Expert Systems

Article

DOI: 10.1111/exsy.12023

Model-to-model and model-to-text: looking for the automation of VigilAgent

José Manuel Gascueña,[1] Elena Navarro,[2]
Antonio Fernández-Caballero[2] and Rafael Martínez-Tomás[3]

(1) Symbia IT S.L., Parque Científico y Tecnológico de Albacete, Albacete, Spain
E-mail: JManuel.Gascuena@symbiait.com
(2) Departmento de Sistemas Informáticos, Universidad de Castilla–La Mancha, Albacete, Spain
E-mail: Elena.Navarro@uclm.es; Antonio.Fdez@uclm.es
(3) Departamento de Inteligencia Artificial, Universidad Nacional de Educación a Distancia, Madrid, Spain
E-mail: rmtomas @dia.uned.es

Abstract: *VigilAgent is a methodology for the development of agent-oriented monitoring applications that uses agents as the key abstraction elements of the involved models. It has not been developed from scratch, but it reuses fragments from Prometheus and INGENIAS methodologies for modelling tasks and the ICARO framework for implementation purposes. As VigilAgent intends to automate as much as possible the development process, it exploits.*
Model transformation techniques are one of the key aspects of the model-driven development approach. A model-to-model transformation is used to facilitate the interoperability between Prometheus and INGENIAS methodologies. Also, a model-to-text transformation is performed to generate ICARO code from the INGENIAS model. A case study based on access control is used to illustrate the fundamentals of the model-to-model and model-to-text transformations implemented in VigilAgent.

Keywords: agent-oriented software methodologies, model-driven development, model-to-model transformations, model-to-text transformations

1. Introduction

Model-driven development (MDD) (Beydeda *et al*., 2005) is having more and more attention for developing software. It revolves around raising the abstraction level at which the developer works to exploit the models as the cornerstone of the software development process. This has the following consequences for the development process: (1) more time is devoted to analysing and designing models; (2) the time necessary to perform coding tasks is reduced, as code generators are developed for the selected target platform, being programmers responsible for completing those parts of the systems that are not generated automatically; (3) the quality of the developed system is improved as the generated code (usually) does not have bugs; (4) productivity is improved as the time necessary for coding is reduced, so that more efforts are devoted to solving errors during early phases of the life cycle, avoiding in this way the *snow ball* effect; and (5) portability is improved as adopting a new technology just requires developing a new code generator. Indeed, the models are independent of any software implementation technology. Furthermore, MDD also provides interoperability among heterogeneous systems thanks to the specification of bridges among different technologies. In short, using a MDD approach for developing software applications offers important benefits in fundamental aspects such as productivity, portability, interoperability and maintenance (Kleppe et al., 2003; Schmidt, 2006).

In the MDD approach, the model transformation techniques (Czarnecki and Helsen, 2006) play a very relevant role. On the one hand, model-to-model (M2M) transformations turn a source model into a target model located in the same or different abstraction level. On the other hand, model-to-text (M2T) transformations are another key for MDD as they automate the last step of the process by generating the final source code of the system. Therefore, the difference between M2M and M2T transformations is the outcome obtained once they are run, as the former generates a model and the latter just generates a document in textual format, usually of string type.

Nowadays, security systems (Haering *et al*., 2008; Kumar *et al*., 2008; Räty, 2010) are being installed in environments such as bank, parking, highway and underground to protect humans from attacks or burglaries. The development of monitoring systems is a very complex task as they work in highly dynamic and heterogeneous environments. Monitoring systems deploy several kinds of sensors that perform actions with a certain degree of autonomy to collect information about their surrounding area in the observed scenarios 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 (Pavón *et al*., 2007; Patricio *et al*., 2008; Rivas-Casado *et al*., 2011). In fact, agent technology has been used in several monitoring systems (Gascueña and Fernández-Caballero, 2011) so far. However, to the best of our knowledge, they are usually developed following an ad hoc approach, that is, without a methodology that guides a stakeholder in achieving the quality standards expected from commercial software. So, this paper proposes the exploitation of an agent-oriented software engineering methodology, named *VigilAgent*, to carry out well-documented monitoring applications throughout the different phases that make up the development process. Moreover, VigilAgent uses agency as the abstraction key element of the models specified to develop monitoring applications.

VigilAgent methodology has not been developed from scratch, but it reuses fragments from the Prometheus (Padgham and Winikoff, 2004) and INGENIAS (Pavón *et al*., 2005) methodologies for modelling tasks and the ICARO framework (Garijo *et al*., 2008) for implementation purposes. VigilAgent implements an M2M transformation to facilitate a seamless transition from Prometheus models to INGENIAS models as they do not use a common modelling language to specify MAS applications. Moreover, VigilAgent applies an M2T transformation to generate ICARO code from INGENIAS models. Thus, VigilAgent takes advantage of one of the aspects of the MDD approach, namely model transformation. This paper focuses on describing the fundamentals of the M2M and M2T transformations implemented in VigilAgent.

The rest of the paper is organized as follows. First, Section 2 presents a brief discussion about some of the most relevant works developed in this area. In Section 3, an overview of the phases that the VigilAgent methodology entails, explaining why Prometheus, INGENIAS and ICARO have been integrated, is offered. Then, Section 4 introduces a case study used to illustrate the M2M and M2T transformations implemented in VigilAgent. Section 5 describes the VigilAgent concepts used by the implemented M2M transformation by using examples extracted from the case study. Next, the implemented M2M and M2T transformations are described in Sections 6 and 7, respectively. Finally, Section 8 offers the main conclusions and hints for future work.

2. Related work

Several works have been already developed in the area of MAS that exploit model transformations to a greater or lesser scope. For instance, INGENIAS has been extended to support the application of M2M transformations (García-Magariño *et al*., 2011). Another related work is the ASEME methodology (Spanoudakis, 2009), which employs the two types of transformations previously described (M2M and M2T) throughout its different phases. Moreover, it also uses a text-to-model transformation for reverse engineering purposes. Ayala *et al.* (2011) proposed M2M transformations to translate PIM4Agents (Hahn *et al*., 2009), a generic agent meta-model used at design phase, into Malaca, a platform neutral meta-model for agents proposed by Amor and Fuentes (2009).

Regarding M2T transformations, several works related to the MAS area are found in the literature. For instance, the INGENIAS methodology provides the possibility to specify and run M2T transformations using a mechanism devised by INGENIAS developers (Pavón *et al*., 2006). Similarly, a tool developed by Kardas *et al.* (2009) supports the execution of M2T transformations specified with MOFScript (OMG, 2011) to automatically generate code for two different agent platforms. The main difference between them is that the former uses models from a higher abstraction level than the latter. Taom4E (Morandini *et al*., 2011), a tool for the development of software following the TROPOS methodology (Morandini *et al*., 2008), includes functionalities to generate code for Jadex language (Braubach *et al*., 2005). Domain Specific Modeling for Multiagent Systems (Dsml4mas) Development Environment (DDE) (Warwas and Hahn, 2009) is an environment for the development of MAS based on a domain specific modelling language for MAS that supports code generation for JACK (Winikoff, 2005) and JADE (Bellifemine *et al*., 2007) languages.

The identified methodologies internally use M2M transformations to facilitate the transition among different phases. However, the proposal presented in this work is the exploitation of M2M transformations to implement the transition between models used by different methodologies, facilitating their integration.

3. Description of the VigilAgent methodology

As aforementioned, VigilAgent has been defined to ease the development of agent-oriented monitoring applications. This aim entails five phases that are briefly described as follows:

1. *System specification*. In this phase, the analyst specifies both the requirements and the environment of the system in hand. They are obtained after several meetings arranged with the customers.
2. *Architectural design*. During this phase, the system architect determines what kind of agents the system needs and how the interaction between them has to be.
3. *Detailed design*. The agent designer and application designer collaborate to specify the internal structure of each entity that makes up the system with regard to the architecture produced in the previous phase.
4. *Implementation*. The software developer generates and completes the application code.
5. *Deployment*. The deployment manager opens out the application according to a specified deployment model.

At this point, several issues have to be spelled out about the identified phases. First, the phases named *system specification* and *architectural design* in VigilAgent correspond to the two first phases of the Prometheus methodology (Padgham and Winikoff, 2004). Another important detail is that the third phase of VigilAgent, named *detailed design*, uses INGENIAS models (Pavón *et al*., 2005) in its definition. Finally, notice that the code generated by VigilAgent is for its deployment in the ICARO framework (Gascueña *et al*., 2010). Several reasons have conducted to this integration:

- One of the main advantages of the exploitation of Prometheus is the guidelines offered to identify both the agents and their interactions. Another advantage of Prometheus is the explicit use of the *scenario* concept, which is also widely exploited in the monitoring domain. Indeed, a monitoring 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 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 (Bordini *et al*., 2005). This means, in principle, a loss of generality. However, INGENIAS does facilitate a general process to transform models specified during the design phase into executable code. Unfortunately, INGENIAS does not offer guidelines to identify the entities of the model, but the developer's experience is necessary for their identification. Therefore, VigilAgent methodology is not developed from scratch but integrates facilities of Prometheus and INGENIAS to take advantage of both.

- Regarding the *implementation* phase, the ICARO framework has been selected because it already provides high-level software components that facilitate the development of agent applications. Also, it is independent of the agent architecture, that is, the developer creates new architectures for their integration in the framework. This is a clear difference regarding other agent frameworks such as JACK or JADE (Bordini *et al*., 2005), which provide a middleware to establish the communications among agents instead of an extensible architecture. An additional advantage is that this framework already implements functionalities to automatically carry out component management, application initialization and shutdown, which are not usually provided by other frameworks. Thus, the ICARO framework enables the developers to reduce their workload and guaranty that all components are under control.

A comparison between the agent-oriented methodologies previously cited and other agent methodologies, as well as a comparison between ICARO and other agent programming languages, has recently been introduced (Gascueña *et al*., 2011).

Lastly, different tools support the development of monitoring applications according to the VigilAgent phases. First, the Prometheus design tool (PDT) (Padgham *et al*., 2008) is used during the *system specification* and *architectural design* phases to specify the Prometheus model of the system. Then, the specified Prometheus model is transformed to an INGENIAS model by executing the M2M transformation with the Medini QVT tool (QVT, 2012), (Pardillo *et al*., 2010). Afterwards, according to the *detailed design* phase, the INGENIAS model generated is completed by using the INGENIAS development kit (IDK) (Gómez-Sanz *et al*., 2008). Finally, the detailed INGENIAS model is transformed into code for the ICARO framework by running our *INGENIAS ICARO framework* (IIF) generator on the IDK.

4. Case study: access control

Access control is the usual and basic term used for monitoring and controlling the entrance to and exit from a specific area. In a previous work (Gascueña *et al*., 2011), we illustrated how to use VigilAgent to develop an intelligent system for access control. It automatically controls entrances/exits of humans to/from an enclosure throughout the installed modules (see Figure 1(a)). Each one of the modules enables the entrances and exits according to their configuration and is composed of the following elements (see Figure 1(b)): a reader device, an automatic door, a contact sensor and an infrared sensor. To go in/out of the enclosure throughout a module, the user inserts a ticket into the reader device that the system verifies against the users' database. Then, a light-emitting diode (LED) illuminates in green if the user is authorized; otherwise, it illuminates in red. If the user is authorized, then the door is opened and closed once the user has crossed the door or too much time has elapsed.

In addition, the system collects and shows statistics to the guard about the number of humans crossing each door and the number of humans located inside the enclosure by using the infrared sensor installed in each module. It has also to control anomalous situations, such as tailgating or when a human blocks a door that the system has opened. A tailgating situation is detected when a 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.

Sections 6 and 7 focus on providing a detailed description about the fundamentals of the M2M and M2T transformations implemented in VigilAgent by using examples of the presented case study. Previously, a summary of the final *system overview diagram* of the access control system obtained as result of the *architectural design* phase is introduced in the following section.

5. System overview diagram

The *system overview diagram* is the final artefact obtained as result of the two first phases of VigilAgent, that is, the *system specification* and *architectural design* phases (Gascueña *et al*., 2011). It provides an overview of the internal system architecture and is the starting point to execute the transformations explained in the following sections. Figure 2

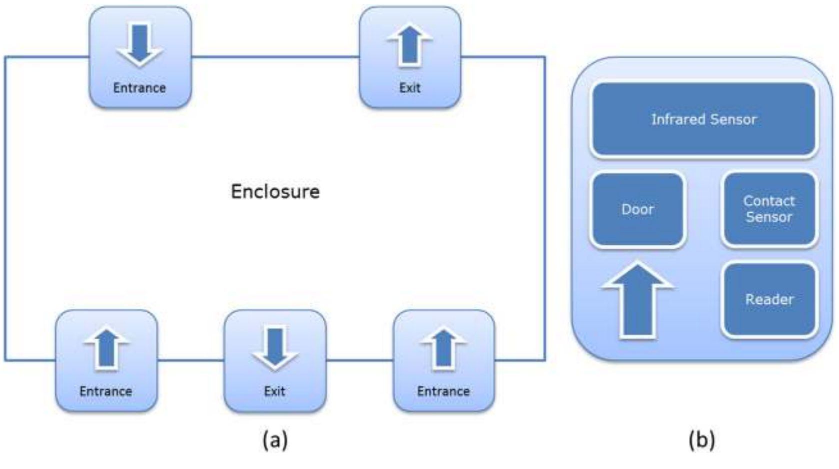


Figure 1: *The case study: (a) Surveillance environment. (b) Module devices.*

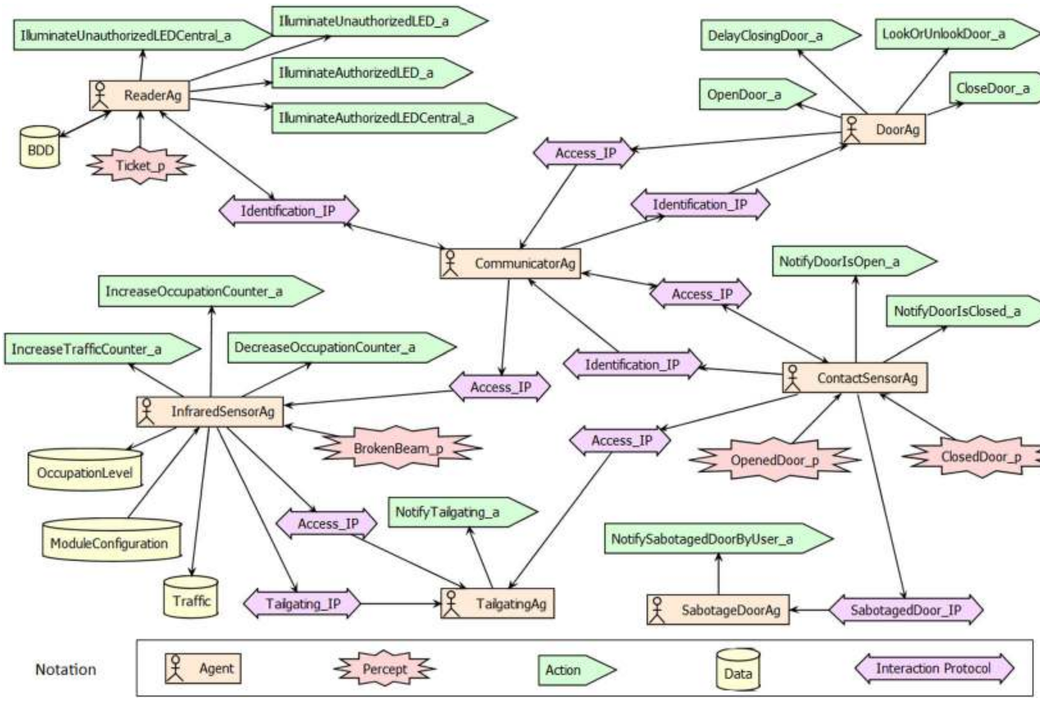


Figure 2: *System overview diagram.*

shows the *system overview diagram* of the case study presented in the previous section. Several kinds of entities are used to describe this diagram:

- *Agent*. It is 'an entity that performs a specific activity in an environment that is aware of and responds to changes' (Sterling and Taveter, 2009). The communications between the identified agents are summarized as follows. All agents that perceive information captured by a device such as *ReaderAg* or *DoorAg* use *CommunicatorAg* as an intermediate agent to establish their communication. Moreover, the *TailgatingAg* and *SabotageDoorAg* agents, responsible for detecting anomalous situations, use the knowledge offered by the *InfraredSensorAg* and *ContactSensorAg* agents related to the infrared and contact sensors, respectively, to carry out their tasks.
- *Percepts*. The information that comes from the environment is identified as *percepts*. In Figure 2, examples of *percepts* are 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*).
- *Actions*. Every operation that the system requests to the actors is identified as an action. Examples of actions are the *OpenDoor_a* and *CloseDoor_a* commands issued to open and close a door, respectively, and the *NotifySabotagedDoorByUser_a* and *NotifyTailgating_a* alert messages displayed on the user interface to notify that an anomalous situation occurred.
- *Data*. *Data* are either information managed by agents or beliefs representing their knowledge about the environment or themselves. For example, the *BDD* data validate every introduced ticket. It is worth noting that *data* have different granularity. For instance, in the access control case study, *Traffic* data is used to count how many people cross each module, *OccupationLevel* data is employed to have control on the enclosure capacity level and *ModuleConfiguration* data is utilized to determine if a user goes in or out the enclosure.
- *Actors*. An actor is an external entity – human or software/hardware – that interacts with the system. In the access control case study, an actor is specified for each device of a module (reader, door, infrared and contact sensors). There is also a *Central_A* actor (see Figure 3) 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 or she can send to the system to enable/disable modules. Moreover, the actors specified in the *interaction protocol*s are also used in the *system overview diagram*. For instance, the *Infrared_A* and *Central_A* actors are specified in the *Tailgating_IP* interaction protocol shown in Figure 3. This interaction protocol is also identified in the *system overview diagram* illustrated in Figure 2.

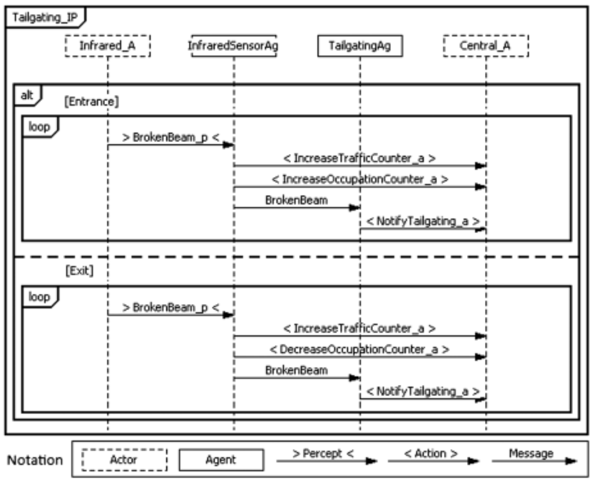


Figure 3: *Tailgating interaction protocol.*

- *Message*. This entity is used in the interaction protocol*s* to represent a communication among agents. For instance, the *BrokenBeam* message, shown in Figure 3, communicates that the infrared sensor beam has been broken.
- *Interaction protocols*. These are a graphical representation that shows (1) interactions among agents and (2) 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, Figure 3 details the internal structure of the *Tailgating_IP* interaction protocol, a sub-protocol of *Access_IP*. As can be noticed, it involves two agents (*InfraredSensorAg* and *TailgatingAg*) and two actors (*Infrared_A* and *Central_A*) that interact to detect a tailgating situation. This situation happens when a door remains open even though an authorized person has already crossed it and the related infrared sensor has been activated once. At this moment, when the *InfraredSensorAg* agent perceives that the infrared sensor beam is 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 notifies the *Central_A* actor an unauthorized access by means of the *NotifyTailgating_a* action. This checking is carried out until the *ContactSensorAg* agent notifies the *TailgatingAg* agent that the door is closed thanks to the *Access_IP* protocol. The defined *Tailgating_IP* interaction protocol detects unauthorized entrances and exists.

6. Model-to-model transformation: from Prometheus to INGENIAS

The M2M transformation from Prometheus to INGENIAS is one of the major aspects and contributions of our VigilAgent methodology. The proposed mappings have mainly been inferred from the documentation of the Prometheus and INGENIAS methodologies, the entities and their relations supported by the PDT and IDK tools, and our own experience (Gascueña and Fernández-Caballero, 2007, 2011). The process of identification of mappings consists in analysing each Prometheus concept and inferring how it can be modelled using an INGENIAS equivalent structure. Franklin and Graesser (1996) defined an *autonomous agent* as ‘a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future’. This definition has been exploited to identify the key concepts that must be taken into account to specify the mappings between INGENIAS and Prometheus: the agent’s perception and action, the interchanged messages to satisfy its objectives and the information to be recorded about both its environment and itself. The next sections describe four conceptual mappings to transform the structures that involve percepts, actions, messages and data related to agents.

The conceptual mapping has been automated specifying an M2M transformation by using the QVT language (OMG, 2008). This language has been selected as it offers characteristics such as directionality and traceability, which are not available in other languages such as ATL (Jouault *et al*., 2008) or XSLT (Tidwell, 2008). The first characteristic means that transformations are executed in both directions, whereas the second one enables to generate a traceability model that establishes relationship among the involved models. Navarro (2007, p. 248) has introduced a comparison illustrating why QVT is the most complete option to perform model transformations versus other model transformation languages, according to the analysed features. Currently, in VigilAgent, a transformation has been defined to generate INGENIAS models from Prometheus models, but in the next future, we want to take advantages of these features to exploit the traceability both top–down and bottom–up between Prometheus and INGENIAS models.

In the following, Entity 1 – *RelationX* ➔ Entity 2 notation is used to specify that Entities 1 and 2 are related through the relation *RelationX*. The direction of the arrow is a graphical representation of the relation that links both entities. In the figures that describe the identified mappings, dotted arrows have been used to stand out how the Prometheus entities are transformed into equivalent INGENIAS entities. Moreover, a table below each figure provides information about the related models. Finally, the notation to represent the entities used in PDT and IDK is shown in Figure 4.

6.1. Mapping Prometheus percepts

A Prometheus *percept* is a piece of information from the environment received through a sensor. Percepts are sent by actors (Actor ➔ Percept) and are received by agents (Percept ➔ Agent). The relations between actors and percepts (Actor ➔ Percept) are specified in the Prometheus interaction protocol diagrams and the relations between percepts and agents (Percept ➔ Agent) are described in the Prometheus *system overview diagram*. Figure 5 depicts an example of their use for the access control case study. It shows how *InfraredSensorAg* agent perceives from the environment that a beam has been broken, that is, the *Infrared_A* actor sends a *BrokenBeam_P* percept, containing the captured signal by the infrared device, to the *InfraredSensorAg* agent.

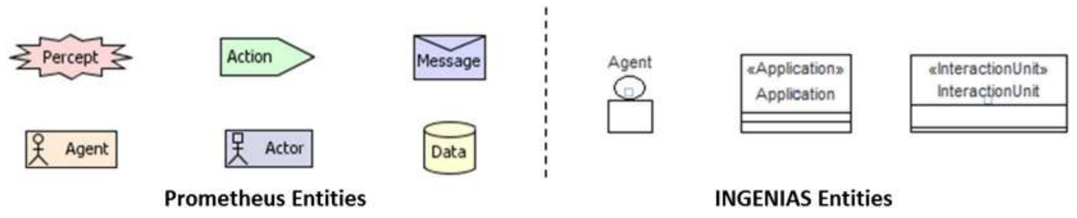
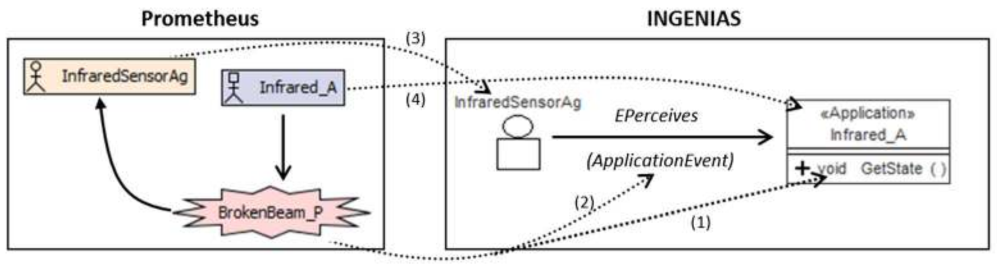


Figure 4: *Graphical representation of Prometheus and INGENIAS entities.*



Prometheus	INGENIAS
Agent ← Percept ← Actor - Percept descriptor: *Information carried* field - System overview diagram: Agent ← Percept - Interaction protocol: Percept ← Actor	Environment model Agent – *EPerceives* → Application - If information carried is not null then type of event *ApplicationEventSlots* associated to EPerceives relation. On the contrary, type of event *ApplicationEvent* is chosen.

Figure 5: *Mapping Prometheus percepts into INGENIAS.*

In INGENIAS, if a software and/or hardware element interacts with the system in hand and cannot be designed as an agent, then it is specified as an application. According to the second consistency criterion of the INGENIAS *environment model*, every agent that perceives changes in the environment must be associated to an *application* in the environment model. Therefore, as arrow 1 in Figure 5 shows, every percept of a Prometheus agent is mapped to an *operation* of an INGENIAS application.

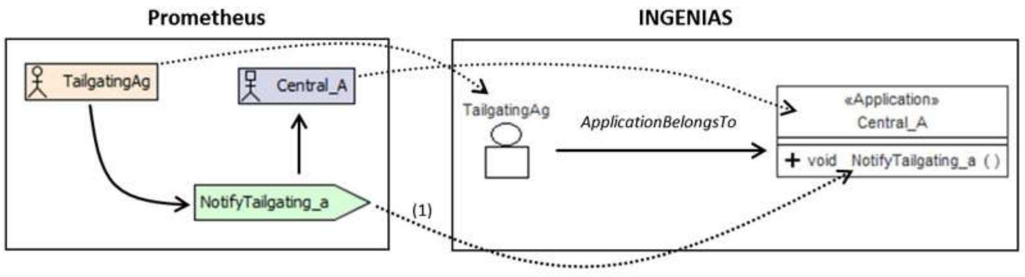
A Prometheus percept has a field, *Information carried*, to specify the carried information. If *Information carried* is null, then, as depicted by arrow 2 in Figure 5, this is described in INGENIAS as a type of event named *ApplicationEvent*, which is associated to the *EPerceives* relation established between the agent and the corresponding application. If *Information carried* is not null, then *ApplicationEventSlots* is used instead of *ApplicationEvent* (e.g. to translate the *Ticket_P* percept). Notice that Prometheus agent and actor concepts are directly mapped to INGENIAS agent and application concepts (see arrows 3 and 4 in Figure 5, respectively). The basic elements of these equivalencies are graphically and textually summarized in Figure 5.

6.2. Mapping Prometheus actions

A Prometheus *action* represents an interaction between an agent and its environment. This is represented in Prometheus through the structure Agent ➔ Action ➔ Actor. For instance, Figure 6 illustrates that a *TailgatingAg* agent sends a *NotifyTailgating_a* action to the *Central_A* actor to display an alert message on the user interface that notifies a tailgating situation. Therefore, a Prometheus action on the environment is transformed into an operation of an INGENIAS application specified in the INGENIAS *environment model* (arrow 1 in Figure 6). In addition, an Agent – *ApplicationBelongsTo* ➔ Application relation is established to specify that an agent uses an application. All these information are summarized in Figure 6.

6.3. Mapping Prometheus data

In Prometheus, *data* represent either external information used by an agent or beliefs describing the agent's knowledge about the environment or itself. It is worth noting that data both represent a simple data type (e.g. a string or Boolean)



Prometheus	INGENIAS
Agent → Action → Actor - System overview diagram: Agent → Action - Interaction protocol: Action → Actor	Environment model Agent – *ApplicationBelongsTo* → Application

Figure 6: *Mapping information related with Prometheus actions into INGENIAS.*

or a complex data type (e.g. database). Figure 7 shows one of the data used in the access control case study. As it can be observed, the *ReaderAg* agent writes and reads information on the user's database (*BDD* data) to check whether the ticket introduced by a user is valid or not. In the Prometheus *system overview diagram*, the relations Agent➔Data and Data➔Agent express that an agent reads and writes a data, respectively. In the INGENIAS *environment model*, those Prometheus data with a coarse granularity (e.g. a database or a complex data structure to store non-persistent information) are translated to Agent – *ApplicationBelongsTo*➔Application. Notice that the INGENIAS application is not the data itself but the entity that provides methods for its management (arrow 1 in Figure 7). All these equivalencies are graphically and textually summarized in Figure 7.

6.4. Mapping Prometheus messages

A Prometheus message represents a communication among agents. When the *system overview diagram* and interaction protocols are specified, the AgentS➔Message➔AgentT structure is used to represent that the AgentS sends a Message to the AgentT. For instance, it can be observed in Figure 8 that the *ReaderAg* agent sends a *ValidIdentification* message to the *CommunicatorAg* agent with information useful to start the process of opening the door because the user has been properly authenticated. In INGENIAS, the term interaction unit is used instead of message (arrow 1 in Figure 8). Thus, it has been considered that AgentS *UIInitiates*← InteractionUnit – *UIColaborates*➔AgentT in the INGENIAS interaction model is the equivalent structure. If the message contains some information, then this is expressed in INGENIAS by means of a *FrameFact* included in the interaction unit. Figure 8 summarizes these equivalencies.

6.5. Automating the Prometheus-to-INGENIAS transformation

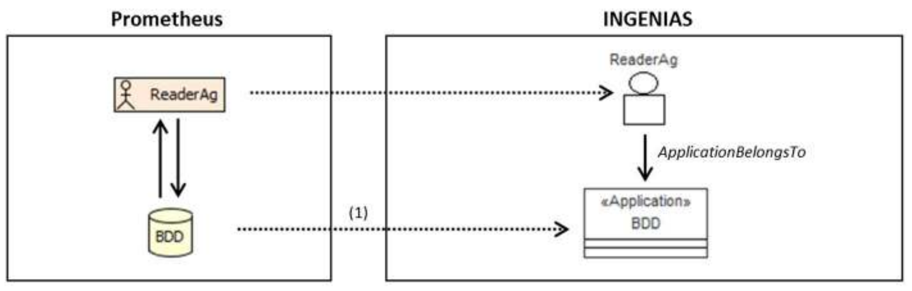
As stated before, the QVT relations language has been used to define the transformation *prometheus2ingenias*, which allows us to execute the mappings previously described in an automatic way. Medini tool (QVT, 2012) is the engine used to execute this transformation. Five inputs are required by Medini to execute the *prometheus2ingenias* transformation: the Prometheus meta-model, the Prometheus model to be transformed, the INGENIAS meta-model, the name of the INGENIAS model to be generated and the transformation defined using QVT relations. The basic idea of a meta-model is to identify the main concepts and their relations of a given problem domain that are used to describe models of that domain. Both Prometheus and INGENIAS meta-models have been described using Ecore (Steinberg *et al.*, 2009), a language to describe meta-models.

As shown next, the *prometheus2ingenias* transformation has two typed candidate models: *pro* represents any model that conforms to the Prometheus meta-model (Gascueña *et al.*, 2012), and *ing* represents any model that conforms to the INGENIAS meta-model (Fuentes-Fernández *et al.*, 2010).

```
transformation prometheus2ingenias (pro:prometheus,
                                     ing:ingenias_model)
```

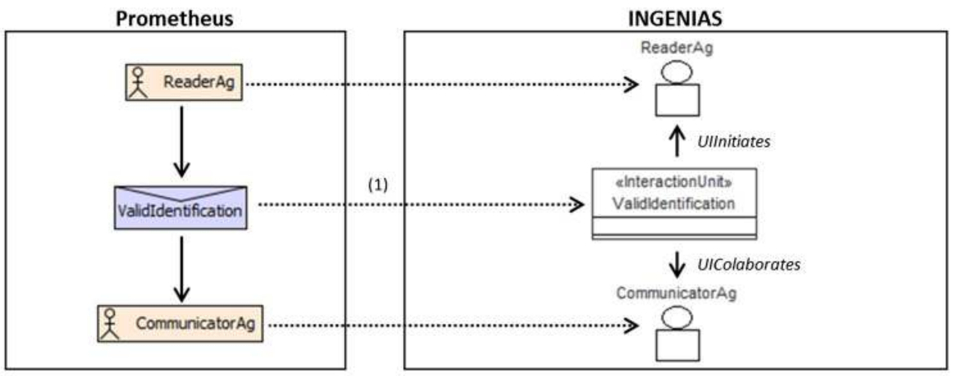
A transformation is made up of a set of relations, specifying each of them one or several mappings between the candidate models. For example, the *MapDataToApplication* and *MapAgentToAgent* relations have been specified to automate the mapping described in Section 6.3 and are illustrated next:

- The *MapDataToApplication* relation has two domains, *pro* and *ing*, which are previously specified in the declaration of the transformation. The relation imposes a pattern on every domain, describing the constraints to be satisfied by the elements of the involved model. When the elements contained in each candidate model simultaneously fulfil their corresponding patterns, then the matching happens, and the relation is successfully executed; otherwise, it fails, and it is not executed. In the *pro* domain, the relation establishes that every Prometheus data (i.e. all elements of type *data*) must be retrieved to be used. But, the pattern also imposes a condition that defines that name and description of these data are bound to the *nd* and *ded* variables, respectively. Simultaneously, in the *ing* domain, every INGENIAS application (i.e. all elements of type *Application*) has the attributes *id* and *description* bound to the same variables *nd* and *ded*. It can also be observed that this relation has a when clause, which describes a condition that must be



Prometheus	INGENIAS
- System overview diagram: Agent → Data; Agent ← Data Data has coarse granularity	Environment model Agent – *ApplicationBelongsTo* → Application

Figure 7: *Mapping information related with Prometheus data into INGENIAS.*



Prometheus	INGENIAS
- System overview diagram and Interaction Protocol AgentS → Message Message → AgentT	Interaction model AgentS ← *UIinitiates* – Interaction Unit Interaction Unit – *UIColaborates* → AgentT
AgentS and AgentR are sender and receiver agents, respectively	Interaction Unit includes a FrameFact if message contains information

Figure 8: *Mapping information related with Prometheus message into INGENIAS.*

held before the relation can be successfully applied. In this case, the when clause specifies that the relation is only applied for data entities, which have a defined data type equals to *DB* string.

```
top relation MapDataToApplication {
    nd : String ;
    ded : String ;
    checkonly domain pro d:prometheus::Data
    {
       name = nd,
       description = ded
    };
    enforce domain ing ap:ingenias_model::entities::Application
    {
       _view_type = 'INGENIAS',
       id = nd,
       description = ded,
       specification = spec:ingenias_model::
          specification::Specification{name=''}
    };
    when{
           (not d.dataType.oclIsUndefined()) and
            d.dataType.trim().equalsIgnoreCase('DB');
    }
}
```

In QVT relations, a relation can be defined either to check the models for consistency or to enforce the consistency by modifying the model selected as target. Therefore, every pattern can be evaluated using two different modes: *checkonly*, which just checks if the pattern is satisfied, reporting an inconsistency otherwise, and *enforce*, which first checks whether the pattern is satisfied and then creates, modifies or erases elements in the target model, as it is necessary to ensure the consistency. In the *MapDataToApplication* relation, the *pro* domain has been marked as *checkonly*, but the *ing* domain has been marked as *enforce*. This means that when the transformation is executed, it is checked whether there are elements in the target INGENIAS model that satisfy the *MapDataToApplication* relation, that is, the pattern described for its *ing* domain. If this is not the case, elements in the target INGENIAS model will be created, deleted or modified to enforce the consistency. This allows either to generate the target INGENIAS model or to check whether inconsistencies emerge between the generated INGENIAS and the Prometheus source models.

- The interpretation of the *MapAgentToAgent* relation is similar to the previous one. Let us highlight that a where clause has been defined to specify a condition that must be held once, the relation has been successfully executed. This is where clause specifies that other relations have to be called once the relation *MapAgentToAgent* is satisfied. For example, *MapAgentWrittenToApplication* and *MapAgentReadToApplication* relations establish the connections between the agents and applications related to the data conceptual mapping.

```
top relation MapAgentToAgent {
    nagp : String ;
    deagp : String ;
    checkonly domain pro agp:prometheus::Agent
    {
       name = nagp,
       description = deagp
    };
    enforce domain ing agi:ingenias_model::entities::Agent
    {
       _view_type = 'INGENIAS',
       id = nagp,
       description = deagp,
       specification = spec:ingenias_model::
          specification::Specification{name=''}
    };
    where{
       MapAgentWrittenToApplication(agp, spec);
       MapAgentReadToApplication(agp, spec);
       MapAgentSentMessageToUIInitiates(agp,spec);
       MapAgentReceivedMessageToUIColaborates(agp,spec);
       MapAgentActionToApplicationBelongsTo(agp,spec);
       MapAgentPerceptToEPerceives1(agp,spec);
       MapAgentPerceptToEPerceives2(agp,spec);
    }
}

relation MapAgentReadToApplication{
    nagp : String ;
    nd : String ;
    idapbtr : String ;
    checkonly domain pro agp:prometheus::Agent
    {
       name = nagp,
       readData = dataW:prometheus::Data{
          name = nd,
          dataType = 'DB'
       }
    };
    enforce domain ing spec:ingenias_model::
       specification::Specification
```

```
        {
           SRelations = apbt:ingenias_model::relations::
              ApplicationBelongsTo{
                 _view_type = 'INGENIAS',
                 id = idapbtr,
                 ApplicationBelongsTosource=apbts:
                 ingenias_model::association_end::
                    ApplicationBelongsTosource{
                       _view_type = 'INGENIAS',
                       attributeToShow = '0',
                       entity = agi:ingenias_model::entities::
                          Agent{id = nagp},
                       ApplicationBelongsTosourceAgent = agi
                    },
                 ApplicationBelongsTotarget=apbtt:
                 ingenias_model::association_end::
                    ApplicationBelongsTotarget{
                       _view_type = 'INGENIAS',
                       attributeToShow = '0',
                       entity = ap:ingenias_model::
                          entities::Application{id = nd},
                       ApplicationBelongsTotargetApplication = ap
                    }
              }
        };
        when{
              idapbtr = nagp + '_r_' + nd;
        }
    }
```

7. Model-to-text transformation: from INGENIAS to ICARO framework

The M2T transformation from INGENIAS to ICARO is the other major contribution of the VigilAgent methodology. The development of IDK modules to support ICARO as the target platform for the implementation of multi-agent system applications has followed a *bottom–up* approach. First, the INGENIAS structures necessary to specify all the concepts and relations of an application implemented in ICARO have been identified. Then, a module has been gradually developed, which automates the task of ICARO code generation from INGENIAS specifications aligned with the identified conceptual relations. Finally, a new module has been developed to provide the ability to upgrade the specification of a model according to the changes made in the implementation. A detailed description of the general process for developing IDK modules can be found in Pavón *et al*. (2006). The next section provides a description about the relations found between the INGENIAS and ICARO concepts, and Sections 7.2 and 7.3 describe the modules developed to generate and update code for ICARO, respectively.

7.1. Conceptual relation between INGENIAS and ICARO

First, it is worth describing some details of the figures used in this section to illustrate the relationship between INGENIAS and ICARO. The left side of the figures provides the notation used to express a fragment by using INGENIAS, whereas the right side of these figures corresponds to the notation chosen to express the same fragment of a model using ICARO concepts. Any communication between the components implemented to develop an executable ICARO application is summarized as follows. First, an *event* is an entity for exchanging information between the producer of the event and the potential receivers. An event is used for communication and information delivery from a *resource* to its agent or among agents. Thus, an agent sends events through its *use interfaces*, and similarly, a resource also employs the use interface of an agent to send an event. Second, an agent uses the use interface of a resource to request the services (methods) offered.

From our point of view, the INGENIAS concepts of agent and application can be mapped to the ICARO concepts of reactive application named agent and resource, respectively. For example, when an *ApplicationBelongsTo* relationship is established between an agent and an application, it is understood that the agent uses the services offered by the resource (see Figure 9). In particular, the actions that the agents execute on the environment are represented by this structure. Services are modelled as application methods.

In ICARO, a resource sends information to an agent. However, in INGENIAS, an agent perceives information from an application (see Figure 10). For this reason, the following mappings have been established: (1) for every INGENIAS *EPerceives* relationship between an agent and an application modelled as an event of type *ApplicationEvent* (to denote that such event has no information), a signal between the corresponding ICARO resource and agent is established, and (2) for every INGENIAS *EPerceives* relationship between an agent and an application modelled as an event of type *ApplicationEventSlots* (to denote that such event does have information), an ICARO entity event is created to specify the sending of such information between the corresponding ICARO resource and agent.

In INGENIAS, the communication among agents is specified using an entity of type *InteractionUnit* related to the producer and consumer agents by means of relationships *UInitiates* and *UICollaborates*, respectively (see Figure 11). If the producer agent sends information, then this is included in the interaction unit by means of an entity of type *FrameFact*, which contains the necessary slots to transport it. This is graphically denoted in the interaction unit because it shows an *Info* attribute (the value shown is the identifier of the *FrameFact*). This communication is mapped to ICARO by creating an entity event to specify the sending of such information between the corresponding ICARO agents. Conversely, if the INGENIAS producer agent does not send information, then the interaction unit does not include a *FrameFact*. This communication will be mapped to ICARO by creating a signal between the corresponding ICARO agents.

The behaviour of an ICARO reactive agent is specified by means of an automaton. This is generated in an automatic way from the INGENIAS *state diagram* of the related INGENIAS agent. In particular, the following five structures, available in INGENIAS *state diagram*, have been identified to generate any ICARO automaton (see Figure 12):

- To generate the ICARO initial state, a *starts* relationship between an INGENIAS *InitialNode* entity and the initial state must exist.
- To generate an ICARO final state, an INGENIAS *ends* relationship has to be specified from the final state to the *EndNode* entity.
- To generate a transition between two different states in ICARO, an INGENIAS *WFollowGuarded* relationship between the corresponding states has to be specified using the syntax event/semantic action in its *Condition* attribute. The event represented in the INGENIAS *state diagram* is related to an *ApplicationEvent* or *ApplicationEventSlots*

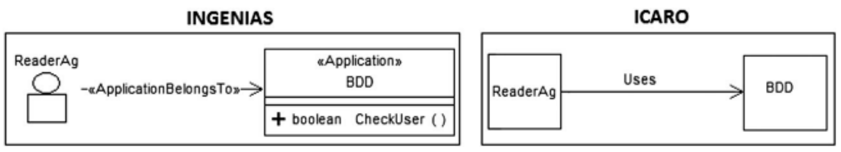


Figure 9: *Modelling an agent using resource's services.*

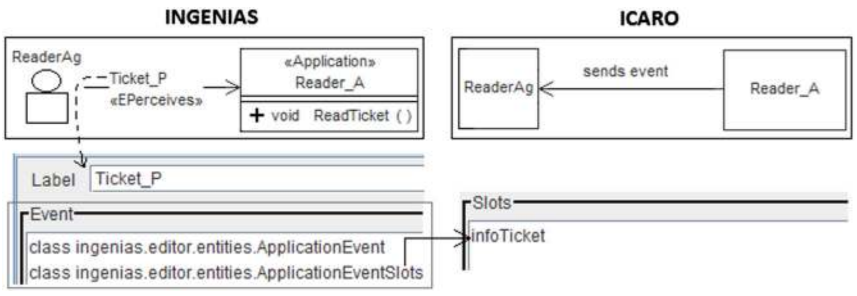


Figure 10: *Modelling an agent's perception.*

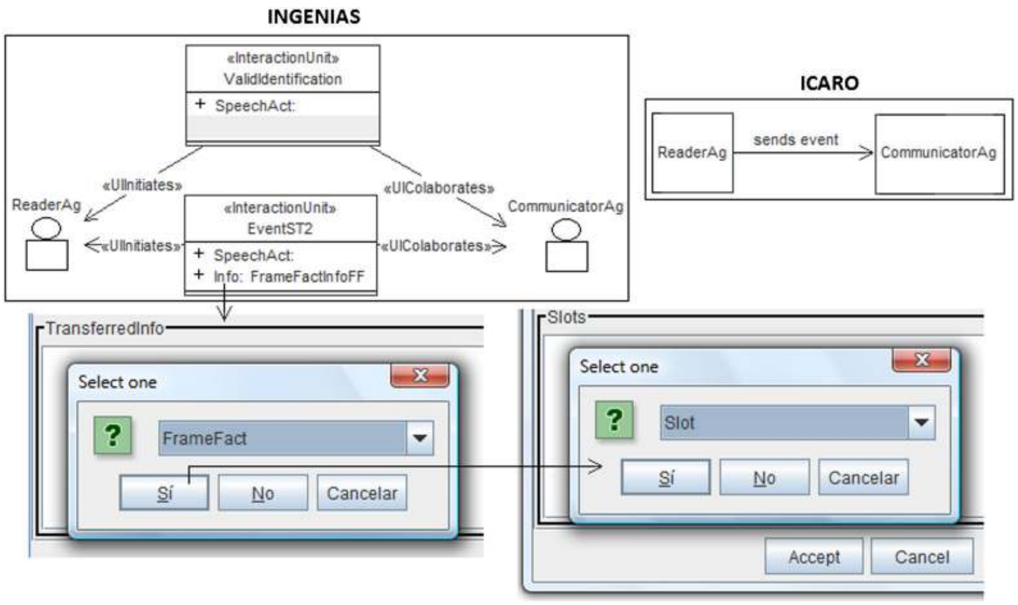


Figure 11: *Modelling a communication among agents.*

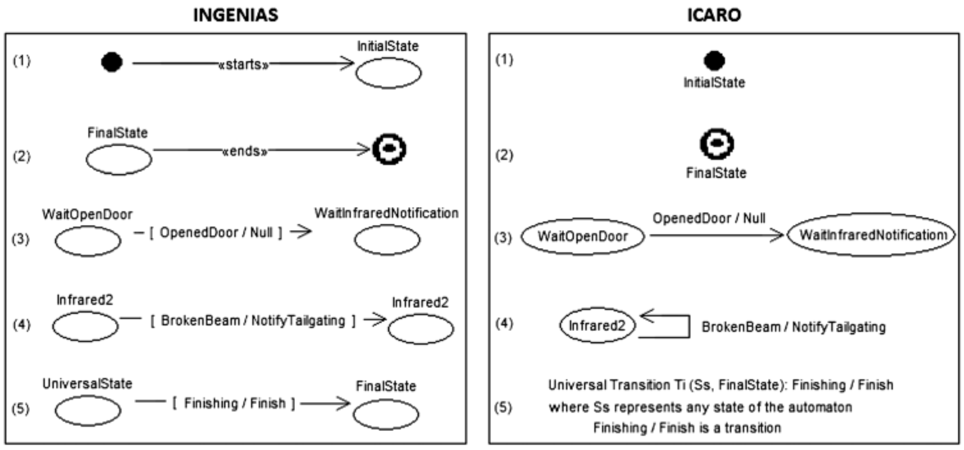


Figure 12: *Modelling an agent's automaton.*

entity when the event is sent by a resource (see Figure 10) or to an *InteractionUnit* entity if the event is sent by an agent (see Figure 11).

- INGENIAS does not enable to explicitly represent relationships that cycle over the same entity. To fix this issue, the fourth structure has been considered: (1) a copy of the state to be iterated over is made; (2) a *WFollowGuarded* relation is established from the copied state to the original state; and (3) a transition is specified following the syntax described in the previous structure. This INGENIAS structure is transformed to ICARO by specifying a cycle over the same state.
- Universal transitions of the automaton of an ICARO reactive agent are valid for any state of the automaton. This means that when the finishing event happens, its related actions are executed and the next state is reached, regardless of the automaton current state. To represent universal transitions in INGENIAS, the adopted solution is to use a state that is always be named *UniversalState* and plays the role of the *source* state of a universal transition. Obviously, *UniversalState* cannot be used for a different purpose in any automaton.

Finally, notice that the agent's name is assigned to the state diagram as a criterion to identify the agent's behaviour.

In ICARO, the organization of an application is described by means of deployment diagrams where the number of instances of each type of agent and resource is specified. However, taking into account that ICARO resources are generated from INGENIAS applications and that the number of INGENIAS application instances cannot be specified in INGENIAS deployment diagrams, an alternative solution is taken, namely to use INGENIAS environment models. These models are specified carrying out the following steps: (1) copy all the agents and applications in an environment model; (2) relate them to entities of type *UMLComment*; and (3) set the attribute text of *UMLComment* to the number of instances to be deployed. Obviously, this process is repeated to create as many deployment configurations as necessary.

7.2. *Code generation*

To generate code for the ICARO framework, the IIF generator IDK module has been developed. For this aim, the *IIFGenerator* class has been extended so that its constructors have the templates that the IIF module uses, in a similar way to any other IDK code generator. Additionally, the extended IIFGenerator class also implements the abstract methods defined in *BasicCodeGeneratorImp*. It is worth noting that the development of the IIF module has been simplified by defining a template for each ICARO artefact (see Table 1).

The IDK templates for code generation have both source code written in the programming language of the target platform and tags to establish where the model information is used during the generation of code. The kinds of tags to be used in an IDK template are very limited (Gómez-Sanz, 2008): *program* is the main tag of the document, *repeat* means that the text enclosed by this tag has to be copied and pasted to have a duplicate, *v* represents a variable and *saveto* is used to save the enclosed text into a file. Therefore, it can be stated that the IDK code generation technology is more straightforward and easy to learn than other technologies for code generation, such as XSLT (Willians, 2009), Java Emitter Template (Vogel, 2009) or XPAND (Gronback, 2009). However, IDK exhibits a disadvantage: it does not allow developers to reuse templates so that fragments have to copied and pasted to be reused, thus hindering the code generator maintainability.

Next, the elements used by the IIF module to generate code for the intermediate states of an ICARO reactive agent automaton are shown. With this aim, the *automaton* template specifies the following pattern: for each (first repeat) intermediate state defined by the *intermediateState* variable, generate code for each (second repeat) transition that starts from such an intermediate state.

```
@@@repeat id="intermediateStates"@@@
<state intermediateId="@@@v@@@intermediateState@@@/v@@@">
   @@@repeat id="isTransitions"@@@
   <transition input="@@@v@@@event@@@/v@@@"
               action="@@@v@@@action@@@/v@@@"
               nextState="@@@v@@@nextState@@@/v@@@"/>
   @@@/repeat@@@
</state>
@@@/repeat@@@
```

When the IIF module is executed using as input an INGENIAS model, a sequence of data is generated. For instance, in the following, a sketch of the sequence of a generated agent automaton is shown.

```
<repeat id="intermediateStates">
   <v id="intermediateState">Infrared2</v>
   <repeat id="isTransitions">
      <v id="event">BrokenBeam</v>
      <v id="action">NotifyTailgating</v>
      <v id="nextState">Infrared2</v>
   </repeat>
   <repeat id="isTransitions">
      <v id="event">ClosedDoor</v>
      <v id="action">Null</v>
      <v id="nextState">WaitOpenDoor</v>
   </repeat>
</repeat>
```

Table 1: *Description of the templates*

Template	Description
Automaton	Used to generate an XML file with the automata agents code.
Semantic Action	Employed to generate the code of the classes that implement the agents' semantic action.
ResourceGeneratorClass	Used to generate the code of the classes that implement the use interfaces of the resources. The code of methods and parameters of these classes is automatically generated as well.
ResourceIseItf	Used to generate the code of the use interfaces of the resources.
Deployment	Employed to generate an XML file with the organization of the ICARO application under development.

Finally, it is worth noting that the IIF module performs a matching between the templates and the data retrieved from the model. Next, following our example, the code generated by IIF is shown for an *Infrared2* intermediate state and two transitions that start from it. The first one means that when the infrared beam is broken (*BrokenBeam*), then the agent remains in the *Infrared2* state and executes a *NotifyTailgating* action to notify that a tailgating situation is happening. The second one means that *TailgatingAg* agent changes to *WaitOpenDoor* state when it is notified that the door has been closed again.

```
<state intermediateId="Infrared2">
   <transition input="BrokenBeam" action="NotifyTailgating"
               nextState="Infrared2"/>
   <transition input="ClosedDoor" action="Null"
               nextState="WaitOpenDoor"/>
</state>
```

7.3. *Code update*

Another important aspect to consider when developing a code generator is to provide developers with facilities that prevent manually written code from being overridden by subsequent generator runs. The solution has been to integrate a facility in the IIF module for marking protected regions where the developers manually write code. The start and the end of a protected region is marked by means of comments. A file has as many protected regions as necessary, labelled each one with a unique identifier. For instance, the classes that implement the agents' semantic actions have a protected region for each semantic action established in the state diagram that specifies the agent automaton. The following fragment of code shows an example of this type of region, where *ACTIONID* has to be replaced with the identifier of the related semantic action (e.g. *NotifyTailgating*).

```
//#start_nodeIDACCION:ACTIONID <--ACTIONID--
//#end_nodeIDACCION:ACTIONID <--ACTIONID--
```

The IIF module uses the specification of the model to generate code. Therefore, it is necessary to store a copy of the code manually written in the protected regions. In this way, each time the IIF module is run, each protected region is overridden with the code manually written by the developer.

Another module named *ICAROTCodeUploader*, in charge of synchronizing code and design, has been developed as well. When it is executed, the design specification is updated with the regions of the generated code. This module, unlike the IIF module, does not require templates for its definition.

8. Conclusions

The learning curve of VigilAgent can be steep at first because users must be used to different terms that have the same meaning depending on the technology used in each phase (Prometheus and INGENIAS for modelling and ICARO for implementation). However, this disadvantage is overcome thanks to the two transformations that are executed automatically. First, an M2M transformation executed by means of Medini automatically transforms Prometheus structures into their equivalent INGENIAS structures. Second, the M2T INGENIAS transformation generates code for the ICARO framework from INGENIAS models. It is worth pointing out that the time spent learning how to develop and implement the IIF and the *ICAROTCodeUploader* modules described in Section 7 was 2 months and 15 days. This effort is worth as new applications are modelled and implemented with an improved productivity. The main reason is that the time necessary for coding is reduced because the developer does not need to learn the structure, location and naming rules of ICARO applications files. The developer only has to manually write the body of both the resource methods and the agents' semantic actions along with some auxiliary classes. The remaining code is automatically generated by using as input the INGENIAS models. We would also like to point out that the presented transformations have been validated in the context of two different applications developed with the VigilAgent methodology. These applications revolve around the problem of a robot team that patrols around a simulated surveillance environment to deal with the alarms that rise in such environment. The communications among robots are established by means of a blackboard in the first application (Gascueña *et al*., 2011). On the contrary, the second application (Gascueña *et al*., 2011) uses an explicit communication mechanism among robots to carry out the coordination tasks, that is, a robot explicitly sends messages to all other robots. A detailed description about the modelling phases using VigilAgent is provided in the previous references. The productivity for developing these applications was high. Indeed, most of the necessary code was automatically generated using as input the INGENIAS models during their development. Moreover, they were developed iteratively in a very easy way as the code manually written was automatically maintained thanks to the IIF module.

Finally, notice that our methodological proposal is named VigilAgent to expose one of the main motivations in our research lines: developing surveillance or vigilance (*Vigil*) systems using agent (*Agent*) technologies. However, let us highlight that the proposal can be applied in other application domains. The technologies used are of general purpose, that is, they are not tied to a specific domain. Thus, an interesting future work is to apply the methodology and proposed transformations in domains different from surveillance.

Acknowledgements

This work is partially supported by the Spanish Ministerio de Economía y Competitividad/FEDER under projects TIN2008-06596-C02-01, TIN2010-20845-C03-01, TIN2010-20845-C03-01 and TIN2012-34003.

References

Amor, M. and L. Fuentes (2009) Malaca: a component and aspect-oriented agent architecture. *Information and Software Technology*, **51**, 1052–1065.

Ayala, I., M. Amor and L. Fuentes (2011) Towards the automatic derivation of Malaca agents using MDE. *Lectures Notes in Computer Science*, **6788**, 128–147.

Bellifemine, F., G. Caire and D. Greenwood (2007) Developing Multi-agent Systems with JADE, New York: John Wiley & Sons Ltd..

Beydeda, S., M. Book and V. Gruhn (2005) Model-driven Software Development, Berlin: Springer-Verlag.

Bordini, R., M. Dastani, J. Dix and A.E.F. Seghrouchni (2005). Multi-agent Programming: Languages, Platforms and Applications. New York: Springer.

Braubach, L., A. Pokahr and W. Lamersdorf (2005) Jadex: a BDI-agent system combining middleware and reasoning, in Software Agent-Based Applications, Platforms and Development Kits. 143–168. Basel, Switzerland: Birkhäuser Verlag.

Czarnecki, K. and S. Helsen (2006) Feature-based survey of model transformation approaches, *IBM Systems Journal*, **45**, 621–646.

Franklin, S. and A. Graesser (1996). Is it an agent, or just a program?: a taxonomy for autonomous agents, *Lecture Notes in Computer Science*, **1193**, 21–35.

Fuentes-Fernández, R., I. García-Magariño, A. Gómez-Rodríguez and J. González-Moreno (2010) A technique for defining agent-oriented engineering processes with tool support, *Engineering Applications of Artificial Intelligence*, **23**, 432–444.

García-Magariño, I., R. Fuentes-Fernández and J.J. Gómez-Sanz (2011) Model transformations for improving multi-agent system development in INGENIAS, *Lectures Notes Computer Science*, **6038**, 51–65.

Garijo, F., F. Polo, D. Spina and C. Rodríguez (2008) ICARO-T user manual. Technical Report, Telefonica I + D.

Gascueña, J.M. and A. Fernández-Caballero (2007) The INGENIAS methodology for advanced surveillance systems modelling, *Lecture Notes in Computer Science*, **4528**, 541–550.

Gascueña, J.M. and A. Fernández-Caballero (2011a) Agent-oriented modeling and development of a person-following mobile robot, *Expert Systems with Applications*, **28**, 4280–4290.

Gascueña, J.M. and A. Fernández-Caballero (2011b) On the Use of Agent Technology in Intelligent, Multi-Sensory and Distributed Surveillance, The Knowledge Engineering Review **26**, 191–208.

Gascueña, J.M., A. Fernández-Caballero and F. Garijo (2010) Using ICARO-T framework for reactive agent-based mobile robots, *Advances in Soft Computing*, **70**, 91–101.

Gascueña, J.M., A. Fernández-Caballero, E. Navarro, J. Serrano-Cuerda and F. Cano (2011a) Agent-based development of multisensory monitoring systems, *Lecture Notes in Computer Science*, **6686**, 451–460.

Gascueña, J.M., E. Navarro and A. Fernández-Caballero (2011b). Agent-oriented VigilAgent methodology for model-driven development of multi-robot surveillance systems, in *IEEE International Conference on Robotics and Automation*, Workshop on Agent Technology in Robotics and Automation, Shanghai, China. Available at http://stinger.wpi.edu/icra11/Submissions/ ICRA11 AT GascuenaNavarroFernandez.pdf

Gascueña, J.M., A. Fernández-Caballero, M.T. López and A. Delgado (2011c) Knowledge modeling through computational agents: application to surveillance systems, *Expert Systems: The Journal of Knowledge Engineering*, **28**, 306–323.

Gascueña, J.M., E. Navarro and A. Fernández-Caballero (2011d) VigilAgent for the development of agent-based multi-robot surveillance systems, Lectures Notes in Computer Science **6685**, 200–210.

Gascueña, J.M., E. Navarro and A. Fernández-Caballero (2012) Model-driven engineering techniques for the development of multi-agent systems, *Engineering Applications of Artificial Intelligence* **25**, 159–173.

Gómez-Sanz, J.J. (2008) INGENIAS agent framework development guide version 1.0. Technical Report, Universidad Complutense de Madrid. Available at http://grasia.fdi.ucm.es/main/myfiles/guida.pdf

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

Gronback, R. (2009) Eclipse Modeling Project. A Domain-specific Language Toolkit. Boston: MA: Addison-Wesley.

Haering, N., P. Venetianer and A. Lipton (2008) The evolution of video surveillance: an overview, *Machine Vision and Applications*, **19**, 279–290.

Hahn, C., C. Madrigal-Mora and K. Fischer (2009) A platform-independent metamodel for multiagent systems, *Autonomous Agent and Multi-Agent Systems*, **18**, 239–266.

Jouault, F., F. Allilaire, J. Bezivin and I. Kurtev (2008) ATL: a model transformation tool, *Science of Computer Programming*, **72**, 31–39.

Kardas, G., E. Ekinci, B. Afsar, O. Dikenelli and N. Topaloglu (2009) Modeling tools for platform specific design of multi-agent systems, *Lecture Notes in Artificial Intelligence*, **5774**, 202–207.

Kleppe, A., J. Warmer and W. Bast (2003) MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc..

Kumar, P., A. Mittal and P. Kumar (2008) Study of robust and intelligent surveillance in visible and multimodal framework, *Informatica*, **32**, 63–77.

Morandini, M., L. Penserini and A. Perini (2008). Modelling self-adaptivity: a goal oriented approach, in *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-organizing Systems*, Isola di San Servolo (Venice), Italy, 469–470.

Morandini, M., C. Nguyen, L. Penserini, A. Perini and A. Susi (2011) Tropos modeling, code generation and testing with the Taom4e tool, in *Proceedings of the 5th International i* Workshop*, Trento, Italy, 172–174.

Navarro, E. (2007) ATRIUM architecture traced from requirements by applying a unified methodology. Ph.D. Dissertation, Computing System Department, University of Castilla-La Mancha.

OMG (2008) Object management group: meta object facility (MOF) 2.0 query/view/transformation specification, version 1.0. OMG document number formal/2008-04-03. Available at http:// www.omg.org/spec/QVT/1.0/PDF.

OMG (2011) MOFScript v1.4.0. http://www.omg.org/spec/QVT/ 1.0/PDF

Padgham, L. and M. Winikoff (2004) Developing Intelligent Agents Systems: A Practical Guide. New York: John Wiley & Sons.

Padgham, L., J. Thangarajah and M. Winikoff (2008) Prometheus design tool, in *Proceedings of the Twenty-third AAAI Conference on Artificial Intelligence*, Chicago, IL, 1882–1883.

Pardillo, J., J. Mazón and J. Trujillo (2010) Extending OCL for OLAP querying on conceptual multidimensional models of data warehouses, *Information Sciences*, **180**, 584–601.

Patricio, M., F. Castanedo, A. Berlanga, O. Pérez, J. García and J. Molina (2008) Computational intelligence in visual sensor networks: improving video processing systems, *Studies in Computational Intelligence*, **96**, 351–377.

Pavón, J., J. J. Gómez-Sanz and R. Fuentes (2005) The INGENIAS methodology and tools, in Agent-Oriented Methodologies, (pp. 236–276). Hershey, PA: Idea Group Publishing.

Pavón, J., J. J. Gómez-Sanz and R. Fuentes (2006) Model driven development of multi-agent systems, *Lecture Notes in Computer Science*, **4066**, 284–298.

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

QVT (2012) IKV++ Technologies Home. Available at http://www. ikv.de

Räty, T. (2010) Survey on contemporary remote surveillance systems for public safety, *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews* **40**, 493–515.

Rivas-Casado, A., R. Martínez-Tomás and A. Fernández-Caballero (2011) Multiagent system for knowledge-based event recognition and composition, *Expert Systems: The Journal of Knowledge Engineering*, **28**, 488–501.

Schmidt, D. (2006) Guest editor's introduction: model-driven engineering, *Computer*, **39**, 25–31.

Spanoudakis, N. (2009) The agent systems engineering methodology (ASEME). Ph.D. Dissertation, Université Paris Descartes.

Steinberg, D., F. Budinsky, M. Paternostro and E. Merks (2009) Eclipse Modeling Framework, 2nd *edn*, Boston, MA: Addison-Wesley.

Sterling, L. and K. Taveter (2009) The Art of Agent-oriented Modeling. Cambridge, MA: The MIT Press.

Tidwell, D. (2008) XSLT, 2nd *Edn*, Sebastopol, CA: O'Reilly Media, Inc..

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

Warwas, S. and C. Hahn (2009) The dsml4mas development environment, in *Proceedings of the 8th Conference on Autonomous Agents and Multi-agent Systems*, Hungary: Budapest 1379–1380.

Willians, I. (2009). Beginning XSLT and XPath: Transforming XML Documents and Data, Indianapolis, IN: Wiley Publishing, Inc..

Winikoff, M. (2005) Jack intelligent agents: an industrial strength platform, in Multi-agent Programming Languages, Platforms and Applications, (pp 175–193). New York: Springer.

The authors

José Manuel Gascueña

José M. Gascueña received his MSc in Computer Science from the University of Castilla–La Mancha at the Superior Polytechnic School of Albacete, Spain, in 2004. In 2006, he received a scholarship from the Spanish Junta de Comunidades de Castilla–La Mancha. In 2010, he received his PhD from the University of Castilla–La Mancha in applying multi-agent systems technology in the computer vision area. His research interests are in software agents and multi-agent systems, monitoring and activity interpretation systems and coordination and communication protocols to assist in decision making process using agent technologies.

Elena Navarro

Elena Navarro is an associate professor of Computer Science at the University of Castilla–La Mancha (Spain). Prior to this position, she worked as a researcher at the Informatics Laboratory of the Agricultural University of Athens (Greece) collaborating in the CHOROCHRONOS project funded by the Training and Mobility of Researchers program of the European Union. Previously, she served as a staff member of the Regional Government of Murcia, at the Instituto Murciano de Investigación y Desarrollo Agrario y Alimentario, collaborating in the INTERREG II project funded by the European Union. During her master degree studies, she was a holder of several research scholarships funded by the Regional Government of Castilla–La Mancha and the National Government of Spain. She received her bachelor degree and PhD at the University of Castilla–La Mancha and her master degrees at the University of Murcia (Spain) and Rey Juan Carlos University (Spain). She is currently an active collaborator of the LoUISE group of the University of Castilla–La Mancha. Her current research interests are requirements engineering, software architectures, model-driven development and agent-oriented software development.

Antonio Fernández-Caballero

Antonio Fernández-Caballero received his master in Computer Science from the Technical University of Madrid, Spain, in 1993, and his PhD from the Department of Artificial Intelligence of the National University for Distance Education, Spain, in 2001. He is a full professor with the Department of Computer Science at the University of Castilla–La Mancha, Spain. He is the director of the n&aIS (natural and artificial Interaction Systems) research group at the Albacete Research Institute of Informatics. His research interests are in image processing, cognitive vision, neural networks and intelligent agents. Antonio Fernández-Caballero is an associate editor of the Pattern Recognition Letters journal. He has authored more than 200 peer-reviewed papers.

Rafael Martínez-Tomás

Rafael Martínez-Tomás received his degree in Physics from the University of Valencia, Spain, in 1983, and received his PhD from the Department of Artificial Intelligence of the National University for Distance Education, Spain, in 2000. Since 2001, he is an associate professor with the Department of Artificial Intelligence of the National University for Distance Education, Spain. His research interests are in knowledge engineering, knowledge-based systems, semantic web and semantic technologies and semantic recognition of human behaviour, publishing research papers related to these areas in various international journals and in major international conferences.