# Towards Ontology-Driven Development of Applications for Smart Environments

Artem Katasonov and Marko Palviainen
*VTT Technical Research Centre of Finland*
*Tampere/Espoo, Finland*
*Email: artem.katasonov@vtt.fi, marko.palviainen@vtt.fi*

***Abstract*—In the environments where heterogeneous devices need to share information, utilize services of each other, and participate as components in various smart applications, it is common to rely on the advantages of the semantic data model and ontologies. Our work extends this approach so that also the process of software development for such environments is ontology-driven. The goals are to raise the level of abstraction of smart application development, to enable development by non-programmers, and to partially automate the development to make it easier and faster. In this paper, we describe the Smart Modeller that consists of 1) a design tool that enables the developer to graphically create a model of a smart space application and 2) a framework that provides core interfaces for extensions supporting both the model and ontology-driven development. These extensions enable: ontology-based creation of model elements, discovery and reuse of both the software components and partial models through a repository mechanism, and generation of executable programming code for models.**

***Keywords*-Smart environment; Interoperability; Ontology-driven software engineering; Semantic technologies**

## I. INTRODUCTION

The work reported in this paper is a part of the project SOFIA [1] which aims at enabling development of devices and applications that can interact across vendor and industry domain boundaries. Consider, for example, the environment of a car. The car commonly has a board computer, there may also be an entertainment system in it, and there will normally be one or more mobile smartphones that the driver and the passengers have brought in. All these devices will possess some pieces of information about the physical world, e.g. location and speed of the car, current activities or context of passengers, and so on. While there are many applications imaginable on the intersection of these datasets, the sharing of information between devices is not easy at present. SOFIA's mission is to overcome the barriers of heterogeneity and lack of interoperability and to enable devices to share information, to utilize services of each other, and to participate as components in various smart applications. Additionally, a solution is targeted in which applications are distributed with some components residing on the devices while some other components residing in the network. In so, SOFIA pursues the goal of making information in the physical world universally available to various smart services and applications, regardless of their location, which aligns well with the vision of the Web of Things. As concrete results, SOFIA targets the development of an Open Innovation Platform (OIP) architecture and an Application Development Kit (ADK) toolset. This paper belongs to the latter effort.

OIP architecture [1] approaches the interoperability issue through separation of a system into three layers. The lowest layer is the Device World, which has devices connected through networks and gateways. The middle layer is the Service World, which has applications, services, and other software-based entities. The highest layer, unifying the lower ones, is the Smart World, which is an information-level world. In OIP, as much as possible interaction between devices is assumed to happen through information sharing rather than low-level invocations. OIP applies the *blackboard* architecture to provide a cross-domain search extent. This means that any smart environment (e.g. that of a car) is assumed to have at least one element called *Semantic Information Broker (SIB)*. Physically, the SIB can be located in the physical environment considered or anywhere on the network. Also, the access to SIB is not restricted to the devices comprising the physical environment. The information in a SIB can be made open to some application and components on the network. IOP defines a SIB as an information-level entity for storing, sharing and governing the information. As word "semantic" in SIB indicates, IOP relies on the advantages of the semantic data model, i.e. Resource Description Framework (RDF), and the ontological approach to knowledge engineering. A SIB is basically a lightweight RDF database that supports a defined interaction protocol that includes both the query and the subscribe functions. The other central element of OIP is *Knowledge Processor (KP)*, which is an information-level entity that produces and/or consumes information in a SIB. KPs form the information-level behaviour of smart applications. In addition to KPs, devices can have software components providing *services* to users and other devices.

The objective of the SOFIA's Application Development Kit (ADK) is to support and facilitate application development for OIP. ADK is supposed to provide and integrate cross-domain tools needed for all the phases of the application development. Thanks to the separation of the information-level and the service-level in IOP, and a higher position of the former, the ontologies defined for the information-level of the run-time operation of a system have

978-1-4244-5328-3/10/$26.00 ©2010 IEEE

an obvious value also for the development phase of that system. The ADK, therefore, follows the *Ontology Driven Software Engineering (ODSE)* approach (see e.g. [2]). Ontologies are used as inputs for developing the software architecture, used in the specification of the system and for discovery of appropriate software components. An additional value of the ontology-driven approach in ADK is that it enables people without an extensive software engineering background to effectively participate in the development or modification of smart applications.

This paper describes the *Smart Modeller* which is a central tool in the ADK toolset. It enables the developer to graphically create a model of a smart space application and automatically generate executable programming code for it. Various ontology-driven extensions to the Modeller enable further automation of the process. Examples of tasks such extensions perform are generating model elements based on domain ontologies, importing sub-models from repositories for re-use, and ontology-driven discovery of appropriate software components to be integrated into the application.

The rest of the paper is structured as follows. Section II discusses ODSE and provides the motivation behind the Smart Modeller. Section III presents the meta-model of the Modeller and available extensions to the Modeller. Section IV provides some details of the Eclipse-based implementation of the Modeller, and, finally, Section V concludes the paper and gives directions for future work.

## II. Ontology-Driven Software Engineering

*Ontology Driven Software Engineering (ODSE)* can be seen as an extension to *Model-Driven Engineering (MDE)* [3], [4]. The MDE approach is based on increasing level of abstraction to cope with complexity. In MDE, models are used not only for design and maintenance purposes, but as a basis for generating executable artefacts for downstream use. MDE insulates business applications from technology evolution through increased platform independence, portability and cross-platform interoperability, encouraging developers to focus their efforts on domain specificity [5].

A common MDE process (e.g. [4]) involves the following steps. First, a *Computation-Independent Model (CIM)* is created, which could be e.g. a UML's use case diagram. Then, based on CIM, the *Platform-Independent Model (PIM)* is developed, which can include UML's class diagram, state-transition diagrams, and so on. In the next step, PIM combined with a *Platform Profile* is transformed (at least some automation is assumed) into the *Platform-Specific Model (PSM)*, which is tailored to and provides more information for a particular operating system, particular programming language, and so on. Finally, the executable programming code is generated from PSM.

The main issue with the MDE process is that its two first steps, CIM and PIM, are fully manual and the connection between them is rather loose. In response to that, ODSE was introduced, which includes the use of ontologies in the MDE process. The most common ontology use in ODSE is having a domain ontology in the place of CIM and trying to utilize it for generating some parts of PIM [6], resulting in some level of automation also in this step. In particular, many works e.g. [7] assume transforming the domain ontology directly into the software application's hierarchy of classes. The ontology can also be used for automated consistency checking of PIM or PSM [5].

In our work, however, we aim at a much wider range of ontologies utilization. Based on the review of ontology classifications given in [2] and putting it in the context of ODSE, we consider that it is important to divide the ontologies into the following three groups:

- *Domain ontologies* - define concepts of the application domain, e.g. "car", "user", etc.
- *Task ontologies* – define the computation-independent problem-solving tasks that exist in the domain, e.g. "adjust", "sell", etc.
- *Ontologies of software* – define concepts used to describe the software world itself, e.g. "component", "method", etc.

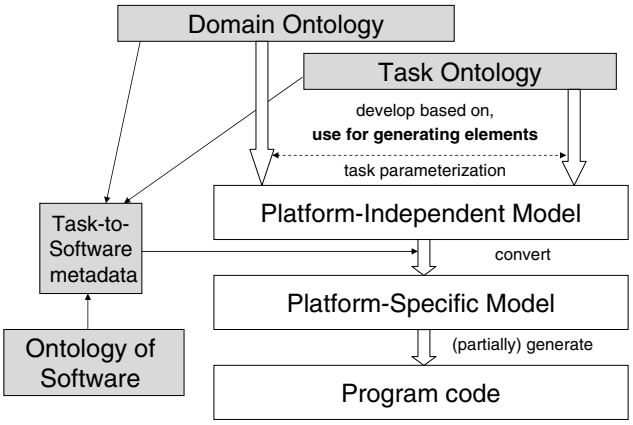


Figure 1. Used ODSE process.

While the domain ontologies are the most common kind of ontologies considered, the importance of defining also task ontologies is argued in [8]. A generic task defined in a task ontology can be imported into the PIM and linked to some domain ontology concepts as task parameters, e.g. creating "adjust car speed" specific task. Then, by matching the task description with the annotations of software components (for a certain platform) residing in a repository, a proper component to implement the given task can be found and included into PSM. Such component annotations (sometimes called *software metadata*) have to refer to concepts from all three ontologies (software, task, and domain for e.g. method parameters description). It is worth mentioning that semantic annotations of software components can be used, and are

used, outside ODSE, for example in so-called process of mashing-up composite applications [9].

Figure 1 depicts the overall ODSE process as we use in our work. We can also list the four values of ontologies steaming from such a process:

- *Specification* - Ontology is used as a part of the system requirements specification or/and design specification – in the place of a CIM in MDE.
- *Search* - Appropriate software components are discovered based on semantic descriptions of them.
- *Verification* - Ontology can be a basis for verification (e.g. consistency checking) of system requirements or/and design.
- *Communication* - Shared ontology acting as a "boundary object", i.e. something that is understood by people coming from different social worlds.

As was stressed before, the level of domain orientation and automation brought by such uses of ontologies also enables people without an extensive software engineering background to more effectively participate in the development or modification of smart applications.

## III. Smart Modeller

This section introduces the *Smart Modeller*, which is a central tool in the ontology-driven ADK developed in the project SOFIA. The Modeller enables the developer to graphically create a model of a smart space application and then automatically generate executable programming code for it. Various extensions to the Modeller enable further automation of the process, contributing to the ease and speed of development. The Modeller is based on a visual composition approach that (1) raises the level of abstraction of smart application development, and (2) supports reuse of both the existing software components and partial models. The reuse of software components and models is supported by *repositories*, which are RDF data storages themselves.

### A. A Meta-Model for Smart Objects

The Smart Modeller models an application for an OIP-based smart environment as a directed graph. This means that the used meta-model (similarly to RDF) contains two kinds of entities: *elements* (nodes) and *connectors*. All the types of elements have an attribute "name", and for some of them it is the only attribute defined – because a bigger part of information in the model is contained in the connectors between elements. Some elements have, however, some additional attributes defined.

The following model elements are currently provided:

- *Smart Object* – device able to interact with a smart environment and participate in a smart application. It may provide Services and may contain Knowledge Processors and SIBs.
- *Semantic Information Broker* – an entity for storing, sharing and governing the information as RDF triples (see Section I). Additional attributes: IP, Port, and Smart Space Name.
- *Knowledge Processor* – an entity that produces and/or consumes information in a SIB (see Section I). KPs form information-level behaviour of smart applications of modelling this behaviour is the principal goal the Smart Modeller.
- *Ontology* – a local or online document containing RDF Schema (RDF-S) and possibly Web Ontology Language (OWL) definitions. Additional attribute: Url.
- *Action* – a specific action (e.g. Java method) or a generic action (a task) that a KP performs. In case of a specific action, additional attributes are: Implementation (e.g. System.out.println) and optionally Listener Interface (e.g. IAlarmListener).
- *Condition* - an element denoting a part of the state of a KP. A condition refers to e.g. the start-up of the KP or to a state immediately following successful execution of an Action. The main use of a Condition is to define when an Action is triggered.
- *Parameter* – an input or output parameter of an Action, or a constituent of a Condition. Additional attributes: Type, Position (to define the order in a method signature), Value (if a constant), and Semantics (to link to a domain ontology concept, see Section II.
- *Service* - some kind of service provided. The service provider and the service consumer can be different smart objects, different KPs of one smart object, or even parts of one KP. It is only a logical entity because there always has to be an Action that "provides" the Service and another Action that "uses" the Service.
- *Repository* – an RDF storage (following this meta-model) containing pre-configured model elements and their groups that can be re-used in modelling. There exist extensions for both importing and exporting elements from/to a Repository. Additional attribute: Url.
- *Composite* – a sub-model that contains other elements and connectors between them. Composite can be expanded to show its contents or collapsed and shown as just an icon. Composites aim at increasing usability of diagram editing and at enabling modelling at even higher levels of abstraction. Additional attribute: Icon.
- *CompositePort* – an input/output port of a Composite that represents an element inside the Composite.

As one can notice, the meta-model of the Modeller includes elements of two kinds: those coming from the OIP architecture (Smart Object, SIB, KP, Service, and Ontology) and those belonging to an ontology of software (Action, Parameter, and Condition). An additional element Repository does not correspond to a part of a smart space application – it is a facilitator of the development process only.

A Connector is a link between two model elements. It has one attribute: Relationship. The graphical presentation

of a connector in a diagram is dependent on the value of this attribute. Below, we describe the main meaningful links between model elements:

- *model:has* from a Smart Object to a KP or a SIB – defines a KP or a SIB that the Smart Object hosts.
- *model:uses* from a KP to a SIB – indicates the SIB, from/to which the KP consumes/produces information.
- *model:has* from a KP to a Condition – defines a start-up Condition of the KP. It is used by the Code Generator as a starting point in KP behaviour creation.
- *model:uses* from a SIB to an Ontology – indicates the ontology, on which the data in the SIB is based. It is not used by the Code Generator, but is used for automated generation of SIB subscriptions from an ontology (see next subsection).
- *model:has* from a Condition to a Parameter – defines a constituent of a control-flow Condition, i.e. some data item that was *model:produce* by an Action led to this Condition.
- *model:triggers* from a Condition to an Action – defines an Action triggered when the Condition becomes true.
- *model:has* from an Action to a Parameter – defines an input parameter for the Action.
- *model:produces* from an Action to a Parameter – defines an output parameter or return value for the Action. For the return value, the name of the Parameter has to be *model:return*.
- Connector from an Action to a Condition – indicates the Condition that becomes true as result of the Action execution. Currently, the link has to be either *model:success* or a name of the method in the listener interface that the Action uses for posting events.
- *model:maps* from a Parameter to a Parameter – defines the mapping/wiring of parameters. The link should go from an output parameter/return value of an Action to an input parameter of another Action.
- Connector from an Action or a Smart Object to a Service – indicative link *model:provides* or *model:uses*.
- *model:refers* from an Ontology to an Ontology – denotes that a more specific Ontology refers to concepts from a more general upper ontology.
- *model:describes* from an Ontology to a Repository – specifies a Metadata set that defines links between some generic tasks (defined in another Ontology, to which Metadata *model:refer*) and specific Actions residing in the Repository. It is used by the extension responsible for search of appropriate implementations for generic tasks (see Section II).

### B. Modeller Extensions

The Smart Modeller currently has the following extensions defined:

*1) Generate Java implementation for the Knowledge Processor:* This extension is applicable to a Knowledge Processor element only. When executed, it generates Java implementation of the modelled KP. The implementation is created as a new Java project with name corresponding to KP's name in the model. The extension also copies into the generated project Actions' implementations, copies or links required libraries, and also copies any additional non-Java resources other than the diagram files found in the modelling project, e.g. images, RDF documents. In the future, we plan having code generators for other programming languages as well, including Python, C++, ANSI C for embedded devices, and Javascript for KPs providing web interfaces.

*2) Generate a SIB subscription from the Ontology:* This extension is applicable to an Ontology element only. When executed, it first shows a dialog window for selecting a type of subscription to be generated: (1) any new triple, (2) new instance of a class, (3) changed value of a property. If (2) or (3) is selected, the extension shows another dialog, where it lists all the classes and properties, correspondingly, that are defined in the ontology. Finally, the extension adds to the model a Composite that includes all needed elements for managing a SIB subscription and a minimum set of CompositePorts: for starting subscription, for receiving notifications, and for received data values.

*3) Import model elements from the Repository:* This extension is applicable to a Repository element only. When executed, it shows a dialog listing all graphs (collections or elements and connectors) defined in the repository and, then, inserts the selected graph into the model.

*4) Export model elements into a Repository:* This extension is applicable to any model element, or a collection of elements including connectors. When executed, it shows a dialog listing all repositories in the model, and then exports the selected elements and connectors into the selected Repository as a single graph.

*5) Initialize Java implementation for the Action:* This extension is applicable to an Action element only. This extension will generate a template for an action implementation, if a non-existing implementation is defined for an Action in the model. The extension will split the implementation attribute into package name, class name, and method name and, then, generate needed package folders, generate .java file for the class, and add a needed method. If the Action is linked in the model to input or/and output Parameters (including a return value), this information will be incorporated into the generated method interface.

*6) Generate a generic Task from the Ontology:* This extension is applicable to an Ontology element only. When executed, it shows a dialog window listing all the generic tasks defined in the task ontology, and when one is selected, generates an Action element with unspecified implementation and name corresponding to the selected task. This element is intermediary (PIM level) – it is assumed that the next extension will be used on it.

*7) Find an implementation from a Repository:* This extension is applicable to an Action element (generic task) only. When executed, it shows a dialog window listing all implementations (graphs) matching the given generic task, and after one is selected, adds it to the model. This extension searches through all the Ontology-Repository connected pairs in the model. The URL of Ontology must define the name of the metadata file (tasks to actions matching).

## IV. Eclipse-based Implementation

When selecting the implementation platform for the Smart Modeller, we followed the following design principles:

- *Openness*. The software is to be based on open-source components and will be published as open-source.
- *Interoperability with other tools*. To support interoperability, standard-based solutions are preferred.
- *Tool-level extensibility*. The software has to consist of a common framework and a set of extensions, so that new extensions can be easily introduced when needed.

Eclipse (http://www.eclipse.org/) was selected as the platform, on top of which the Smart Modeller is developed. Eclipse is an open, extensible, and well-standardized development environment providing an extensible application framework upon which software can be built [10]. An Eclipse-based implementation enables interoperability with many other existing tools as well as ready-made extensibility framework thanks to Eclipse component-based architecture.

The Smart Modeller is implemented in Java utilizing the Eclipse Graphical Modelling Framework (GMF) [11]. GMF provides a generative component and runtime infrastructure for developing graphical editors. GMF is build on the top of Eclipse Modelling Framework (EMF) and Graphical Editing Framework (GEF). First, the meta-model described in Section III-A was encoded using EMF Ecore. Then, a set of additional GMF-specific definitions was added, including the palette of creation tools, labels to use in the diagram, and so on. Then, the GMF framework generated the readily-working code for the diagram editor.

Thanks to the Eclipse component-based architecture, the generated diagram code was easily extended with a set of *extension points*, through which *plug-ins* can be connected. The *tool extension point*, was added to enable the interface towards custom plug-ins that can programmatically modify the diagram or utilize the information encoded in it for some purpose. All the Modeller extensions described in Section III-B were then implemented and connected using this extension point. New extensions can always be added later, also by 3rd parties.

Figure 2 shows a simple example model, as it appears in the Eclipse-based Smart Modeller. The application domain is an environment that has a set of sensors providing measurements (e.g. temperature) and a set of actuators, through which some parameters of the environment can be controlled (e.g. lighting). Actuators have status, e.g. ”on” and ”off”,

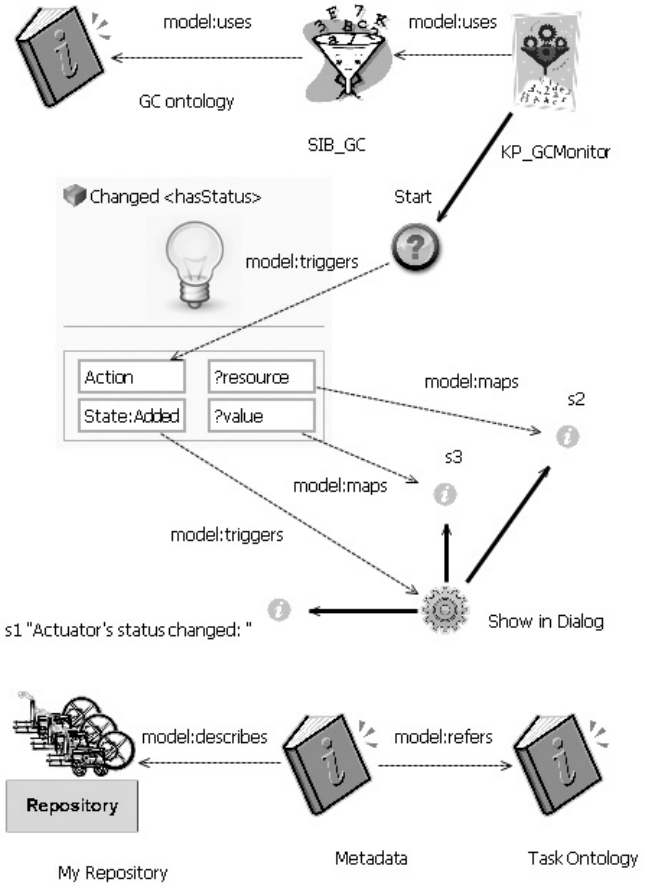


Figure 2. A model example.

and in the environment's SIB this status is reflected using *gc:hasStatus* property. This means that the SIB, at any given time, has a set of RDF statements in it, one per actuator, of the type *ex:ActuatorA gc:hasStatus ”off”*. The model in Figure 2 includes the SIB with the associated Ontology and one simple reactive Knowledge Processor. That KP has one subscription to the SIB with the pattern *?resource gc:hasStatus ?value* and a triggered by it Action – one that shows in a dialog window up to three strings, one after the other. The generated code for this KP will subscribe to SIB for data matching the pattern given, and when SIB will send an update, a dialog will pop up with the text like ”Actuator's status changed: ex:ActuatorA on”.

Without going much into the details, the modeling process was the following:

1) Creating a Repository element in an empty diagram, and setting its *Url* attribute.
2) Engaging *Import model elements* plug-in and importing the same Repository but extended with metadata and task Ontology elements.
3) Engaging *Import model elements* plug-in and import-

ing the SIB with the domain Ontology.

4) Creating a KP element. Creating a Condition element. Connecting KP to SIB and to Condition. The relationships on connectors are set automatically.
5) Engaging *Generate a SIB Subscription* plug-in, selecting "Changed value of a Property" and then selecting *gc:hasStatus* property. The Composite will be added. Connecting its "Action" port to the Condition.
6) Engaging *Generate a generic Task* plug-in and importing *task:InformUser* task.
7) Engaging *Find an implementation* plug-in and selecting from the presented list *Show in Dialog* action – other options will include e.g. the standard Java print out. The Action and its Parameters will be added.
8) Setting constant value for the Parameter *s1* and then connecting *s2* and *s3* to the data ports of the subscription Composite. Connecting the event port of the Composite to the Action itself.
9) The model is ready. Engaging *Generate Java implementation* plug-in to generate Java code.

## V. Conclusions

In this paper, we described an ontology-driven approach to development of software applications for smart environments. In comparison to the traditional way of software development, our approach partially automates development processes, facilitates reuse of components through metadata-based discovery, and raises the level of abstraction of smart application development. All these aims at making development easier and faster, and also at enabling development by non-programmers. As the example in Section IV showed, as long as existing software components are sufficient, the development may not involve any coding and, moreover, the software composition process is made much faster through ontology/metadata support. We believe that such an example process (numbered list in Section IV) could already be performed by a person without software engineering background, but can be even further facilitated through the "wizards" mechanism.

The main inputs to our development process are ontologies and components' metadata, and these can be published in some way in a smart environment. An interesting and promising scenario would be then a case where a person entering the environment was able to compose an application matching his needs *on-the-fly*. Working on realizing such a scenario is an important direction for future work in SOFIA.

Other directions for future work include at least the following. First, we have to enable better and more flexible modelling of a KP behaviour, so that more complex behaviours can be developed and transformed into code. As mentioned before, we also plan to have code generation for other programming languages. Second, we are working on adding an *opportunistic* way of software composition, as in [9]. Instead of going from a developer-defined generic task to a software component, the system may automatically search and list the components that are *applicable* given the data and components already present in the application (e.g. "you have data on restaurants; would you like to show them on a map?"). Such a way of utilizing components' metadata is one of several topics (e.g. task ontologies is another) we study in our endeavour into *ontology-driven composition of software*, which is our main longer-term research direction, even beyond SOFIA.

## Acknowledgements

This work is performed in the project SOFIA, which is a part of EU's ARTEMIS JU. SOFIA is coordinated by Nokia and the partners include Philips, NXP, Fiat, Elsag Datamat, Indra, Eurotech, as well as a number of research institutions from Finland, Netherlands, Italy, Spain, and Switzerland.

## References

[1] P. Liuha, A. Lappeteläinen, and J.-P. Soininen, "Smart objects for intelligent applications – First results made open," *ARTEMIS Magazine*, vol. 5, pp. 27–29, 2009.

[2] F. Ruiz and J. R. Hilera, "Using ontologies in software engineering and technology," in *Calero, C., Ruiz, F., Piattini, M. (eds) Ontologies for Software Engineering and Software Technology*. Springer Verlag, 2006, pp. 49–102.

[3] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[4] Y. Singh and M. Sood, "Model driven architecture: A perspective," in *Proc. IEEE International Advance Computing Conference*, 2009, pp. 1644–1652.

[5] W3C, *Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering*, 2006, online: http://www.w3.org/2001/sw/BestPractices/SE/ODA/.

[6] A. Soylu and P. de Causmaecker, "Merging model driven and ontology driven system development approaches: Pervasive computing perspective," in *Proc. 24th Intl. Symposium on Computer and Information Sciences*, 2009, pp. 730–735.

[7] M. Vanden Bossche, P. Ross, I. MacLarty, B. Van Nuffelen, and N. Pelov, "Ontology driven software engineering for real life applications," in *Proc. 3rd Intl. Workshop on Semantic Web Enabled Software Engineering*, 2007.

[8] K. M. de Oliveira, K. Villela, A. R. Rocha, and G. H. Travassos, "Use of ontologies in software development environments," in *Calero, C., Ruiz, F., Piattini, M. (eds) Ontologies for Software Engineering and Software Technology*. Springer Verlag, 2006, pp. 276–309.

[9] M. P. Carlson, A. H. H. Ngu, R. Podorozhny, and L. Zeng, "Automatic mash up of composite applications," in *Proc. Intl. Conference on Service-Oriented Computing, LNCS Vol.5364*, 2008, pp. 317–330.

[10] D. Rubel, "The heart of Eclipse," *ACM Queue*, vol. 4, no. 6, pp. 36–44, 2006.

[11] Eclipse Foundation, *Graphical Modeling Framework*, online: http://www.eclipse.org/modeling/gmf/.