Lightweight Ontology-driven Representations in Pervasive Computing

Jarosław Domaszewicz and Michał Rój[1]

Institute of Telecommunications, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
j.domaszewicz@tele.pw.edu.pl, m.roj@tele.pw.edu.pl

Abstract. A clearly specified representation of diverse entities is needed to refer to them in pervasive computing applications. Examples of such entities include physical objects, operations, sensor and actuator resources, or logical locations. We propose a novel way to systematically generate representations of entities for programmable pervasive computing platforms made of tiny embedded nodes. Our original idea is to generate a very lightweight, though semantically-rich, representation from a possibly complex ontological specification. At the platform development phase, a domain ontology is used to describe the target environment. A preprocessing tool produces the ontology-driven, lightweight representation, which comes in two flavors: a human-readable one, to be used for programming, and a binary one, to be used at runtime. Our approach makes it possible to take advantage of all the benefits of ontology-based modeling and, at the same time, to obtain a representation light enough to be embedded in even the tiniest nodes.

1 Introduction

Pervasive computing applications may need to refer to a great variety of entities to perform their tasks[2]. Categories of such entities include physical objects, operations to be performed on those objects, various resources (e.g., sensors and actuators), and logical locations. The applications refer to entities to discover, control, or use them in some way. By "representation" we mean any agreed upon convention enabling the applications to unambiguously refer to entities. A programmable pervasive computing platform should clearly specify how to represent entities that might be of interest to the applications. Application programmers need a human-friendly representation (e.g., descriptive identifiers or function names) to refer to entities in their source code. At runtime, deployed applications and the pervasive computing platform need a common binary representation.

In many pervasive computing platforms the representation of entities is not generated in a systematic way. A common practice is to produce a representation in

[1] The order of authors was determined by a coin flip.

[2] The research reported in this paper has been partly supported by the Polish Ministry of Scientific Research and Information Technology, grant no. 3T11D 011 28.

an ad-hoc manner, e.g., by arbitrary assignment of identifiers to arbitrarily selected entities.

We believe that a good representation of entities in pervasive computing should result from a systematic procedure. Specifically, the representation should be derived from an explicit, formal domain model of the pervasive computing platform's target environment (i.e., the domain where the applications run). The model should be comprehensive enough to capture all relevant aspects of the domain, including possible diversity of entities. To ensure high quality of the model, it should be created by domain experts, not programmers.

Producing a so-called ontology is an excellent way to model a domain. Ontologies are in widespread use in Semantic Web, and they have been successfully used in pervasive computing. However, most ontology-based techniques require relatively big amounts of memory and processing power, especially when a domain ontology itself is used at runtime (which is the case for all ontology-based pervasive computing systems known to us).

Applying ontologies becomes challenging if the target pervasive computing platform consists exclusively of tiny, energy-constrained, battery-powered nodes, like Berkeley Motes [1]. As in such a system there is no room for any big central repository or server, the representation must be stored and processed locally, by the nodes themselves. It has to be extremely lightweight and compact. This case is the focus of the paper.

This paper makes the following contribution. We propose to methodically generate ontology-driven representations of entities for pervasive computing platforms made of tiny embedded nodes. Our original idea is to produce a very lightweight, though semantically rich, representation of entities (down to binary encodings) from a possibly complex ontological specification.

The paper is organized as follows. In Section 2, we discuss related work. In Section 3 we present our approach in general terms. Section 4 gives an example of a domain ontology and a lightweight ontology-driven representation and how the final representation is acquired. In Section 5, we give a specific example of how it can be used to represent sensor and actuator resources in a pervasive computing middleware. The paper is concluded in Section 6.

2 Related work

Ontology-based domain modeling has an established position in the field of pervasive computing. For example, a so-called GAS ontology is used to model the functionality of devices [2]. The ontology, which aims at augmented home objects (such as "eLamp," or "eBook"), defines operations that can be performed on devices (e.g., switch on/off). In a different approach [3], capabilities (sampling rate, physical units, etc.) of sensor nodes in a wireless sensor network are ontologically described, and the ontology is used mainly to dynamically calibrate the whole system.

In the above examples, the ontology itself is present at runtime. Handling an ontology directly is definitely not suitable for small, Berkeley Mote-class nodes. Even if the ontology is kept as small as possible (e.g., GAS-CO in the GAS architecture

[2]), and lightweight ontology languages are employed (e.g., OWL-Lite in [3]), a node has to be more a PDA than a mote. The primary difference between our approach and the above ones is that in our case not the ontology itself but an ontology-derived lightweight representation suitable for tiny embedded nodes is used at runtime.

We also identified several areas of ontology-derived software artifacts - in fields other than pervasive computing. Usually, ontological models are used to generate a class hierarchy in object-oriented languages (especially Java). For example, Jena API [4] includes a program called schemagen[3], which generates Java representations of concepts from an OWL [5] ontology. Also, a class generator has been included into the Protégé OWL plugin[4]. A general approach to mapping OWL into Java is discussed in [6]. Algorithms and simple heuristics to generate class diagrams from ontologies are presented in [7]. An ontology editor able to generate knowledge from ontologies in various formats (including Java classes) is introduced in [8].

Even though our approach shares some similarities with the above ones, representations of entities generated with our approach are lighter and more elementary and so can be aimed at tiny embedded nodes. In particular, we claim that exploring the possibility of deriving simple (yet semantically meaningful) binary encodings from a complex, abstract ontology has not been done before.

A very recent, promising attempt to employ ontologies in software engineering is presented in [9]. Our work can contribute to those efforts.

The techniques presented is this paper can be applied in existing pervasive computing platforms, in which the representations are produced in an ad-hoc way. To the best of our knowledge such platforms include Agilla [10], tinyDB [11], tinyLIME [12], as well as many others.

3 Lightweight ontology-driven representations

Our key idea is presented in Fig. 1. We propose a tool, called "ontology preprocessor," that takes a domain ontology as its input and produces a lightweight representation for a category of entities as its output. The representation comes in two parts, named O-API and O-ABI.

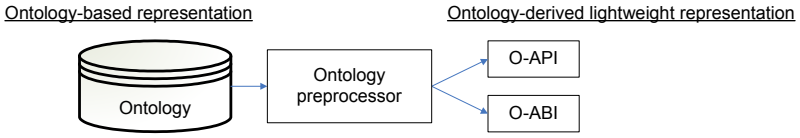

Fig. 1. Deriving lightweight representation from an ontology

[3] http://jena.sourceforge.net/how-to/schemagen.html
[4] http://protege.stanford.edu/plugins/owl/

The ontology is created by a domain expert. It describes the target environment (e.g., home, office), where a pervasive computing platform is to be deployed. In particular, it may classify and describe different kinds of objects, logical locations, or resources (e.g., sensors and actuators) that are common in the target environment. The ontology can be as big and complex as desired.

The ontology preprocessor, which derives the lightweight representation, is not tied to any specific ontology. It is ontology-independent, so ontologies for various domains can be used as its input. We are currently developing algorithms for ontology preprocessing. Some preliminary results are presented in the next section.

The ontology preprocessor produces the representation of entities in two flavors: O-API and O-ABI. They bear some resemblance to, but are not the same as, a regular API and ABI (i.e., Application Programming Interface and Application Binary Interface, respectively). The main difference between O-API and O-ABI is that the former is targeted at humans (programmers) and the latter at machines (compilers and the runtime system).

O-API is meant to be used by programmers to refer to entities in source code. In O-API, entities are represented by human-readable, meaningful names, e.g., constant identifiers. O-API should be simple enough to be usable by a programmer without any background in ontology engineering. Even though full O-API can be big (its size grows with that of the ontology), any single application is likely to use only a small subset.

O-ABI is meant to be used by applications to refer to entities when interacting with the system software of the pervasive computing platform. In O-ABI, entities are represented by simple binary encodings "understood" by the system software. Normally, there is a one-to-one correspondence between O-API names and O-ABI encodings.

The O-ABI representation is lightweight in that it can be embedded into even most severely constrained nodes. While full O-ABI can be quite big (just as the ontology and O-API), any single node is likely to be related to only a fairly limited number of entities. For example, an intelligent node embedded into an object is usually equipped with a handful of sensors and actuators. Handling a limited number of binary encodings is possible even if a node's processing power, memory, or available energy are extremely scarce.

4 Deriving O-API and O-ABI from ontology: an example

We now provide a simple example of what the input ontology and the derived lightweight representation may look like. Consider a pervasive computing platform where the entities to be represented are operations that can be performed on home objects by embedded nodes. A part of a home domain ontology for this case might be the one presented in Fig. 2. The ontology consists of three basic hierarchies: `HomeItem`, `Location`, and `Operation`. They classify home objects (for brevity we include light sources and meters only), logical locations of the objects, and possible operations, respectively.

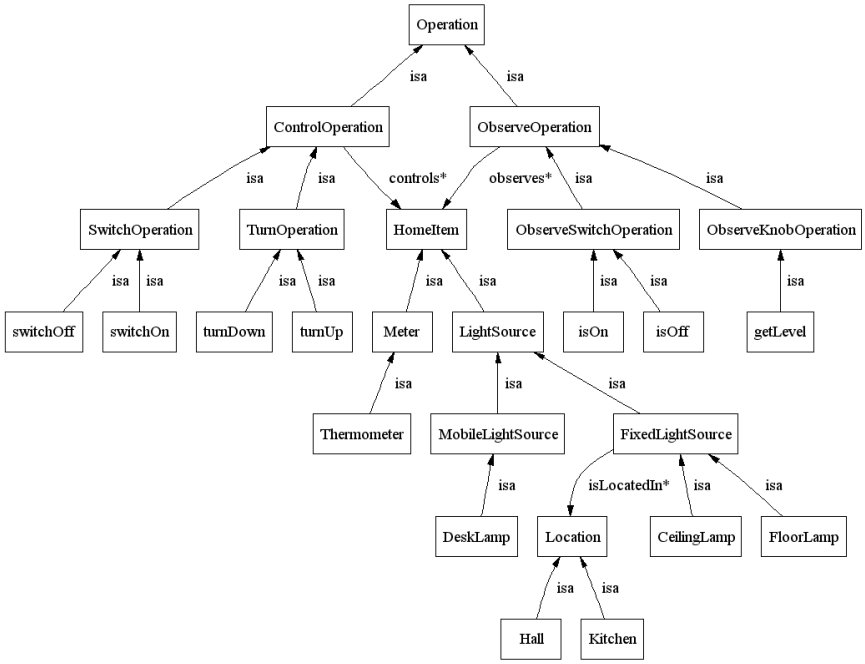

Fig. 2. A simple home domain ontology

All the `Operation` instances either affect an object's state (the `ControlOperation` operations) or to inquire about it (the `ObserveOperation` operations). The discrete state variables are handled with `SwitchOperation` and `ObserveSwitchOperation` operations, while the continuous state variables with `TurnOperation` and `ObserveKnobOperation` operations. The operations are linked to home item classes using the `controls` and `observes` properties. For example, since `ControlOperation` is linked to `HomeItem` using the `controls` property, `LightSource` can be controlled with any `ControlOperation` (note that properties are "inherited" here). In our ontology we assumed that only permanently attached objects (the `FixedLightSource` class) have a logical location.

A part of a lightweight representation that could be produced from this ontology by an ontology preprocessor is shown in Tab. 1. To produce entities we used a simple algorithm. For every "concrete" operation class (`turnUp`, `turnDown`, `switchOn`, `switchOff`, `isOn`, `isOff` and `getLevel`) we follow the `controls` and `observes` properties. We explore all possible paths from an `Operation` class to a `HomeItem` class (example paths are `turnUp-controls-FloorLamp` and `getLevel-observes-Thermometer`). Then we go further, by following the `isLocatedIn` property. For instance, for fixed light sources we explore paths consisting of three classes, such as `turnUp-controls-FloorLamp-islocatedIn-Hall` and

turnUp-controls-FloorLamp-isLocatedIn-Kitchen. Then we can assign an API name to every path (e.g., turnUp-controls-FloorLamp-islocatedIn-Hall becomes turnUpFloorLampInHall). For a more complex ontology, the algorithm can follow longer paths and produce more semantically rich operations.

Proper ontological modeling ensures that all the paths (and corresponding entities) are meaningful. For example, for mobile light sources there are no paths that include a logical location and so an entity like switchOnMobileLightSourceInKitchen is not produced by the algorithm. As another example, assume the controls operation is restricted not to take on values in the Meter class (the restriction not shown in Fig. 2). Then entities like turnUpThermometer are not produced.

Tab.1 O-API and O-ABI pairs for the category of operations, based on the ontology presented in Fig. 2. O-ABI encodings have been selected arbitrarily.

O-API	O-ABI
switchOnHomeItem	0x00
…	…
switchOnLightSource	0x10
…	…
switchOnMobileLightSource	0x40
…	…
switchOnFixedLightSource	0x50
…	…
switchOnFixedLightSourceInHall	0x60
switchOnFixedLightSourceInKitchen	0x61
…	…
switchOnCeilingLamp	0x70
…	…
switchOnCeilingLampInHall	0x80
…	…

Some remarks are in order at this point. The entities (in this case – operations) are no longer selected in an arbitrary fashion. The set of entities is systematically derived from the ontology. Thus, ontology preprocessing produces not only representations of entities, but, in a sense, the entities themselves.

In our example, there is no direct mapping between entities and existing classes of the domain ontology (or their instances). Rather, the entities are "produced" by manipulating and combining concepts present in the domain ontology. Thus, the ontology processor can be considered as a value adding tool. We are working on an approach allowing the entities to be expressed in OWL, as new concepts based on the ones present in the original ontology.

Different relationships captured by the ontology may be reflected in some structuring of the derived set of entities. One example is a hierarchical structuring. For example, the increasing specificity in the object hierarchy leads to increasingly specific operations (compare switchOnLightSource, switchOnFixedLight-Source, and switchOnCeilingLamp). Whenever an object has an additional attribute (e.g., a logical location), an even more specific operation (e.g., switchOn-CeilingLampInKitchen) can be produced. Another example of structuring the set of entities is semantically organizing their binary encodings (explained in Section 5).

The entities might include quite complex and abstract concepts ("semantic richness"). For example, `switchOnCeilingLampInKitchen` conveys information on what activity is to be taken (switching on), what the object of the operation is (a ceiling lamp), and where the operation is to be performed (in the kitchen). Even such semantically rich entities are ultimately represented by simple O-ABI encodings.

Even though the example covers the category of operations, our approach could be applied to other categories of entities as well. Some obvious examples are the objects themselves (e.g., `KitchenCeilingLamp`), sensors and actuators embedded in the objects (e.g., `DeskLampSwitch`), logical locations (e.g., `HallWithWallLamp`), and events generated by users (e.g., `CeilingLampSwitchedOn`). The choice of category depends on the architecture of a pervasive computing platform.

A lightweight representation of entities, like the one presented in Fig. 3., can in principle be produced by hand in an ad-hoc manner. However, such a task quickly becomes unfeasible and the resulting representation hardly maintainable, even if the domain is described with only a moderate number of concepts.

5 ROVERS: exposing a node's resources through a virtual machine

In this section we provide an example of applying a lightweight, ontology-driven representation in a pervasive computing platform. The example is based on ROVERS – a middleware that we are developing [13]. The middleware targets peer-to-peer networks of constrained heterogeneous nodes. The nodes are embedded in everyday objects and equipped with different combinations of resources, primarily sensors and actuators.

In ROVERS, applications are composed of tiny collaborating mobile code units, called micro-agents[5]. To enable code mobility, a virtual machine is deployed on each node. As nodes differ in terms of resources they are equipped with, so do their virtual machines. A node's virtual machine is specified by instructions it can execute. All possible instructions are classified into generic and non-generic. Generic ones, like arithmetic operations or program flow instructions are supported by the virtual machine on every node.

In ROVERS, a node's sensor and actuator resources are represented by (non-generic) instructions of the node's virtual machine. For example, a temperature sensor may be represented by the `getLevelThermometer` instruction, while a desk lamp's switch by the `switchOnDeskLamp`, `switchOffDeskLamp`, `isOnDeskLamp`, and `isOffDeskLamp` instructions. A node's virtual machine supports only those non-generic instructions that represent the node's sensors and actuators.

An instruction (generic or non-generic) has a human-readable name and a binary encoding. The names constitute a specific assembly language, while the encodings – a machine language. What is unique is that the non-generic parts of the both languages are derived from a domain ontology as O-API and O-ABI, respectively.

[5] We cover only those aspects of ROVERS that are directly related to the topic of this paper.

An instruction encoding space may look like the one presented in Fig. 3. The instructions shadowed in Fig. 3 (a) and (b) might be supported by nodes embedded into a desk lamp and a thermometer, respectively. As each node is equipped with only a small number of sensors and actuators, handling the ontology-derived representation of resources amounts to interpreting a couple of encodings.

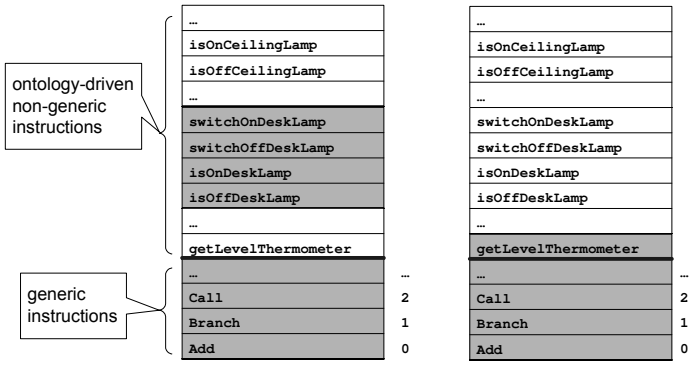

Fig. 3. An example of the instruction encoding space for ROVERS virtual machines. A specific virtual machine supports the shadowed items only: (a) a desk lamp, (b) a thermometer

5.1 Semantically-structured binary encodings

The instruction encoding space presented in Fig. 3 does not suggest any structuring of the binary encodings (O-ABI). However, deriving the encodings from an ontology, as advocated in this paper, gives rise to an additional benefit – being able to automatically organize them based on their semantics. This subsection gives an example of such O-ABI structuring. The goal of this particular one is to reduce the size of ROVERS micro-agents' binaries (and so the energy cost of their mobility).

The structuring is based on the following observation. Consider the home environment domain. Assume that the ontology classifies domain concepts into sub-domains. Examples might include the "physical" sub-domain (temperature, pressure, humidity, etc.), the lighting sub-domain (ceiling lamps, desk lamps, etc.), or the heating sub-domain. Since each micro-agent should do a single job well, non-generic instructions used by most micro-agents are likely to originate exclusively from a single sub-domain. For example, a temperature reporting micro-agent might use the `getLevelThermometer` instruction, originating from the "physical" sub-domain, as its only non-generic instruction. Similarly, a "light manager" micro-agent would use instructions from the lighting sub-domain. Of course, a complete application will likely include micro-agents working in different sub-domains.

The above observation could be used to structure the encoding space of the ROVERS virtual machine instructions. Assume there are no more than 128 generic

instructions and no more than 128 non-generic ones originating from a single sub-domain. Then the instruction encoding space could be the one presented in Fig. 4.

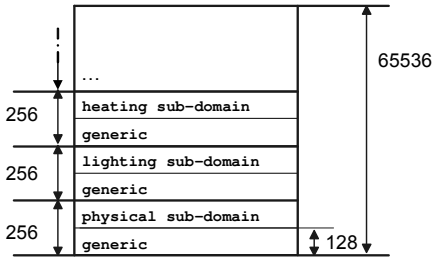

Fig. 4. Semantically-structured O-ABI

The space is divided into 256-instruction segments. The lower half of each segment is occupied by the generic instructions (which in effect have multiple encodings), while the upper half belongs to non-generic ones originating from a single sub-domain. Assume there is a generic "segment prefix" instruction that specifies the segment that the subsequent instructions belong to (until the next segment prefix instruction). Then, in spite of the fact that the encoding allows as many as $2^{16}/2 = 32768$ non-generic instructions, each instruction can be encoded with only 8 bits. Obviously, the average encoding efficiency (number of bits per instruction) depends on how often the segment prefix instruction is used. Since a typical micro-agent is limited to a single sub-domain, only one segment prefix instruction per micro-agent is needed, and the average encoding efficiency approaches 8 bits per instruction.

Even though the example pertains to the ROVERS middleware, semantic structuring of binary encodings seems to be a promising technique of wider applicability. It is easy to implement as a feature of O-ABI generation.

6 Summary and future work

This paper presents a novel approach to using ontologies in the development of pervasive computing platforms. We made initial experiments with generating representations from OWL ontologies, using the Jena API [4]. Our plan now is to develop more advanced ontology preprocessing algorithms. One of the challenges is to make them ontology-independent, so ontologies for various domains can be used as inputs. In addition, we plan to make the ontology preprocessor tunable so that, for example, the "granularity" of generated entities can be specified as a parameter.

References

1. Hill, J., Horton, M., Kling, R., and Krishnamurthy, L.: The Platforms Enabling Wireless Sensor Networks. Communications of the ACM, 2004. 47(6).
2. Christopoulou, E., Goumopoulos, C., Zaharakis, I., and Kameas, A.: An Ontology-based Conceptual Model for Composing Context-Aware Applications. in Workshop on Advanced Context Modelling, Reasoning and Management in conjunction with Sixth International Conference on Ubiquitous Computing (UbiComp 2004). 2004. Nottingham, England.
3. Avancha, S., Patel, C., and Joshi, A.: Ontology-driven Adaptive Sensor Networks. in First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous '04). 2004. Boston, USA.
4. McBride, B.: Jena: A Semantic Web Toolkit. IEEE Internet Computing, 2002. 6(6): p. 55 - 59.
5. Bechhofer, S., et al.: OWL Web Ontology Language Reference, http://www.w3.org/TR/owl-ref/, July 2005
6. Kalyanpur, A., Pastor, D.J., Battle, S., and Padget, J.: Automatic mapping of OWL ontologies into Java. in Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE). 2004.
7. Bonancin, R. and Baranauskas, C.C.: From Ontology Charts to Class Diagrams: Semantic Analysis Aiding Systems Design. in International Conference on Enterprise Information Systems. 2004. Porto, Portugal.
8. Mian, P.G. and de Almeida Falbo, R.: Building Ontologies in a Domain Oriented Software Engineering Environment. in IX Congreso Argentino de Ciencias de la Computación. 2003. La Plata, Argentina.
9. Tetlow, P., et al.: Ontology Driven Architectures and Potential Uses of the Semantic Web in Software Engineering, http://www.w3.org/2001/sw/BestPractices/SE/ODA/, June 2005
10. Fok, C.-L., Roman, G.-C., and Lu, C.: Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. in 24th International Conference on Distributed Computing Systems (ICDCS'05). 2005. Columbus, Ohio, USA.
11. Madden, S., Franklin, M.J., Hellerstein, J.M., and Hong, W.: The design of an acquisitional query processor for sensor networks. in International Conference on Management of Data. 2003. San Diego, California.
12. Curino, C., et al.: Tiny Lime: Bridging Mobile and Sensor Networks through Middleware. in 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005). 2005. Kauai Island, Hawaii.
13. ROVERS Homepage, http://meag.tele.pw.edu.pl/ROVERS/index.htm, July 2005