VigilAgent for the Development of Agent-Based Multi-robot Surveillance Systems

José M. Gascueña[1], Elena Navarro[1], and Antonio Fernández-Caballero[1,2]

[1] Universidad de Castilla-La Mancha, Departamento de Sistemas Informáticos & Instituto de Investigación en Informática de Albacete (I3A), 02071-Albacete, Spain
{JManuel.Gascuena,Elena.Navarro,Antonio.Fdez}@uclm.es
[2] Instituto de Desarrollo Industrial (IDI), Computer Vision Research Lab, c/ Investigación s/n, 02006-Albacete, Spain

Abstract. Usually, surveillance applications are developed following an ad-hoc approach instead of using a methodology to guide stakeholders in achieving quality standards expected from commercial software. To solve this gap, our conjecture is that surveillance applications can be fully developed from their initial design stages by means of agent-based methodologies. Specifically, this paper describes the experience and the results of using a multi-agent systems approach according to the process provided by the *VigilAgent* methodology to develop a mobile robots surveillance application.

Keywords: Multi-agent systems, Agent-based software engineering, Mobile robots, Surveillance.

1 Introduction

At present, it is becoming more and more common to use mobile robots in assisting humans in surveillance tasks. The potential benefits are saving costs, not exposing humans to dangerous situations, and performing surveillance routines more effectively since humans get bored when performing tasks that last long working hours [1]. Sometimes, the capabilities of a single robot are enough to carry out a task (e.g., detecting and following an intruder [4], [9]). However, it is also usual to face scenarios requiring the collaboration among multiple mobile robots to achieve a common goal such as exploring or covering a larger area [13]. These complex tasks can successfully be solved through incorporating multi-agent systems technologies [15] due to the characteristics inherent to agents such as (1) autonomy, necessary to achieve individual and collective goals, and, (2) sociability, indispensable to form robot teams performing cooperatively.

The previous remarks demonstrate the feasibility of using well-known methods, techniques and tools in Agent-Oriented Software Engineering (AOSE) [17] to improve the development of robotic surveillance systems. In fact, agent technology has already been used in several surveillance systems [10]. However, to the best of our knowledge they are not developed following a methodology. Our proposal is to introduce techniques proposed in AOSE to produce well-documented

J. O'Shea et al. (Eds.): KES-AMSTA 2011, LNAI 6682, pp. 200–210, 2011.

surveillance applications from requirements to implementation [14], [7], [5]. The exploitation of methodologies in software development provides two main advantages. Firstly, methodologies allow stakeholders to share the same terminology and development process. Second, methodologies are useful to convey more easily the knowledge acquired along a project for the development of future projects showing similar features. The problem in AOSE is that there is not a unique and specific useful methodology without some level of customization [3]. This paper focuses on describing how the process of *VigilAgent* methodology is applied to face the problem of a collection of robots patrolling around a surveillance environment. This methodology has not been developed from scratch as it reuses fragments from Prometheus [11] and INGENIAS [12] methodologies for modelling, and ICARO-T framework [6] for implementation purposes.

The rest of the paper is organized as follows. In section 2, an overview of the phases of the *VigilAgent* methodology is offered. A justification of why Prometheus, INGENIAS and ICARO-T technologies are integrated is provided. Then, section 3 introduces a case study used to demonstrate the applicability of *VigilAgent*. Finally, section 4 offers some conclusions.

2 Overview of the VigilAgent Methodology

The five phases of *VigilAgent* are briefly described next. (1) *System specification* - the analyst identifies the system requirements and the environment of the problem, which are obtained after several meetings arranged with the client; (2) *Architectural design* - the system architect determines what kind of agents the system has and how the interaction between them is; (3) *Detailed design* - the agent designer and application designer collaborate to specify the internal structure of each entity that makes up the system overall architecture produced in the previous phase; (4) *Implementation* - the software developer generates and completes the application code; and (5) *Deployment* - the deployment manager deploys the application according to a specified deployment model.

At this point, several issues about this development process are worth noting. The first one is that the phases named system specification and architectural design in *VigilAgent* coincide with the two first phases of the Prometheus methodology [11]. Another detail is that the third phase of *VigilAgent* (detailed design) uses models of INGENIAS [12]. Finally, notice that code is generated and deployed for ICARO-T framework [6]. Several reasons that are introduced in the following paragraph have conducted to this integration.

Prometheus is significant because of the guidelines proposed to identify which the agents are. Another advantage of Prometheus is the explicit use of the concept *scenario* which is closely related to the specific language used in the surveillance domain. Indeed, a surveillance application is developed to deal with a collection of scenarios. Nevertheless, notice that the last phase of Prometheus has not been integrated in *VigilAgent* because it focuses on BDI agents, and entities obtained during the design are transformed into concepts used in a specific agent-oriented programming language named JACK [16]. This supposes, in

principle, a loss of generality. On the contrary, INGENIAS does facilitate a general process to transform models specified during the design phase into executable code. However, INGENIAS does not offer guidelines to identify the entities of the model; the developer's experience is necessary for their identification. Therefore, *VigilAgent* methodology is not developed from scratch but integrates facilities of both Prometheus and INGENIAS to take advantage of both of them. In [8] the an in-depth comparison between Prometheus and INGENIAS methodologies and supporting tools can be found.

Regarding implementation, the ICARO-T framework has been selected as it provides high level software components that facilitate the development of agent-oriented applications. Moreover, it is independent of the agent architecture; that is, the developer can develop new architectures and incorporate them in the framework. This is a clear difference regarding other agent frameworks such as JACK or JADE [2], which provide a middleware instead of an extensible architecture to establish the communications among agents. An additional advantage are the functionalities already implemented in the framework to automatically carry out component management, application initialization and shutdown, reducing in this way the developers' amount of work and guarantying that the components are under control. These last functionalities are usually not provided by other frameworks.

The novelty of the *VigilAgent* methodological approach is to take advantage of both technology along their development process. Specifically, our contribution in this process consists in integrating the technologies selected (Prometheus, INGENIAS and ICARO-T) by using model transformations. On the one hand, a model transformation to translate Prometheus models into INGENIAS models is proposed (see subsection 3.3). Indeed, both methodologies use a different modeling language. On the other hand, another model transformation is proposed to automatically transform an INGENIAS model into ICARO-T code (see subsection 3.4).

3 Case Study: Collaborative Mobile Robots

The selected case study to illustrate the development of robot-based surveillance applications using the *VigilAgent* methodology process implements the collaboration among several mobile robots to carry out a common surveillance task in an industrial estate. The robots navigate randomly through pre-defined surveillance paths in a simulated environment. When there is an alarm in a building, a robot is assigned the role of the chief, three robots are subordinate to the chief, and the other ones are waiting in rearguard to receive orders from the chief (e.g. to replace a damaged subordinate robot). Failures are discovered by the robot itself when any of its mounted devices (e.g., sonar, laser, camera, etc.) does not work in a right way. The robots perceive that an alarm has occurred through two mechanisms: (1) the security guard notifies robots that an alarm has occurred and where it has taken place, (2) the robot is equipped to perceive an alarm itself when it is close enough to the corner of a building; therefore it does not

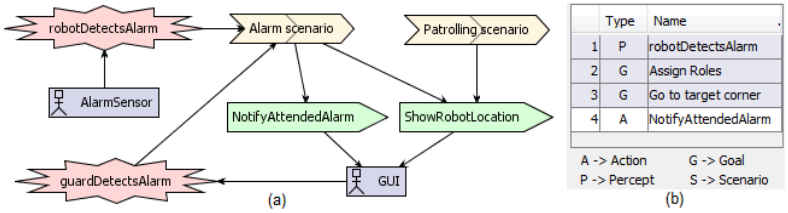

Fig. 1. (a) Analysis Overview Diagram; (b) Steps of the Alarm scenario

have to wait for the security guard announcement. The alarm is covered when a robot coalition (one chief and three subordinates) surrounds the building, that is, the robots that form a coalition are located at the four corners of the building where the alarm has occurred. In this case study three hypotheses are assumed: (1) several alarms do not take place simultaneously, (2) robots do not collide as the streets are wide enough, and, (3) robots navigate from corner to corner.

3.1 System Specification

The analyst starts the system specification phase by developing an *analysis overview diagram*, which shows the interactions between the system and the environment (see Fig. 1a). At this level, firstly, an *AlarmSensor* actor for the device mounted on the robot to detect alarms has been identified. There is also a *GUI* actor representing the user interface that supports the human interaction with the system, that is, it shows the monitoring activity to the security guard, and the commands that he/she can send to the system to announce an alarm, to simulate the failure of a robot, and so on. On the one hand, the information that comes from the environment is identified as *percepts*. For example, the command issued by the guard in order to notify to the robots that an alarm has been triggered (*guardDetectsAlarm*) and the signal captured automatically by the robot device when it is close to a building with an alarm (*robotDetectsAlarm*). On the other hand, every operation performed by the system on the actors is identified as an *action*. For example, an the alert message displayed on the user interface to notify that an alarm has been attended (*NotifyAttendedAlarm*). Finally, relations with the scenarios identified to navigate along the environment when there is no alarm and when there is an alarm are established (see *Patrolling* and *Alarm* scenarios, respectively).

A scenario is a sequence of structured steps - labelled as action (A), percept (P), goal (G), or other scenario (S) - that represents a possible execution pathway of the system. As an example, Fig. 1b illustrates the process performed by the system to attend an alarm. This scenario begins when an alarm has occurred and it is perceived using the sensor mounted on the robot (step 1). Alternatively, the scenario also starts with *guardDetectsalarm* percept. Then, the process to assign roles for each robot to deal with the alarm is started (step 2). Afterwards, robots playing roles chief and subordinates go to the assigned target corners (step 3).

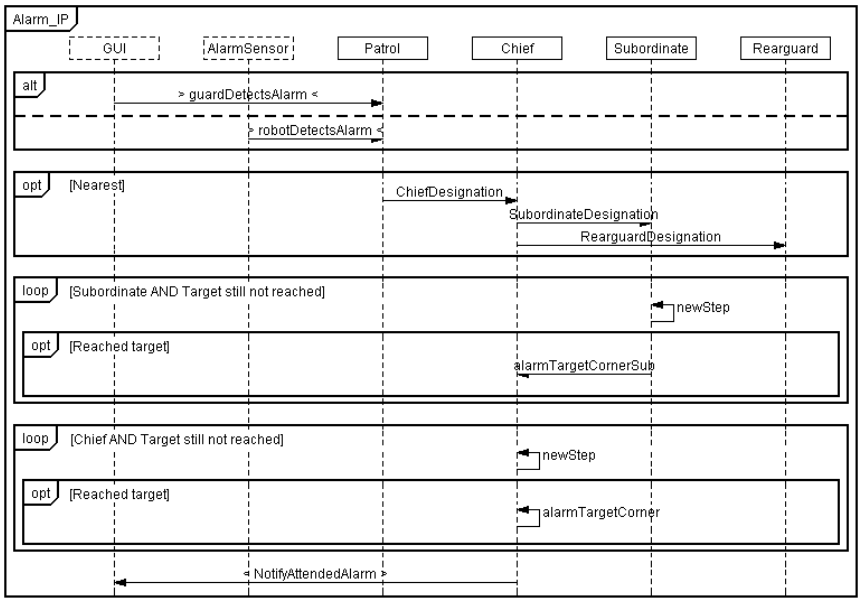

Fig. 2. Alarm interaction protocol

The scenario finishes showing a message when four robots are surrounding the building with alarm (step 4).

Notice that a scenario has a goal representing the objective to be achieved by the scenario. This goal is decomposed into new sub-goals to denote how to achieve the parent goal. For instance, the general goal *Alarm* associated to *Alarm scenario* has been refined into two goals (*Assign Roles* and *Go to target corner*).

3.2 Architectural Design

A relevant task in architectural design phase is to define the conversation entities (interaction protocols, IP) for describing what should happen to realize the specified goals and scenarios. In this case study, IP are used to graphically represent (i) interactions among robots, and, (ii) interactions between a robot and the environment. For example, Fig. 2 details the *Alarm_IP* interaction protocol internal structure. As can be noticed, it involves the roles that four robots and two actors can play to deal with alarm situations (identified by the dotted squares in the diagram). Firstly, the robot is patrolling and captures a percept sent by the *GUI* or *AlarmSernsor* actor when the alarm is triggered. This perception contains the identifier of the building with alarm. Secondly, if a robot is the closest to the building then it sends itself a *ChiefDesignation* message to become the chief robot. Then, on the one hand, the chief sends a *SubordinateDesignation* message to the three next closer robots to convert them into subordinate robots and to notify the target corner where they must go. On the other hand, it sends a *RearguardDesignation* message to the rest of the robots. From this moment

on, each subordinate robot is continuously sending itself a *newStep* message to move towards its target corner until the assigned target has been reached; this is communicated to the chief robot sending an *alarmTargetCornerSub* message. A similar approach is used by the chief to go the chief target corner. Finally, the chief displays a text message on the GUI to communicate that the alarm has been attended (four robots surround the building). Moreover, two protocols are specified to describe the management of failures and patrolling performed for the robots when there is no alarm, respectively.

Finally, another task performed during the architectural design phase is to identify the information managed by the agents or the beliefs describing agent knowledge about the environment or itself. For example, *RobotLocation* data is used to store the robots location, whereas *Environment* data provides information about the simulated environment (industrial estate dimensions, building where the alarm takes place, robots that do not work well, and robots initial locations). Let us point out that this data is public for all agent instances. So, any agent (robot) knows the location of the other ones and the information about the environment.

3.3 Detailed Design

The *VigilAgent* models of the detailed design phase are developed using INGENIAS concepts, which are different from the concepts used in the previous phases. Therefore, transformations from Prometheus to INGENIAS models are carried out to be able to perform the third modelling phase (detailed design). Four conceptual mappings have been developed [8] to transform the structures that involve percepts, actions, messages and data related to agents. These mappings have been inferred considering both the definition of these concepts, and how each Prometheus structure can be modelled using an INGENIAS equivalent structure.

For example, a *percept* is a piece of information from the environment received by means of a sensor. *Percepts* are sent by *actors* (Actor → Percept) and received by agents (Percept → Agent). In INGENIAS, any software or hardware that interacts with the system and that can not be designed as an agent is considered an *application*; and every agent that perceives changes in the environment must be in the *environment model* associated to an application. Therefore, as Fig. 3 shows (arrow 1), the percepts of a Prometheus agent can be triggered in INGENIAS by specifying a collection of *operations* in an *application*. A Prometheus percept has a field, *Information carried*, to specify the information it carries. As Fig. 3 depicts (arrow 2), in INGENIAS this information is described in a type of event named *ApplicationEventSlots* that is associated to the *EPerceives* relation established between the agent and the corresponding application. Notice that the Prometheus agent and actor concepts have been directly mapped to INGENIAS agent and application concepts (see arrows 3 and 4, respectively).

Another interesting example is how Prometheus data are translated into equivalent INGENIAS concepts. It is important to mention that Prometheus data are of any granularity. So, any information represented with a simple data

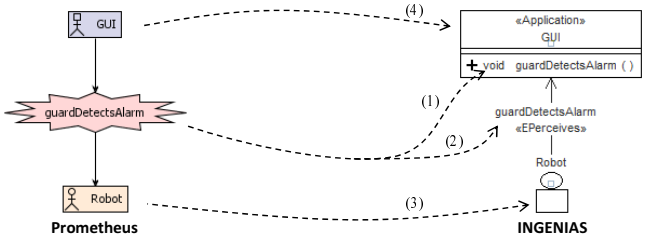

Fig. 3. Mapping information related with Prometheus percepts into INGENIAS

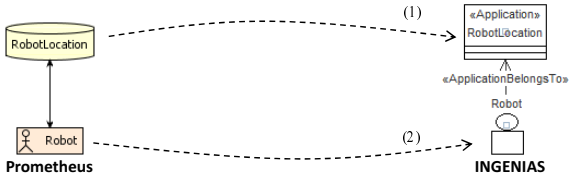

Fig. 4. Mapping information related with Prometheus data into INGENIAS

type (e.g. a string or boolean) or a complex data type (e.g. database) is a data. The model fragment depicted on the left in Fig. 4 represents that the *Robot* agent writes and reads information (*RobotLocation* data) about the robot's location. The relations Agent → Data and Data → Agent express in Prometheus that data is written and read by the agent, respectively. These structures are specified in the INGENIAS environment model with the Agent - *ApplicationBelongsTo* → Application structure, when data has coarse granularity (e.g. a database or a complex data structure to store non-persistent information). Notice that the application is not the data, but the entity that provides methods to data management. Regarding simple data, they are not translated to an INGENIAS concept but they are declared when the implementation is carried out. Finally, let us highlight that action and message concepts in Prometheus are equivalents in INGENIAS to operation defined in the applications, and interaction unit concepts, respectively.

Once the transformation has been performed, new activities should be carried out to complete the modelling. Firstly, it is necessary to identify the task performed by agents for every received perception or message. After that, the behaviour of each agent is specified using information about tasks, received percepts and messages. The behaviour is modelled with a finite state automaton, where the states represent concrete situations of the agent life cycle. For example, the state diagram interpretation corresponding to the reactive agent that controls a robot is as follows (see Fig. 5) - the term event is used to refer either to a perception or a message.

– There are three kinds of states: initial (*InitialState*), final (*FinalState*) and intermediate (e.g., *AlarmDetection*, *Rearguard*, and so on).

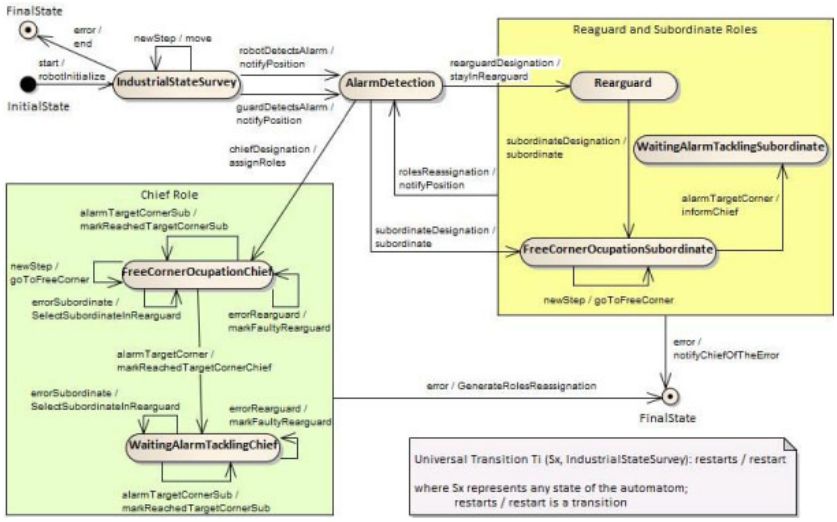

Fig. 5. State diagram for the robot's behaviour control

- The agent starts executing when it receives a start event, causing the agent to change to *IndustrialStateSurvey* state and to execute the *robotInitialize* action. The agent does not consult if there are events associated with transitions that go from *IndustrialStateSurvey* state to *robotInitialize*. Once the agent has been initialized, it navigates randomly (cycling in *IndustrialStateSurvey* state) until an alarm appears (a *robotDetectsAlarm* or *guardDetectsAlarm* event arrives). In this case, *notifyPosition* action is executed to determine who the chief is. After that, *assignRoles* action produces the assignment of subordinate and rearguard roles to the other agents. The agent with chief role enters the states enclosed in the 'Chief Role' boundary, whereas the agents with other roles go to states enclosed into 'Rearguard and Subordinate Roles' boundary.
- There is a kind of particular transition, the universal transition, which is valid for any automaton state. This transition takes place for a given input; the action is executed and the automaton transits to the next state, regardless of the automaton's state.
- In *robotInitialize* action the agent sends itself the *newStep* event to start moving the robot. *End* action marks the agent as damaged. *Restart* action restarts the simulation process, and, afterwards the agent sends itself a *newStep* event. The rest of actions are described in Table 1, where the role played by the agent is mentioned only when it is meaningful.

In Fig. 5 the following notation is adopted for the purpose of clarity. A transition that goes from a boundary to a state means that such a transition is able to go from any state enclosed into the boundary to the specified target state (see, for instance, *rolesReassignation / notifyPosition*).

Table 1. Description of the automaton's actions

Action	Description
move	The agent sends itself a *newStep* event if there is no alarm.
notifyPosition	Determine if the agent becomes the chief. The chief is the agent closer to the alarm and the ties are solved in favour of the agent that has a lower index. The agent sends itself a *ChiefDesignation* event if it becomes the chief.
stayInRearguard	The agent updates its role as rearguard and learns who the chief is. The robot does not move while it plays this role.
subordinate	The agent (1) updates its role as subordinate and learns who the chief is, and, (2) sends itself a *newStep* event.
assignRoles	The agent (1) updates its role as chief, (2) assigns to what corner the chief should go, (3) assigns to what corners the three next closest agents to the alarm should go to, and sends them a *subordinateDesignation* event that contains the following information: the target corner that it should occupy and who the chief robot is, (4) send to the other agents a *rearguardDesignation* event, and finally, (5) sends itself a *newStep* event. The *notifyPosition* action may be consulted to know how the ties are solved.
goToFreeCorner	If the chief/subordinate agent is on the assigned target corner, then it sends itself an *alarmTargetCorner* event; otherwise it determines to what corner it moves next, and it sends itself a *newStep* event.
notifyChiefOf-TheError	If the agent is a rearguard, then it sends an *errorRearguard* event to the chief; if it is a subordinate agent, then it sends an *errorSubordinate* event, which contains its identification number to the chief. In both cases, the agent marks the controlled robot as damaged.
informChief	The subordinate agent sends to the chief agent an *alarmTargetCornerSub* event, which contains the subordinate agent identification number.
markReached-TargetCornerSub	The chief agent (1) marks the target corner that the subordinate agent has occupied, (2) increases the number of occupied corners, and, (3) notifies the user when four target corners have been occupied.
markReached-TargetCornerChief	The chief agent (1) marks the target corner that it has occupied, (2) increases the number of occupied corners, and, (3) notifies the user when four target corners have been occupied.
selectSubordinate-InRearguard	The chief agent (1) increases the number of damaged robots, (2) identifies the closest rearguard agent to the target corner to be occupied by the damaged subordinate agent, and, (3) sends a *subordinateDesignation* event that contains the target corner and the chief identification number.
markFaulty-Rearguard	The chief agent (1) marks the rearguard agent that sent an *errorRearguard* event as damaged, and, (2) increases the number of damaged robots.
generateRoles-Reassigantion	The chief agent (1) marks itself as damaged, (2) increases the number of damaged robots, and, (3) sends to the rest of agents a *rolesReassignation* event that contains the location of the building where the alarm occurred.

3.4 Implementation and Deployment

Now, we are in front of the second major contribution of the *VigilAgent* methodology. Our approach considers that the tool supporting INGENIAS, Ingenias Development Kit (IDK) [12], is an exceptional agent tool to develop a Model-To-Text transformation for generating code for any target language chosen. This is ICARO-T in *VigilAgent*, as it provides the necessary functionalities for developing new modules capable to carry out this task. These modules are developed following a general process based on both the definition of specific templates for each target platform, and procedures to extract information from INGENIAS models.

To facilitate the implementation and deployment, we have developed modules to automatically generate ICARO-T code of the system being developed from the models specified with IDK. The process for using our modules is carried out as follows. (1) The *INGENIAS ICARO-T Framework generator module* (*IIF*) is used to automatically generate code for the detailed design specification. *IIF* generates several XML files that describe the behaviour of each agent, java classes for each agent and application, and the XML file describing the application deployment. (2) The developer manually inserts code in the protected regions of the generated java classes and implements those new classes he/she needs. (3) The developer uses the *ICAROTCodeUploader* module to update the model with the modifications introduced in the protected regions. Finally, the script file generated by *IIF* module is executed by the deployment manager to launch the developed application.

4 Conclusions

The development of an application using *VigilAgent* has been described in this paper. The learning curve of *VigilAgent* can be steep at first because users must get used to different terms that have the same meaning depending on the technology used in each phase (Prometheus and INGENIAS for modelling, and ICARO-T for implementation). However, this disadvantage is overcome thanks to the two transformations that are automatically executed.

It is worth pointing out that the time spent learning how to develop and implement the *INGENIAS ICARO-T Framework generator* and the *ICAROT-CodeUploader* modules described in section 3.4 was two months and fifteen days. This effort is rewarded as new applications are modelled and implemented. Our future work consists in applying *VigilAgent* methodology to new case studies and domains.

Acknowledgements

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

References

1. Anisi, D.A., Thunberg, J.: Survey of patrolling algorithms for surveillance UGVs. Scientific report (2007), www.math.kth.se/~anisi/articles/CTAPP_survey.pdf
2. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. John Wiley and Sons, Chichester (2007)
3. Cossentino, M., Gaglio, S., Garro, A., Seidita, V.: Method fragments for agent design methodologies: From standardisation to research. International Journal of Agent-Oriented Software Engineering 1(1), 91–121 (2007)
4. Fernández-Caballero, A., Castillo, J.C., Martínez-Cantos, J., Martínez-Tomas, R.: Optical flow or image subtraction in human detection from infrared camera on mobile robot. Robotics and Autonomous Systems 58(12), 1273–1280 (2010)
5. Fernández-Caballero, A., Gascueña, J.M.: Developing multi-agent systems through integrating Prometheus, INGENIAS and ICARO-T. Communications in Computer and Information Science (Agents and Artificial Intelligence) 67, 219–232 (2010)
6. Garijo, F.J., Polo, F., Spina, D., Rodríguez, C.: ICARO-T User Manual. Technical Report, Telefonica I+D (2008)
7. Gascueña, J.M., Fernández-Caballero, A.: Agent-based modeling of a mobile robot to detect and follow humans. In: Håkansson, A., Nguyen, N.T., Hartung, R.L., Howlett, R.J., Jain, L.C. (eds.) KES-AMSTA 2009. LNCS, vol. 5559, pp. 80–89. Springer, Heidelberg (2009)
8. Gascueña, J.M., Fernández-Caballero, A.: Prometheus and INGENIAS agent methodologies: A complementary approach. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 131–144. Springer, Heidelberg (2009)
9. Gascueña, J.M., Fernández-Caballero, A.: Agent-oriented modeling and development of a person-following mobile robot. Expert Systems with Applications 38(4), 4280–4290 (2011)
10. Gascueña, J.M., Fernández-Caballero, A.: On the use of agent technology in intelligent, multi-sensory and distributed surveillance. The Knowledge Engineering Review (2011) (in press)
11. Padgham, L., Winikoff, M.: Developing Intelligent Agents Systems: A Practical Guide. John Wiley and Sons, Chichester (2004)
12. Pavón, J., Gómez-Sanz, J., Fuentes, R.: The INGENIAS Methodology and Tools. In: Agent-Oriented Methodologies, pp. 236–276. Idea Group Publishing, USA (2005)
13. Rogge, J.A., Aeyels, D.: A strategy for exploration with a multi-robot system. In: Informatics in Control, Automation and Robotics. Lecture Notes in Electrical Engineering, vol. 24, part II, pp. 195–206 (2009)
14. Sokolova, M.V., Fernández-Caballero, A.: Facilitating MAS Complete Life Cycle through the Protégé-Prometheus Approach. In: Nguyen, N.T., Jo, G.-S., Howlett, R.J., Jain, L.C. (eds.) KES-AMSTA 2008. LNCS (LNAI), vol. 4953, pp. 63–72. Springer, Heidelberg (2008)
15. Weiss, G.: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press, Cambridge (1999)
16. Winikoff, M.: Jack Intelligent Agents: An Industrial Strength Platform. Multi-Agent Programming Languages, Platforms Applications, pp. 175-193 (2005)
17. Wooldridge, M.: Agent-based software engineering. IEEE Proceedings - Software Engineering 144(1), 26–37 (1997)