner's Approach, 7th ed. McGraw-Hill.

Rodríguez-Seda, E.J., Troy, J.J., Erignac, C.A., Murray, P., Stipanovic, D.M., Spong, M.W., 2010. Bilateral teleoperation of multiple mobile agents: coordinated motion and collision avoidance. IEEE Trans. Control Syst. Technol. 99 (1), 984–992.

Ronald, N., Sterling, L., 2005. A BDI approach to agent-based modelling of pedestrians. In: Proceedings of 19th European Conference on Modelling and Simulation (ECMS), pp. 169–174.

Rumbaugh, J., Jacobson, I., Booch, G., 2004. The Unified Modeling Language Reference Manual, 2nd ed. Addison-Wesley.

Smolík, P.C., 2006. Mambo Meta-modeling Environment. Doctoral Thesis. Brno University of Technology, Czech Republic.

StarUML 〈http://staruml.sourceforge.net/en/〉. Last visited October 2010.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. Eclipse Modeling Framework, 2nd ed. Addison-Wesley.

Vinyals, M., Rodriguez-Aguilar, J.A., Cerquides, J., 2011. A Survey on Sensor Networks from a Multiagent Perspective. Comput. J. 54 (3), 455–470.

Vogel, L., 2009. Java Emitter Template (JET)—Tutorial. Available at 〈http://www.vogella.de/articles/EclipseJET/article.html〉.

Warwas, S., Hahn, C., 2009. The DSML4MAS development environment. In: Proceedings of 8th International Conference on Autonomous Agents and Multi-Agent Systems, pp. 1379–1380.

Winikoff, M., 2005. Jack™ intelligent agents: an industrial strength platform. In: Multi-Agent Programming Languages, Platforms Applications, pp. 175–193.

Wooldridge, M., Jennings, N.R., Kinny, D., 2000. The Gaia methodology for agent-oriented analysis and design. J. Autonomous Agents Multi-Agent Syst. 3 (3), 285–312.