IDK and ICARO to develop multi-agent systems in support of Ambient Intelligence

José M. Gascueña, Elena Navarro, Patricia Fernández-Sotos, Antonio Fernández-Caballero, and Juan Pavón

Instituto de Investigación en Informática de Albacete (I3A), Albacete, Spain
Universidad de Castilla-La Mancha, Departamento de Sistemas Informáticos, Albacete, Spain
Universidad de Castilla-La Mancha, Facultad de Medicina, Albacete, Spain
Universidad Complutense de Madrid, Facultad de Informática, Departamento de Ingeniería del Software e Inteligencia Artificial, c/Profesor José García Santesmases, s/n, Madrid, Spain

Abstract. An important issue in Ambient Intelligence (AmI) is the massive deployment of intelligent agents embedded in the environment which are adaptive to the users' profiles, preferences and needs. This is why, the use of agent-oriented methodologies and frameworks is almost mandatory to easily develop software for AmI scenarios. ICARO is a software framework that promotes the use of different organizational and behavioral patterns to implement multi-agent systems (MAS). Its extensive use in several projects demonstrates a substantial increase in software productivity. Also, in order to reduce the coding effort it is usual to design MAS at a higher level. In this sense, code generation from MAS specifications into ICARO framework has been performed. INGENIAS Development Kit (IDK) supports both the specification of MAS models, including any feature required to implement MAS with ICARO, and a set of facilities for code generation. This paper describes the development of AmI applications thanks to the integration of ICARO and IDK. Two IDK modules have been developed, namely a "code generation" and a "code update support" module.

Keywords: Multi-agent systems, agent-oriented software engineering, code generation, ambient Intelligence, assisted living systems, ambient assisted systems, smart homes

1. Introduction

Ambient Intelligence (AmI) is being a very active topic of research in the last decade. Ducatel et al. [14] envisioned some of the ideal scenarios that AmI should offer to society, community as well as individuals. As stated by the authors, AmI promotes the development of innovative and intelligent user interfaces "embedded in an environment that is capable of recognizing and responding to the presence of different individuals in a seamless, unobtrusive and very invisible way". AmI user interfaces become transparent (people do not perceive complexity neither presence) and they are "intelligent" to react in a proactive and sensitive way [1] at the same time. The idea of making technology transparent but usable to people has opened the door to its exploitation in a wide variety of previously unforeseen ambiences. For instance, it is being widely exploited in the context of smart homes, also called ambient assisted systems and assisted living systems. These are home-like environments that exploit AmI to respond to the behavior of their residents and to provide them with novel services and facilities [13]. Home care is one major category of smart home application. Some systems have been developed for detecting falls of elders by processing the information of sensor infrastructures [37]. Other interesting systems are being developed to detect health care problems such as heart
As stated by Cook et al. [10] AmI technologies are expected to be sensitive, responsive, adaptive, transparent, ubiquitous, and intelligent. The three first features greatly rely on the context-aware computing field, but transparency and ubiquity depend on the area of ubiquitous computing, also called pervasive computing. But it is intelligence which becomes a critical feature as it makes AmI systems more sensitive, responsive, adaptive, transparent and ubiquitous. The main reason is that intelligence helps in understanding user environments and, consequently, in providing adaptive assistance [9]. This explains why AmI entails contributions from different AI areas, such as machine learning [3], ontologies [24], neural networks [9] and, specially, MAS. Indeed, MAS are specially good at modeling, simulating and representing meaningful entities such as rooms and persons in AmI systems [34]. The AmI system properties turn the development into a challenging work demanding adequate tools that assist throughout a well established software development process. Both tools and process should be able to adapt to the diversity of AmI devices and communication technologies from an implementation point of view. The major contribution of this paper is the description of our experience after developing a couple of software modules facing the mentioned issues. The software consists in a code generator module and module to support code update. The modules have not been developed from scratch but INGENIAS Development Kit (IDK) [20], the tool supporting the agent-oriented software engineering methodology named INGENIAS [31], has been used. IDK provides a template-based proprietary mechanism for developing new modules which automatically generate code for any target language. As far as we know this functionality is not provided by any other tool dedicated to designing multi-agent systems (MAS) or Prometheus Design Tool (PDT) [30], Tool for Agent Oriented visual Modeling for the Eclipse platform (ToADE) [26] or DSM4MAS development environment (DDE) [38].

The two aforementioned modules generate code for the ICARO framework that provides high-level software components to facilitate the development of agent-based applications. Moreover, it promotes the use of different organizational and behavioral patterns that enable the specification of MAS at a higher level during design phases. With this aim in mind, ICARO provides engineers with concepts and models, together with a customizable MAS design, Java code fully compatible with software engineering standards. Also, it can be integrated into the most popular integrated development environments (IDE). Moreover, it is independent of the agent architecture, this way enabling developers to create new architectures and to integrate them into the framework. This is a clear difference with regard to other agent frameworks, such as JADE [4] or ICARO [4], as they provide a middleware instead of an extensible architecture to establish the communications among agents. This advantage provides developers with the necessary flexibility to deal with the diversity of AmI devices and communication technologies. An additional advantage is that the framework already implements functionality for automatic component management and application initialization and shutdown. This relieves the developers’ workload and ensures that all components are under control. This last functionality is usually not provided under other frameworks.

The rest of the article is organized as follows. Section 2 describes the related work. Then, Section 3 introduces IDK modules developed for ICARO. After that, an AmI oriented case study on personal health monitoring illustrates our approach in Section 4. Finally, some conclusions are provided in Section 5.

2. Related work

As stated by Cook et al. [10] AmI technologies are expected to be sensitive, responsive, adaptive, transparent, ubiquitous, and intelligent. The three first features greatly rely on the context-aware computing field, but transparency and ubiquity depend on the area of ubiquitous computing, also called pervasive computing. But it is intelligence which becomes a critical feature as it makes AmI systems more sensitive, responsive, adaptive, transparent and ubiquitous. The main reason is that intelligence helps in understanding user environments and, consequently, in providing adaptive assistance [9]. This explains why AmI entails contributions from different AI areas, such as machine learning [3], ontologies [24], neural networks [9] and, specially, MAS. Indeed, MAS are specially good at modeling real-world and social systems, where problems are solved in a concurrent and cooperative way without the need of...
reaching optimal solutions [34]. This is why the natural relationship between AmI and MAS is being widely exploited. Indeed, AmI proposes the development of context aware systems equipped with devices that recognize context and act accordingly. Agents provide an effective way to develop such systems since agents are reactive, proactive and exhibit an intelligent and autonomous behavior [2]. Agents react to humans based on information obtained by sensors and their knowledge about human behaviors within agent-based AmI applications [6].

As aforementioned, the development of AmI systems is not a trivial issue, being necessary the use of tools and well-defined processes that guide the stakeholders. Taken into account the relevance that MAS has for AmI, several alternatives providing the necessary support have recently been developed. One of them is the methodology defined by Serrano et al. [36], based on the use of MAS-oriented simulations for the validation of AmI-based ubiquitous computing systems. This proposal is very interesting as it helps to validate AmI applications whose real tests would be impractical because of the unavailability of resources, high cost, and so on. Unfortunately, the methodology only provides guidelines for validation purposes and not for the whole development of AmI systems; so it has to be used jointly with other proposals. Muñoz et al. [28] present an argumentative MAS architecture that exploits semantic web ontologies to discover inconsistent contexts, but the approach does not offer a complete support for the development of AmI systems from an implementation point of view. Yi-bin et al. [41] have also defined a set of models applicable for design and development that unfortunately cannot be extended to provide the necessary flexibility. Ayala et al. [2] have recently developed a platform-neutral framework called MalacaTiny. It is a quite powerful framework which manages the diversity of communication technologies by using aspect-oriented programming. However, it has been specifically designed for mobile applications so that its capabilities to develop general AmI systems remain unexplored. Another framework is JaCaMo [7] which encompasses a multi-dimensional concept (organization, agent and environment) to develop MAS. Unfortunately, the JaCaMo related agent programming language uses belief-desire-intention (BDI) [35] concepts, reducing its reusability in different applications. Although these proposals are relevant to the development of AmI systems and provide great advantages, all of them experience some serious problems when dealing with two challenging issues at the implementation level, namely the diversity of AmI devices and the variety of communication technologies.

A clear alternative is ICARO which provides patterns to build reactive [18] and cognitive [22] agents. The use of component patterns for modeling MAS is a differentiating factor from other agent-oriented programming frameworks and languages [5]. Moreover, while other agent-based platforms that are FIPA-compliant (e.g. JADE) mainly focus on communication standards, ICARO targets on provisioning high-level software components for easy development of complex agent behavior, agent coordination and MAS organization. Regrettably, to date there are no tools for modeling MAS applications and generating code for this framework. So, there is a need for code generation from design specifications to ICARO framework.

Code generators are useful tools for software development due to the evident benefits provided by their exploitation [15, 29]. One of these benefits is an improved productivity since the time necessary to perform coding tasks is reduced. Another important benefit is that the quality of the developed systems is also improved, as the generated code (usually) does not contain bugs. MAS are not an exception to this rule. Several tools for developing MAS applications [33] already provide functionality to generate code for a given agent programming language or framework. For example, the supporting tool of the Prometheus methodology [30] offers a code generation facility to automatically produce JAdE agent language code [40]. Tagom4E [27], a tool for the development of software following the TROPOS methodology [26], includes functionality to generate code for Jadex language [8]. DDE [38] is an environment for the development of MAS that is based on a Domain Specific Modeling Language for MAS and supports code generation for JACK and JADE languages [4]. Another proposal develops a code generator with MOFScript (see http://marketplace.eclipse.org/content/mofscript model-transformation-tool) that transforms agent models, compliant to its meta-model PIM4Agents, to code for the MalacaTiny platform-neutral framework [2]. Finally, our code generator was developed with IDK [20]. Several reasons led us to this decision. First, it offers a graphical editor for modeling MAS applications and functionality for developing new modules able to automatically generate code for any target language. This graphical editor, generated from the INGENIAS meta-model [31], can be extended by introducing new concepts and relations needed to build new MAS applications. Afterwards, the IDK is again regenerated from
the new meta-model specification. An advantage of
this approach is that changes in the definition of the
meta-model are easily applied to generate personalized
editors, facilitating the needed flexibility for handling
the inherent diversity of AmI systems. Second, the
development of the modules follow a general process
based on both the definition of specific templates for
each target platform and procedures to retrieve infor-
mation from INGENIAS models [19]. Currently, IDK
incorporates modules to generate code for JADE lan-
guage from design artifacts as well as documentation in
HTML format. Finally, a module named code uploader,
which is used to keep the code components design arti-
facts updated with changes made in the implementation,
is also available.

3. IDK Modules for ICARO

The development of IDK modules has followed a
"bottom-up" approach to support ICARO as the target
platform chosen for the final implementation of a MAS
application. In first place, the INGENIAS structures for
specifying all concepts and their relations which are
necessary to implement an application in ICARO are
identified. Then, a module which automates the task
of ICARO code generation from INGENIAS specifica-
tions, in line with the identified conceptual relations, is
gradually developed. Finally, a new module upgrade
the specification of a model when there are changes in
the implementation. A detailed description of the gen-
eral process for developing IDK modules can be found
in [31]. The next subsections provide a description
about the relations between INGENIAS and ICARO
concepts, as well as the development of the modules to
generate code for ICARO and to support the update of
code, respectively.

3.1. Conceptual relation between INGENIAS and
ICARO

First, it is worth explaining some details of the fig-
ures that describe the relationship between INGENIAS
and ICARO. The right side of the figures correspond to
the notation chosen to express a fragment of a model
using ICARO concepts and the left side is the notation
used to express the same fragment but in terms of the
INGENIAS language.

Any communication between the components imple-
mented to develop a new executable ICARO application
can be 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, agents
send events through their use interfaces and, in the same
way, a resource also employs the use interface of an
agent to send it an event. Second, an agent utilizes
the resource use interface to request the offered ser-
dices (methods). From our point of view, the concepts
of reactive application agent and application resource
used in ICARO can be modeled in INGENIAS by using
the concepts of agent and application, respectively. For
example, when establishing an ApplicationBelongsTo
relationship between an agent and an application, it
is understood that the agent uses the services offered
by the resource (see Fig. 2). In particular, the actions
that agents execute on the environment are represented
by this structure. Services are modeled as application
methods.

Sending information from a resource to an agent is
modeled in INGENIAS by establishing an EPerceives
relationship between the agent and the application
which represents the resource (see Fig. 3). In INGE-
NIAS, this information falls within the EPerceives
relationship that is modeled with an event of type Appli-
cationEvent when a resource simply sends a signal to
the agent. But, it is modeled with an event of type Appli-
cationEventSlots when more information has to be sent.
In the latter case, the information and its type is modeled
with slots entities.

Now, sending information among reactive agents is
modeled in INGENIAS by specifying an entity of
type InteractionUnit and relating it to the producer
and consumer agents by means of the UInitiates and
UCollaborates relationships, respectively (see Fig. 4).
If the producer sends information, then it is included in
the interaction unit through an entity of type Frame-
Fact containing the necessary slots to transport it.
Conversely, if it only needs to send a signal, then the FrameFact is not included in the interaction unit. Visually, it is possible to know that an interaction unit includes a FrameFact because it shows the “Info” attribute (the value shown is the identifier of the FrameFact).

Whenever an ICARO user wants to implement the behavior of a reactive agent, he/she has to create an automaton modeled with a state diagram. In particular, five structures available in INGENIAS ‘state diagram’ and a criterion to name agents are needed to specify any automaton (see Fig. 5):

- A relationship is established between an “InitialNode” entity and the state to represent the initial state.
- A relationship is established between an “EndDate” entity and the state to represent a final state.
- A “WFollowGuarded” relationship is established between two different states to represent a transition; and a transition is specified using the syntax event / semantic action in its “Condition” attribute. The event represented in the state diagram is related to an ApplicationEvent or an ApplicationEventSlots entity when the event is sent by a resource (see Fig. 3). It is related to an InteractionUnit entity if the event is sent by an agent (see Fig. 4). The semantic action takes the same name as the task created in the components diagram.
The IDK tool does not enable to explicitly represent relationships that cycle over the same entity, so that a fourth structure has been considered. In order to represent a transition that comes back to the same state, first a copy of the state is made, afterwards a “WFollowGuarded” relation is established from the copied state to the original state, and finally the transition is specified following the syntax described in the previous third structure.

Universal transitions of an automaton of an ICARO reactive agent are valid for any state of an automaton. That is to say, when the event arrives, actions are executed and the next state is reached, regardless of the automaton current state. The solution for graphically representing them in INGENIAS is the “UniversalState” that represents any state and takes the role of the “source” state of the universal transition. Obviously, “UniversalState” has not to be used with a different meaning.

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

The XML file that describes the organization of an ICARO application represents its deployment. The IDK tool offers the possibility of creating deployment diagrams. The number of instances of each type of agent is specified using entities such as DeploymentPackage and/or DeploymentUnitByType. However, the existence of an application instance is conditioned by the existence of the instance of an agent. For this reason, it is necessary to find an alternative way of independently expressing the number of instances of agents and applications, such as in ICARO. The solution is to create an environment model and to follow the following steps: (1) to copy all the agents and applications, (2) to relate them with entities of type UMLComment, and (3) to set the number of instances to be deployed in the attribute Text of UMLComment. Obviously, this process can be repeated over and over to create different deployment configurations.

The IDK module named INGENIAS ICARO Framework generator) IIF has been developed to generate code for the ICARO framework. For this aim, the IIFGenerator class is extended so that its constructors possesses the templates that the IIF module uses in a similar way to any other IDK code generator. Moreover, 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 artifact that an ICARO user has to implement (see Table 1).

The IDK templates for code generation contain source code written in the programming language of the target platform and tags to establish where the model information is used during the code generation. The...
kind of tags in an IDK template is limited [19]. program is the main tag of the document, repeat means that the text enclosed by this tag has to be copied and pasted in order to be duplicated, v represents a variable, and save 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 [39] or XPAND [21]. However, IDK exhibits a disadvantage as it does not enable developers to reuse templates. They have to copy and paste the fragments to be reused, this way 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 with such intermediate state.

When the IIF module is executed using a model created with IDK, a sequence of data is generated. For instance, a sketch of the sequence of an agent automaton is shown next.
4. Case study: AmI personal health monitoring system

A case study consisting in an AmI scenario for personal health monitoring is introduced to show the effectiveness of our approach. Personal health monitoring refers to any electronic device or system that monitors a health-related aspect of a person’s life on a constant basis outside a hospital setting [25]. Some details related to the monitoring scenario are described next. There are several electronic devices used to monitor the vital signs of a patient (see Fig. 6):

• A heart rate monitor (HRM) enables to measure a person’s heart rate in real time.
• A pulse oximeter (PO) enables the monitoring of the saturation of a patient’s hemoglobin.
• A continuous respiratory monitor (CRM) is used for monitoring of respiratory problems.
• Ambulatory blood pressure monitoring (ABPM) is carried out to measure blood pressure at regular intervals.
• A continuous glucose monitoring system (CGM) measures glucose levels.
• A body temperature sensor measures human body temperature.

This way, each time the IIF module is run, each protected region is overridden with the code manually written by the developer.

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

The patient is described through the following parameters: man; 60 years old; 172 cm tall; 90 kg weight; smoker. The patient is constantly monitored by the previously mentioned devices. Let us assume that at a given moment the monitors show the following values about his/her vital signs:

• The heart rate ranges from 80 to 90 heartbeats per minute.
• The level of oxygen saturation (SpO₂) measured by the pulse oximeter is equal to 91%, which reveals that the human presents a mild desaturation.
• The respiratory rate is 24 breaths per minute.
• The clinical systolic and diastolic blood pressures are equal to 150 and 100 mm Hg, respectively. This suggests hypertension.
• The concentration of glucose in blood is 70 mg/dl.
• The body temperature is 37.6 degrees (Celsius).

Although the heart rate is normal, there are some alterations in other vital signs that may infer an acute myocardial infarction (AMI). Moreover, it is known that two family members died of AMI. Therefore, a mobile intensive care unit (MICU) is immediately called to take care of him. It is supposed that there are ten MICUs available in the health care system.

The scenario describes an example of execution of an AmI personal health monitoring system. Once each type of sensor needed for monitoring the patients has been identified, it can be easily modeled as a software agent [32]. For instance, Fig. 6 shows that agent “TAg” in charge of manipulating the “Body Temperature Sensor” is modeled. The scenario introduces two additional requirements, autonomy and communication, for which MAS are especially appropriate [23]. The patients live at their homes, usually suffering mobility problems, so that the AmI personal health monitoring system has to communicate with them to monitor their state and to autonomously reason whether an anomalous situation happens in order to act properly.

The AmI personal health monitoring system has to deal with the described scenario. For this, firstly the entities responsible for coordination are modeled as agents, whereas entities responsible for managing persistent information are modeled as applications (see Fig. 7). Moreover, two applications are identified to denote the graphical user interfaces that interact with humans and medical professionals. Then, the
communications among agents are modeled. In INGE-
NIAIS these communications are specified as interaction
units that contain a framefact which stores the trans-
ported information in the communication (see Fig. 8).
So, similar structures to those depicted in Fig. 4 are
specified with IDK to represent the communications
among agents.
Next, the different scenarios have to be specified
for the personal AMI personal health monitoring sys-
tem. For instance, the interaction protocol illustrated in
Fig. 9 is specified to deal with the scenario described
above integrating the information collected by the
different devices. These agents communicate with a
personal health agent which is responsible for integrat-
ing the information obtained from the devices and to
evaluate if the symptoms are related to some health
problem. Moreover, the personal health agent queries
a user database to request additional information about
the monitored human (e.g. age, gender, previous dis-
cases and family health history). A UserDB application
entity models the management of this database, that is,
to get, delete and update user persistent information.
The personal health agent behavior concludes that an
AMI might be happening and communicates this fact
to a medical resource manager agent. The last agent
selects the best MICU according to the geographical
location of the human.
As aforementioned, AMI systems have to monitor,
reason and react. Monitoring of the health system as
well as its reactions can be specified through INGE-
NIAIS. Moreover, INGENIAS can also be used to model
the behavior of the reactive agents through automata
(state diagrams). For example, the personal health agent
reacts to the information related to the devices as sent by
other agents. Moreover, it is responsible for reasoning
if any health problem is happening through the human
health state by using different rules, such as:

If heart rate > 100 Then Tachycardia;
If heart rate < 60 Then Bradycardia;
If breaths per minute > 20 Then Tachypnea;
If breaths per minute < 12 Then Bradypnea;
...

So, a particular health problem is detected when an
individual rule is satisfied. However, the interest of its
function increases when several conditions are simul-
taneously satisfied, because the integration of different
information sources allows the diagnosis of more com-
plex problems (as an AMI in the described scenario).
The personal health agent holds an initial state, a final
state and remains in an intermediate state to receive
information sent by the agents related to the devices
(see Fig. 10). Each certain time a diagnosis is issued by
the agent.
It is also necessary to specify another model for
the deployment of the application (see Fig. 11). For
instance, this model would specify that there is one
instance for each type of agent related to a device, one
instance of personal health agent, one instance of med-
ical resource manager agent, ten instances of MICU
agents and MicuGUI applications, and one instance
for applications of type UserDB and UserGUI for the
described scenario. It is worth noting that the above
described models are just a partial view of the system,
as they would have to be extended to consider all its
requirements. For instance, this is the case in order to
create new patients or to define additional reasoning.
Finally, once the different INGENIATAS models are specified, the following process is carried out using the developed modules in order to obtain the final application (see Fig. 12):

1. The developer uses the INGENIATAS ICARO Framework generator module (IIF) to automatically generate code from the design specified with the IDK tool (see Fig. 12, step 1). IIF generates several...
Fig. 11. Deployment model.

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. After that, the developer manually inserts code into the protected regions of the generated java classes and implements the new necessary classes (see Fig. 12, step 2). Specifically, the developer only needs to develop the graphical user interfaces, knowing how an agent accesses the resources and how an agent and/or a resource send an event to an agent to complete the implementation of the application.

Let us clarify that resource code is generated from application entities specified in the detailed design model using the INGENIAS modeling language.

3. The developer uses the ICAROTCodeUploader module to update the model with the modifications introduced in the protected regions (see Fig. 12, step 3). It enables the developers to always keep synchronized the model and the source code, so that changes introduced in the source code are kept when the code is again regenerated from the model.

Finally, the script file generated by the IIF module is executed by the deployment manager to launch the developed application.

5. Conclusions

This paper has shown the easiness to implement AmI applications using the IDK tools to design them as MAS. The paper has also described the two developed modules to generate code for the ICARO framework from the design. More specifically, the development of these two modules (code generator and code update) that provide support for the implementation of ICARO reactive agent applications has been described. These modules are integrated into the IDK tool. It is worth pointing out that the time spent for learning how to develop and implement the IIF and ICAROTCodeUploader modules described in section 3 was two months and fifteen days. This effort shows an improvement in modeling and implementing new applications in terms of productivity. The main reason is that the time necessary for coding is reduced as developers do not need to learn the structure, location and naming rules of ICARO applications files.

We would like to point out that the presented modules have been validated through their use in the development of two different applications. The first application [16] was developed to face the problem of a collection of robots patrolling around a surveillance environment.
The second application [17] was developed for monitoring and controlling the normal and anomalous situations that happen when humans access to a specific area. During the development of both applications, the developer only had to write manually the body of both the resource methods and semantic actions along with some auxiliary classes. The remaining code was automatically generated by using as input the models created with the IDK.

Finally, two challenges constitute our on-going work. The first one is related to the validation of the proposal. The developed modules have shown their usefulness and suitability when used in the two aforementioned projects. They allow to reduce the time necessary to perform coding tasks and to increase the quality of the developed systems since the generated code contains no errors. Although these results are very promising, we are currently designing an experimental evaluation with developers to determine the acceptance of this proposal in a close future. The second challenge is mainly related to the extension of the IDK modules to provide additional support. Currently, we are extending the IF module to generate the event notification code.

Acknowledgements

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

References


