Agent-Based Development of Multisensory Monitoring Systems

José Manuel Gascueña[1], Antonio Fernández-Caballero[1,2], Elena Navarro[1,2], Juan Serrano-Cuerda[1], and Francisco Alfonso Cano[1]

[1] Instituto de Investigación en Informática de Albacete (I3A), n&aIS Group, Campus Universitario s/n, 02071-Albacete, Spain
{JManuel.Gascuena,Juan.SerranoCuerda,Francisco.Cano}@uclm.es
[2] Departamento de Sistemas Informáticos, Universidad de Castilla-La Mancha, Campus Universitario s/n, 02071-Albacete, Spain
{Antonio.Fdez,Elena.Navarro}@uclm.es

Abstract. This paper introduces the use of the *VigilAgent* agent methodology to develop monitoring systems. This work is based on the suitability of the specific characteristics of agency for developing monitoring systems. It is usual to develop them following an ad-hoc approach instead of using a methodology for achieving quality standards expected from commercial software. In this paper, the five phases of *VigilAgent*, namely System specification, Architectural Design, Detailed design, Implementation and Deployment, are introduced. The proposal is validated through the case study of controlling the access of human beings to a specific area.

1 Introduction

Security systems [9] are being installed in environments such as bank, parking, motorway and underground to protect humans from attacks or burglaries. The development of monitoring systems is very complex as they work in highly dynamic and heterogeneous environments. They deploy, in the observed scenarios, several kinds of sensors that perform actions with a certain degree of autonomy to collect information about their surrounding area, and to cooperate in the recognition of special situations in a semi-automatic way. The characteristics of autonomy and cooperation are often cited as the rationale of why multi-agent systems (MAS) are especially appropriate for monitoring tasks [8], [6], [10]. In fact, agent technology has already been used in several monitoring systems [4]. However, to the best of our knowledge, they are usually developed following an ad-hoc approach without using a methodology that guides stakeholders in achieving quality standards expected from commercial software. So, this paper proposes the introduction of an Agent-Oriented Software Engineering (AOSE) methodology, named *VigilAgent*, to carry out well documented monitoring applications throughout the different phases that form the development process.

In this paper, the *VigilAgent* methodology is theoretically introduced and also applied to face a particular monitoring task supported in these systems. In

J.M. Ferrández et al. (Eds.): IWINAC 2011, Part I, LNCS 6686, pp. 451–460, 2011.

this case *VigilAgent* is used to control the access to a specific area. Assigning human personal to carry out this task is a common solution. However, this is an expensive solution where human's attention decreases after some elapsed time working or watching video. Thus, it sounds interesting to perform this task automatically via software which is responsible for collecting information to inform when some anomalous situation occurs. In addition, it is relevant to record normal situations to guarantee some usual activities; for instance, that the current occupation capacity does not exceed above a threshold. So, a company takes knowledge about economical losses arising for cunning users, where it is necessary to have more personal to prevent risky situations. The cost of the personnel is decreased because it is not necessary to have a guard in every access control point. Notice that monitoring software does not replace the security personnel but is complementary to offer a more efficient solution.

The rest of the paper is organized as follows. In section 2, an overview of *VigilAgent* methodology phases is offered, justifying why Prometheus [5], INGENIAS [7] and ICARO-T [3] technologies are integrated. Then, section 3 introduces the case study used to demonstrate the applicability of *VigilAgent*. Finally, section 4 offers some conclusions.

2 Description 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, it is worthy of notice several issues about this development process. The first one is that phases named system specification and architectural design in *VigilAgent* are the two first phases of Prometheus methodology [5]. Another detail is that the third phase of *VigilAgent* (detailed design) uses models of INGENIAS [7]. Finally, notice that code is generated and deployed for ICARO-T framework [3]. Several reasons that are introduced in the following have conducted to this integration.

Prometheus is significant because of the guidelines it offers to identify which the agents and their interactions are. Another advantage of Prometheus is the explicit use of the concept *scenario* which is closely related to the specific language used in the monitoring domain. Indeed, a monitoring application is developed to deal with a collection of scenarios. Nevertheless, notice that Prometheus last phase has not been integrated in *VigilAgent* because it focuses on belief-desire-intention (BDI) agents and how the entities obtained during the design

are transformed into concepts used in a specific agent-oriented programming language named JACK [1]; 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.

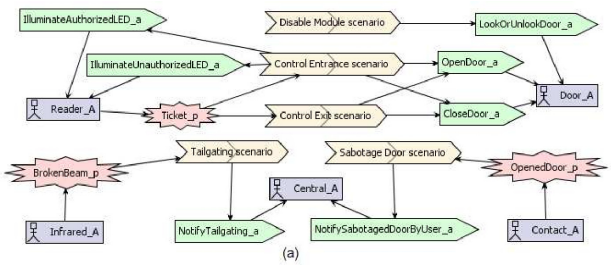
Regarding implementation, the ICARO-T framework has been selected because it provides high level software components that facilitate the development of agent applications. Moreover, it is independent of the agent architecture; that is, the developer can develop new architectures and incorporate them in the framework. This is a clear difference regarding other agent frameworks such as JACK or JADE [1], 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 the developers' amount of work and guarantying that the components are under control. These last functionalities are usually not provided by other frameworks.

3 A Case Study: Access Control

Access control is the usual and basic term used for monitoring and controlling entrances to and exits from a specific area. This section illustrates how to use *VigilAgent* in order to develop an intelligent system that automatically controls entrances/exits of humans to/from an enclosure throughout the installed modules. Specifically, each module facilitates the entrances and exits according to its configuration and is composed of the following components: a reader device, an automatic door, a contact sensor and an infrared sensor.

In order to go in/out of the enclosure throughout a module, first, the user inserts a ticket into the reader device that the system verifies against the users' database. Then, a LED illuminates in green if the user is authorized, otherwise it illuminates in red. Moreover, if the user is authorized then the door is opened, and closed once the user has crossed or some time has elapsed.

In addition, the system collects and shows the guard statistics about the number of humans crossing each door and the number of humans located inside the enclosure by using the infrared sensor located in each module. It should also control if any anomalous situation happens, such as tailgating or if a door is blocked by a human when the system opens it. A *tailgating* situation is detected when some cunning human crosses a door that has been opened by a user correctly authenticated. The system also shows the state of the devices and offers the guard the possibility of disabling a module if its door remains closed despite having correctly authenticated a user.



	Type	Name	Role
1	P	ClosedDoor_p	Control contact_R
2	P	Ticket_p	User identification_R
3	G	Check identification	User identification_R
4	A	IlluminateAuthorizedLED_a	User identification_R
5	A	IlluminateAuthorizedLEDCentral_a	User identification_R
6	G	Manage opening process	Facilitate acces_R
7	G	Control door state	Control contact_R
8	A	DelayClosingDoor_a	Facilitate acces_R
9	P	BrokenBeam_p	Control infrared_R
10	G	Update entrance traffic	Control infrared_R
11	G	Manage closing process	Facilitate acces_R
12	G	Control door state	Control contact_R

(b)

Fig. 1. (a) Analysis overview diagram; (b) Steps of Control Entrance scenario

3.1 System Specification Phase

Usually, the system specification phase begins with the *analysis overview diagram*, which shows the interactions between the system and the environment (see Fig. 1a). At this level, firstly, an *actor* for each device of a module (reader, door, infrared and contact sensors) has been identified; there is also a *Central_A* actor representing the user interface that supports the human interaction with the system, that is, it shows the monitored activities to the security guard, and the commands he/she can send to the system to disable modules. Moreover, on the one hand, the information that comes from the environment is identified as *percepts*. For example, the code associated to the card introduced by the user into the reader (*Ticket_p*) and the signal captured by the infrared device when its beam is broken (*BrokenBeam_p*). On the other hand, every operation performed by the system on the actors is identified as an *action*. For example, the commands issued to open and close a door (*OpenDoor_a* and *CloseDoor_a*) and alert messages displayed on the user interface to notify that an anomalous situation happens (*NotifySabotagedDoorByUser_a* and *NotifyTailgating_a*). Finally, relations with the scenarios identified to control the entrance, the exit and anomalous situations are established (see *Tailgating* and *Sabotage Door* scenarios).

A scenario is a sequence of structured steps - labeled as action (A), percept (P), goal (G), or other scenario (S) - that represents a possible execution way of the system. As an example, Fig. 1b illustrates the process performed by the system to control the crossing of authorized users through a module. This scenario begins when it is perceived that the door is closed (step 1) and the user introduces his/her identification ticket into the reader (step 2). Then, the user database is queried to check if this ticket belongs to an authorized user (step 3). Based on this information, the LEDs are illuminated to notify which result is obtained with this checking (steps 4 and 5). After, the door is opened (step 6), the door state is supervised (step 7), a delay is introduced allowing the user to cross the module before the door is closed again (step 8), it is perceived when the infrared sensor is interrupted (step 9), the counter of people that cross the module is updated (step 10) and the process to close is started (step 11). The scenario finishes monitoring the door state (step 12).

Table 1. Roles description

Role	Description
User identification_R	It manages the user identification process to gain access to or come out of enclosure.
Facilitate access_R	It manages the door actuator to provide the users authenticated correctly with access.
Control infrared_R	It aims to update statistics when the infrared sensor detects a person located in front of it.
Control contact_R	Its objective is to monitor the door state thanks to the information perceived by the contact sensor installed in the door.
Management tailgating_R	It is responsible for detecting and notifying when the anomalous situation named tailgating happens.
Management sabotage door_R	It is responsible for detecting and notifying when someone, different from the security guard, blocks the door.
Intercommunication_R	It is responsible for managing the communication between software entities for person access control.

For every scenario a goal is identified that represents the goal to be achieved by the scenario. In the proposed multi-agent system approach, several agents communicate and coordinate to pursue the common general goal *Control entrance and exit*. In the *goal overview diagram*, this general goal has been refined into five goals (*Control Entrance*, *Control Exit*, *Tailgating*, *Sabotage Door*, and *Disable Module*) related to the scenarios identified. Similarly, some of these goals have also been decomposed into several sub-goals to denote how to achieve each parent goal. Roles are identified by clustering goals, and linking perceptions and actions (see details in Table 1).

3.2 Architectural Design Phase

Usually, applications need also reading and/or writing data to achieve the functionality related with some roles. This information is specified during the architectural design phase in the *data coupling diagram* (see Fig. 2a). A data is information managed by agents or beliefs representing agent knowledge about the environment or itself. For example, a data (*BDD*) should be available to validate every ticket introduced. Furthermore, there are data to count how many people cross each module (*Traffic*), to have control on the enclosure capacity level (*OccupationLevel*) and determine if a user goes in or goes out the enclosure (*ModuleConfiguration*). Therefore, notice that data are able to have any granularity.

Once the functionality of the roles has been described through their relations with other entities (percepts, actions, goals and data), the guidelines offered by Prometheus to decide the types of agents that are included in the system are applied during the second phase namely architectural design. In general terms, these guidelines consist of grouping roles to obtain the system. The cohesion and coupling criteria are used to decide which the best groupings are to ease their maintenance. These two concepts are essential to obtain a suitable software development that is distinguished by its maximum cohesion and minimum coupling. In the proposed case study, to achieve the system requirements, an agent is introduced for each role identified (see Fig. 2b).

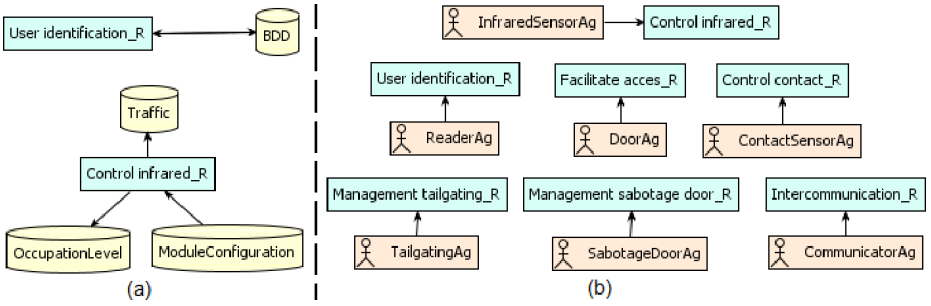


Fig. 2. (a) Data coupling diagram; (b) Agent-Role grouping diagram

Another relevant task performed in architectural design phase is to define the agent conversations (Interaction Protocols, IP) in order to describe what should happen to realize the specified goals and scenarios. Interaction Protocols are a graphical representation that shows (i) interactions among agents, and (ii) interactions among agents and the environment. It should be highlighted that percepts are originated by actors that communicate with agents, whereas actions describe a communication of agents with actors. For example, Fig. 3a details the *Tailgating_IP* interaction protocol internal structure, a sub-protocol of *Access_IP*. As can be noticed, it involves two agents and two actors (identified by the dotted squares in the diagram) to detect tailgating situations. This situation happens when the door remains open after an authorized person has passed through it and the related infrared sensor has been activated once. From this moment, whenever the *InfraredSensorAg* agent perceives that the infrared sensor beam has been broken again, a new crossing through the module is counted (*IncreaseTrafficCounter_a*), the number of people inside the enclosure is updated (*IncreaseOccupationCounter_a*), and the detection is communicated to the *TailgatingAg* agent sending a *BrokenBeam* message. Finally, the *TailgatingAg* agent executes the *NotifyTailgating_a* action in order to notify the *Central_A* actor an unauthorized access. This checking is carried out until the *ContactSensorAg* agent notifies the *TailgatingAg* agent that the door is closed thanks to the *Access_IP* protocol. Tailgating detection is carried out in the same way for entrance and exit alternatives.

Fig. 3b shows a fragment of *system overview diagram* for the proposed system design that includes the Tailgating Interaction Protocol. It can be observed that once roles have been grouped into agents, the information about percepts, actions, and data related to roles is automatically propagated and linked with the agents in the system overview diagram. Finally, the *agent acquaintance diagram* that contains the communication links between agents is automatically generated using the messages of information included in the interaction protocols. In short, every agent that perceives information captured by a device uses an intermediate agent (*CommunicatorAg*) to establish its communication. On the other hand, the agents responsible for detecting anomalous situations (*TailgatingAg* and *SabotageDoorAg*) can benefit from the knowledge offered by agents related

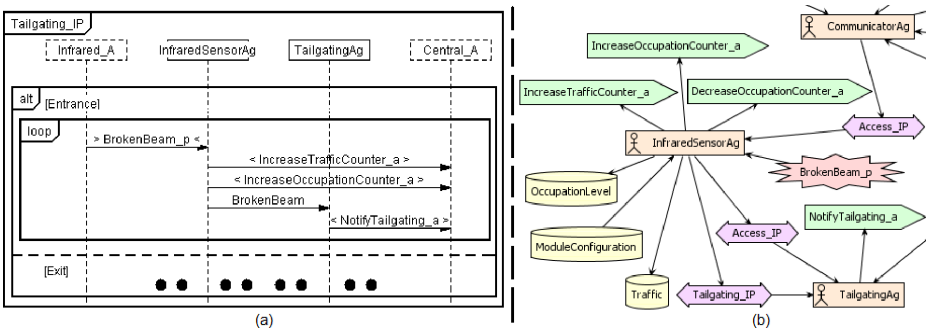


Fig. 3. (a) Tailgating interaction protocol; (b) Fragment of system overview diagram

to the contact and infrared sensors (*InfrarredSensorAg* and *ContactSensorAg*) to carry out their tasks.

3.3 Detailed Design Phase

Now, we are in front of one of the major aspects and contributions of this proposal. The system overview diagram previously obtained (see Fig. 2b) describes the overall system architecture, that is, the agents identified, the perceptions received, the actions to be executed, the interaction protocols among agents, and the data read and/or written by agents. These entities are described using Prometheus concepts. However, the *VigilAgent* models of the detailed design phase are developed using INGENIAS concepts, which are different from the previous ones. Therefore, a transformation from Prometheus models to INGENIAS models must be carried out in order to perform the next modeling phase.

We have developed four conceptual mappings [2] 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 modeled 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) by means of *interaction protocols*, these percepts are received by agents (Percept → Agent). The relations among actors, percepts and agents (Actor → Percept → Agent) are described in the *system overview diagram*. In INGENIAS, all the software and hardware that interacts with the system and cannot 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. 4 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. 4 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

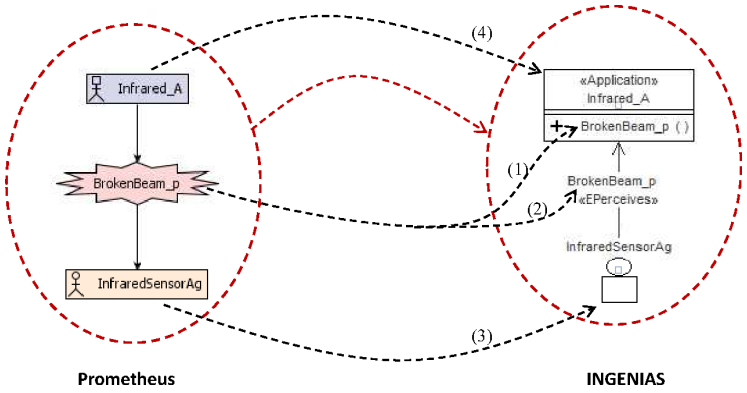


Fig. 4. Example of mapping information related with Prometheus percepts into INGENIAS

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). Finally, let us highlight that action, message and data concepts in Prometheus are equivalents to operation defined in the applications, interaction unit and application concepts, respectively, in INGENIAS. The conceptual mappings have been automated specifying a Model-To-Model (M2M) transformation using the Query/View/Transformation (QVT) language.

Once the transformation has been performed, three activities should be carried out to complete the modeling. Firstly, it is necessary to identify the tasks performed by each agent by examining the *interaction model* and the initial *environment model* which are automatically generated from the Prometheus model applying the transformation. Specifically, an agent should perform a task for every received perception or message. After that, a *state diagram* describing the behavior of each agent is specified using information about tasks, received percepts and messages. Secondly, the *Applications*, obtained after once the transformation is executed, only include methods to send perceptions to the agents and to specify actions to be executed by the agents. Therefore, these applications should be refined to include new methods depending on the specific needs of the system being developed. For example, in the proposed case study, the *CheckUser* method was added to the *BDD* application to get information stored in a database, and the *GetId* method was added in *Reader_A* application to get its identifier. Finally, a model to describe the *application deployment* is specified attaching numbers in notes to represent the number of instances that there are in execution for each type of entity.

3.4 Implementation and Deployment Phases

Our approach considers that supporting tool for INGENIAS, Ingenias Development Kit (IDK) [7], is an exceptional agent tool to develop a Model-To-Text (M2T) 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. In the literature there is no proposal for mapping INGENIAS models to ICARO-T code. So, our contribution to solve this gap has been to identify them and to develop modules to automatically carry out the Model-to-Text transformation from INGENIAS to ICARO-T.

The process for using the IDK modules in order to automatically generate ICARO-T code of the system being developed that can be updated with new code introduced by the user when necessary 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 behavior 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 in order to update the model with the modifications introduced in the protected regions. It allows the developers to always keep the model and the source code synchronized, so that those changes introduced in the source code are kept when code is regenerated again from the model. Finally, the script file generated by IIF module is executed by the deployment manager to launch the developed application.

4 Conclusions

The lack of consensus about which the best methodology to develop agent-oriented applications is has driven the main idea presented in this paper: not to propose a new methodology from scratch but to combine several methodologies, *VigilAgent*, to take advantage of each one of the analyzed methodologies. 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 modeling, and ICARO-T for implementation). However, this disadvantage is overcome thanks to the two transformations that are executed automatically. First, a Model-to-Model transformation executed by means of Medini automatically transforms Prometheus structures into their equivalent INGENIAS structures. Second, the Model-to-Text transformation tool transforms INGENIAS models into code for the ICARO-T framework.

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 was two months and fifteen days. This effort is rewarded as new applications are modeled and implemented because (i) the quality of the developed system is improved as the automatically generated code does “not” have bugs; (ii) productivity is improved as the time necessary for coding is reduced because the developer does not need to learn the structure, location and

naming rules for ICARO-T applications files; and (iii) more efforts can be devoted to solve errors during early phases of the life cycle, avoiding in this way the "snow ball" effect. Our future work consists in going on to apply *VigilAgent* to face new case studies.

Acknowledgements

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

References

1. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A.: Multi-Agent Programming: Languages, Platforms and Applications. Springer, Heidelberg (2005)
2. 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)
3. Gascuñea, J.M., Fernández-Caballero, A., Garijo, F.J.: Using ICARO-T framework for reactive agent-based mobile robots. In: Advances in Practical Applications of Agents and Multiagent Systems. Advances in Soft Computing, vol. 70, pp. 91–101. Springer, Heidelberg (2010)
4. 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)
5. Padgham, L., Winikoff, M.: Developing intelligent agents systems: A practical guide. John Wiley and Sons, Chichester (2004)
6. Patricio, M.A., Castanedo, F., Berlanga, A., Pérez, O., García, J., Molina, J.M.: Computational intelligence in visual sensor networks: Improving video processing systems. SCI, vol. 96, pp. 351–377. Springer, Heidelberg (2008)
7. Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: The INGENIAS methodology and tools. In: Agent-Oriented Methodologies, pp. 236–276. Idea Group Publishing, USA (2005)
8. Pavón, J., Gómez-Sanz, J.J., Fernández-Caballero, A., Valencia-Jiménez, J.J.: Development of intelligent multi-sensor surveillance systems with agents. Robotics and Autonomous Systems 55(12), 892–903 (2007)
9. Räty, T.D.: Survey on contemporary remote surveillance systems for public safety. IEEE Transactions on systems, man, and cybernetics - Part C: Applications and reviews 40(5), 493–515 (2010)
10. Rivas-Casado, A., Martínez-Tomás, R., Fernández-Caballero, A.: Multiagent system for knowledge-based event recognition and composition. Expert Systems: The Journal of Knowledge Engineering (2011) (in press)