Opportunistic Pervasive Computing with Domain-oriented Virtual Machines

J. Domaszewicz, M. Rój, A. Pruszkowski
Institute of Telecommunications, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
{j.domaszewicz, m.roj, a.pruszkowski}@tele.pw.edu.pl

Abstract

The paper targets heterogeneous sensor-actuator networks, in which nodes differ as to resources (sensors and actuators) they are equipped with. Each node contributes its specific sensors and actuators to be used by applications. The key assumption of "opportunistic pervasive computing" is that the actual mix of nodes (and that of available resources) is not known in advance to the programmer. An opportunistic pervasive computing application is supposed to take the best advantage of whatever sensors and actuators happen to be available in the network. The paper presents a technique that can be used in middleware layers supporting such applications. The technique uses virtual machines to orderly expose sensor and actuator resources of a node to the programmer. The virtual machines are domain-oriented, node specific, and able to work with the resources at multiple levels of abstraction. They can be implemented on severely constrained nodes (e.g., of the TinyOS class).

1. Introduction

The paper targets heterogeneous sensor-actuator networks of tiny, constrained, embedded nodes (e.g., a home network of nodes embedded into everyday objects, like a lamp or refrigerator). The network is heterogeneous in that the nodes differ in sensor and actuator resources they are equipped with (we consistently use the term "resources" to denote a node's sensors and actuators). A node embedded into a lamp may offer an actuator allowing an application to switch the lamp on and off; it may also contain a sensor that makes it possible to determine the lamp's current state. Clearly, a node embedded into a refrigerator is likely to offer a collection of sensors and actuators of very different functionality (e.g., a temperature sensor and a sensor detecting a door opening event). Each node contributes its specific sensors and actuators to be used by pervasive computing applications.

The key assumption behind opportunistic pervasive computing is that the actual mix of nodes (and thus that of available sensors and actuators) is not fully known to the application programmer. The assumption is true whenever (a) applications are written not for a specific network, but for an entire class of target networks, and (b) the target networks unavoidably differ in their composition. The former statement can be justified in terms of development cost. The latter one is true whenever networks grow without any master plan (e.g., this is the case for home networks of intelligent everyday objects).

By definition, an opportunistic pervasive computing application is supposed to take the best advantage of (offer the best possible functionality with) whatever resources happen to be available in the network where the application runs. One can think of such applications in two ways. The first one, which could be called *bottom up*, is to assume some minimal resources as a prerequisite, and search for opportunities to make a better job, if more than the minimal resources are available. The other one, *top down*, is to assume some maximal resources that the application is capable of using and scale down gracefully as the actual resources are less than maximal. A similar concept of opportunistic computing is presented in [1].

Clearly, the development of opportunistic pervasive computing applications is far from trivial. We would like to ease the programmer's job by pushing a number of recurring tasks into a middleware. Certainly, many complex tasks, especially those of "algorithmic" nature are likely to stay in the application layer. However, even if the programmer is to be freed only from minor (but troublesome) "programming" chores, the effort seems to be worthwhile.

This paper contributes a technique that can be used in middleware layers for opportunistic pervasive

This work was supported in part by the Polish Ministry of Science and Higher Education, project no. 3 T11D 011 28.

computing with heterogeneous sensor-actuator networks. The technique, *domain-oriented virtual machines*, is a part of the ROVERS middleware [2]. However, we believe that it is of much wider applicability. It can be applied in a whole class of mobile code-based systems. Importantly, domain-oriented virtual machines can be implemented on severely constrained nodes (e.g., of the TinyOS class).

This paper is organized as follows. Domain-oriented virtual machines are described in detail in Section 2. A class of middleware layers in which the technique can be applied is discussed in Section 3. Possible applications to opportunistic pervasive computing are outlined in Section 4. Related work is reported in Section 5. The paper is concluded in Section 6.

2. Domain-oriented virtual machines

Each node of a heterogeneous sensor-actuator network has its own mix of resources (sensors and actuators). In this section we present a method to systematically expose a node's resources to the programmer. We assume that each node is equipped with a virtual machine (the primary motivation for the virtual machine is to support mobile code). In our approach, the resources of a node are available through the node's virtual machine. The virtual machines, as presented in this paper, are (a) domain-oriented, (b) node-specific, and (c) able to work with resources at multiple levels of abstraction.

2.1. Virtual machine architecture

The virtual machine architecture is tailored towards event-driven programming. The virtual machine generates events and executes instructions. Software running on the virtual machine consists of event handlers, and each event handler is composed of instructions.

We start by specifying the sets of all possible events and instructions, **E** and **I**, respectively. Both events and instructions can be classified into *generic* and *non-generic*: $\mathbf{E} = \mathbf{GE} \cup \mathbf{NGE}$ and $\mathbf{I} = \mathbf{GI} \cup \mathbf{NGI}$. Generic events and instructions are supported by all virtual machines, no matter what the resources of the underlying nodes are. An example of a generic event is `TimerEvent`, the timer expiry event (it is assumed that each node is equipped with a timer). Examples of generic instructions are typical and include arithmetic instructions (`add`), data movement instructions (`load`), program control instructions (`jump`), etc. There is nothing special about generic events and instructions.

Non-generic events and instructions, the elements of **NGE** and **NGI**, are used to expose sensors and actuators to the programmer. We postulate that they represent the underlying resources not at a low, physical level, but at an abstract, *domain-oriented* level (thus making the virtual machines themselves domain-oriented). A specific non-generic event or instruction should imply the meaning of the interaction with the resource, the kind of object the node is embedded in, and possibly the location of the object, all expressed in some high-level domain terminology.

For example, consider the domain of home objects and a node that is embedded in a ceiling lamp located in a kitchen. Assume the node is equipped with a switch. The lamp can be switched on and off either by a human or programmatically. Then the lamp switch could be represented by two non-generic events and three non-generic instructions, as shown in Fig. 1. The figure also shows a possible representation, two non-generic events, for a sensor built into a refrigerator's door. Note the level of abstraction and "semantic richness" of the non-generic items shown in Fig. 1.

Kitchen ceiling lamp switch	
events	CeilingLampInKitchenSwitchedOnEvent CeilingLampInKitchenSwitchedOffEvent
instructions	switchOffCeilingLampInKitchen switchOnCeilingLampInKitchen isCeilingLampInKitchenOn
Refrigerator door sensor	
events	RefrigeratorDoorOpenEvent RefrigeratorDoorClosedEvent
instructions	-

Figure 1. Non-generic instructions and events for an actuator and a sensor

The classification of all the instructions and events into generic and non-generic is illustrated in Fig. 2 (the home environment being the target domain). Just a few representative samples are given for each of the four categories. In particular, the sets of non-generic items can be very large. To properly describe a domain, **NGE** and **NGI** may have hundreds or even thousands of elements.

	Instructions (I)	*Events (E)*
Generic (G)	GI (samples): add push call	GE (samples): OnceEvent TimerEvent PowerDownEvent
Non-generic (NG)	NGI (samples): switchOnDisplay switchOnPDADisplay switchOffPDADisplay isOnDeskLamp switchOffHomeItemInKitchen turnDownCeilingLampInHall	NGE (samples): LightSourceSwitchedOffEvent CeilingLampInDiningRoomEvent DisplayInHallSwitchedOnEvent LightSourceTurnedDownEvent LightSourceEvent CeilingLampInKitchenEvent

Figure 2. Classification of the home domain virtual machine building blocks

Any specific node is likely to be equipped with only a handful of resources. For example, a node embedded into a lamp may not have other resources but the lamp switch. As a result, most virtual machines are going to support only a few non-generic events and instructions (tiny subsets of the whole sets **NGE** and **NGI**). Moreover, due to the network heterogeneity, the subsets for two different virtual machines are most likely to be different as well. To reflect this in our notation, the subsets of **NGE** and **NGI** supported by a virtual machine **v** will be denoted by **NGE**(**v**) and **NGI**(**v**), respectively. The *event set* and the *instruction set* of a virtual machine **v** will therefore be $\mathbf{E}(\mathbf{v}) = \mathbf{GE} \cup \mathbf{NGE}(\mathbf{v})$ and $\mathbf{I}(\mathbf{v}) = \mathbf{GI} \cup \mathbf{NGI}(\mathbf{v})$. Virtual machines are *node-specific* in that their event and instructions sets differ: on top of all generic items, each virtual machine supports its unique collection of non-generic ones.

The decision as to which non-generic items from **NGE** and **NGI** should a virtual machine support is made by the designer of the underlying node. The key factors are the resources of the node and the functionality of the object it is to be embedded in. For example, a virtual machine in a desk lamp node may have the instruction set shown in Fig. 3. Three non-generic instructions have been added to represent a switch, which happens to be the only resource of the node.

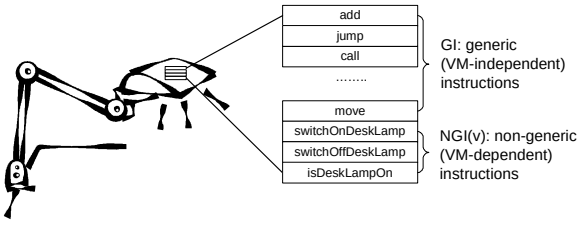

Figure 3. Instruction set I(v) for a desk lamp virtual machine v

A noteworthy feature is that generic and non-generic items are used in a program in exactly the same way, simply as events and instructions.

2.2. Ontology-driven non-generic items

The sets **NGE** and **NGI** are meant to include all meaningful events and instructions that can possibly be considered within the target domain. It would be next to impossible to produce them by hand. Instead, what is needed is a formal, modular model of the domain, out of which all the non-generic events and instructions could be systematically derived by a computer.

In our approach, an ontology is used to model the target domain. The ontology contains basic domain-specific concepts and relationships. The ontology is developed first (we use the OWL language). Then, a software tool, called *ontology preprocessor*, accepts the ontology as its input and produces all the non-generic events and instructions as its output (see Fig. 4) [3]. The ontology preprocessor derives the non-generic items by automatically combining existing concepts (OWL classes) into more complex ones.

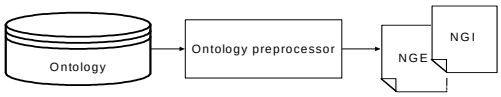

Figure 4. Ontology preprocessor

In our current implementation, the original ontology organizes its concepts into multiple modular hierarchies. For example, there may be three of those for the home domain: an interaction with resources hierarchy, an object hierarchy, and a location hierarchy (see Fig. 5 for a greatly simplified view).

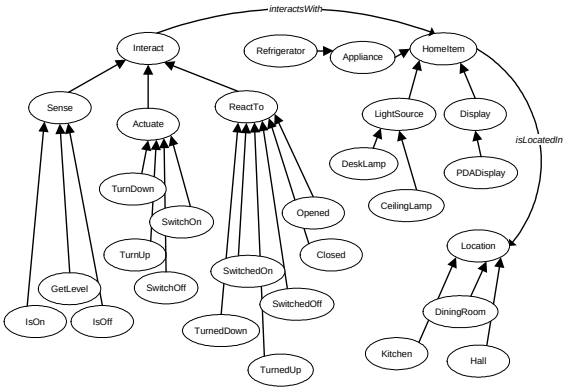

Figure 5. Concept hierarchies for the home domain

The ontology preprocessor produces non-generic items by following relations and combining concepts (classes) from different hierarchies. For example, combining `TurnUp` from the interaction hierarchy, `CeilingLamp` from the object hierarchy, and `Kitchen` from the location hierarchy gives rise to the `turnUpCeilingLampInKitchen` instruction[1]. (Not all combinations make sense; the ontology preprocessor generates only meaningful ones.)

The concept combining process is fully OWL-compliant in that a combined concept can be expressed

[1] To differentiate instructions from events, we do not capitalize the names of the former.

as a new ontology (OWL) class. For example, the above instruction could be written as in Fig. 6 (we use the more compact description logic notation, rather than OWL). The capability to turn down a specific ceiling lamp in a specific kitchen could be considered an instance of the instruction class.

$$turnUpCeilingLampInKitchen \equiv TurnUp \sqcap \exists interactsWith.(Lamp \sqcap \exists isLocatedIn.Kitchen)$$

Figure 6. Non-generic instruction as a computer-derived ontology class

Even though it may be useful to look at a non-generic item as a class (see the next subsection), for most purposes it suffices to consider it to be a simple, "atomic" entity. In particular, we never work directly with any instances of an event or instruction class.

As can easily be seen, even an original ontology of a fairly limited size can give rise to a huge number of derived concepts. This is not a problem, however, as the latter are generated by a computer. A resulting big catalogue of non-generic items can be conveniently browsed by node designers and programmers alike.

2.3. Event and instruction inheritance

The above derivation procedure implies a useful relationship between non-generic items. Specifically, when a non-generic item is interpreted as an ontology class (in fact, it is a class), it may happen to be an (ontological) subclass of some other non-generic item. For example, the `switchOnDeskLamp` instruction class is an (ontological) subclass of the `switchOnLightSource` instruction class. This relationship strongly resembles class inheritance known in object-oriented programming, and we borrow the term. We say that `switchOnDeskLamp` *inherits* from `switchOnLightSource`.

To provide a general definition, the non-generic instruction `A` inherits from the non-generic instruction `B` if the class corresponding to `A` is an (ontological) subclass of the class corresponding to `B`. We then call `A` a *sub-instruction* of `B`, and `B` a *super-instruction* of `A`. Such an inheritance relationship can be similarly defined for events. Parts of the "inheritance graphs" for non-generic events and instructions derived from the ontology shown in Fig. 4 are presented in Fig. 7 and 8. A graph of all the events from which `LightSourceInKitchenSwitchedOnEvent` inherits (all of its super-events) is given in Fig. 9.

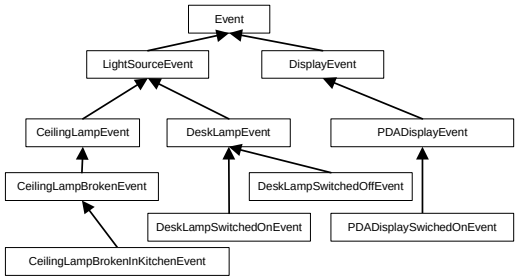

Figure 7. Part of non-generic event inheritance graph

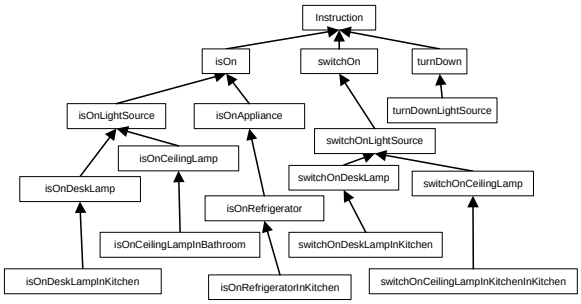

Figure 8. Part of non-generic instruction inheritance graph

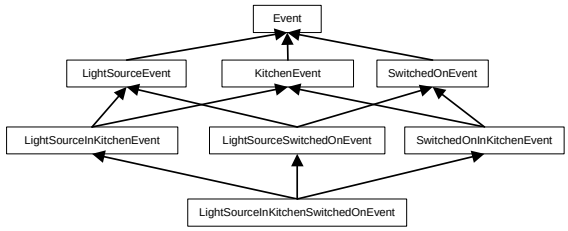

Figure 9. All super-events for `LightSourceinKitchenSwitchedOnEvent`

In object-oriented programming, if the `Car` class is a subclass of the `Vehicle` class, then every car is a vehicle. We adopt a similar approach for events and instructions. However, at this point it suffices to treat non-generic item classes as atomic entities: we do not consider instances (objects) at all. We identify non-generic items directly. For example, as `CeilingLampEvent` inherits from `LightSource-Event`, we say that `CeilingLampEvent` is `LightSourceEvent`. Similarly, as `switchOnDesk-Lamp` inherits from `switchOnLightSource`, we say that `switchOnDeskLamp` is `switchOnLightSource`. Intuitively, referring to the last example, to switch a desk lamp is definitely to switch a light source. The former offers a higher level of *specificity*, and the

latter – a higher level of *abstraction*. Obviously, the identification is only one-way: to switch a light source is not necessarily to switch a desk lamp. An operational significance for this inheritance-based one-way identification is presented below.

2.4. Micro-agent

In this section we describe a mobile agent that is developed for a domain-oriented virtual machine. As its architecture is simple enough for it to run on severely constrained nodes, we call it a *micro-agent* [2]. We assume that a pervasive computing application is composed of communicating and cooperating micro-agents. To take advantage of all resources available in a heterogeneous sensor-actuator network, micro-agents comprising a single application usually run on different nodes.

In accordance with event-driven programming, a micro-agent is simply a collection of event handlers for some set of events; the handlers, as usual, consist of instructions. Both the events and instructions can be either generic or non-generic. A micro-agent's execution amounts to invoking its handlers in response to corresponding events.

As, by definition, not all the non-generic events and instructions are available on every virtual machine, it is important to clearly identify all the non-generic items used by a micro-agent. The sets of all non-generic events, for which the micro-agent $\mathbf{m}$ has a handler and of all non-generic instructions occurring in any of its handlers are denoted by $\mathbf{NGE}(\mathbf{m})$ and $\mathbf{NGI}(\mathbf{m})$, respectively.

2.5. Micro-agent hosting

A micro-agent can run on a node only if sensors and actuators that the micro-agent needs to do its job are available there. As explained, a micro-agent refers to sensors and actuators through non-generic events and instructions. Therefore, if a micro-agent uses a non-generic item, one implicitly requires that the item be supported by a target virtual machine. Clearly, not every micro-agent can run on every virtual machine. A micro-agent $\mathbf{m}$ can run on a virtual machine $\mathbf{v}$ if and only if $\mathbf{v}$ supports all the non-generic items used by $\mathbf{m}$. Without inheritance, one would write the condition simply as $\mathbf{NGE}(\mathbf{m}) \subset \mathbf{NGE}(\mathbf{v})$ and $\mathbf{NGI}(\mathbf{m}) \subset \mathbf{NGI}(\mathbf{v})$. However, this time, when talking about non-generic items supported by a virtual machine, we take into account their inheritance-based one-way identification.

For example, let a micro-agent use the `switchOnLightSource` instruction, which is not in $\mathbf{NGI}(\mathbf{v})$. Without inheritance-based identification, the micro-agent could not run on $\mathbf{v}$. However, if $\mathbf{NGI}(\mathbf{v})$ includes `switchOnDeskLamp`, which is a sub-instruction of `switchOnLightSource`, then the latter is also implicitly supported by $\mathbf{v}$. In that case the micro-agent can run on $\mathbf{v}$ (provided there are no problems with other non-generic items used by the micro-agent). The `switchOnDeskLamp` sub-instruction is executed whenever the `switchOnLightSource` super-instruction were to be executed (note that the micro-agent does not specify what kind of light source is to be switched on). Similarly, if a micro-agent has a handler for the `DeskLampEvent` event, which is not in $\mathbf{NGE}(\mathbf{v})$, but the `DeskLampSwitchedOnEvent` sub-event is in $\mathbf{NGE}(\mathbf{v})$, then the handler for the super-event is invoked whenever the sub-event occurs.

A virtual machine, on which a micro-agent can run, is called a *host* for the micro-agent. A virtual machine is a host if and only if every non-generic item used by the micro-agent is available at a level of specificity equal to or higher than that required by the micro-agent.

To define a host more formally, we introduce some notation. For an event (instruction) x, let $\mathrm{super}(x)$ denote the set of all super-events (super-instructions) of x in $\mathbf{NGE}$ ($\mathbf{NGI}$). Similarly, let $\mathrm{sub}(x)$ denote the set of all sub-events (sub-instructions) of x in $\mathbf{NGE}$ ($\mathbf{NGI}$). (Note that $\mathrm{super}(x)$ and $\mathrm{sub}(x)$ include x itself). Finally, let $\mathrm{host}(\cdot,\cdot)$ be the predicate representing the relation of being a host.

A virtual machine $\mathbf{v}$ is a host for a micro-agent $\mathbf{m}$, ($\mathrm{host}(\mathbf{v},\mathbf{m})$ is true) if and only if $\mathbf{NGE}(\mathbf{m}) \subset \bigcup_{e \in \mathbf{NGE}(\mathbf{v})} \mathrm{super}(e)$ and $\mathbf{NGI}(\mathbf{m}) \subset \bigcup_{i \in \mathbf{NGI}(\mathbf{v})} \mathrm{super}(i)$. Equivalently, $\mathrm{host}(\mathbf{v},\mathbf{m})$ is true if and only if for each $e \in \mathbf{NGE}(\mathbf{m})$, we have $\mathrm{sub}(e) \cap \mathbf{NGE}(\mathbf{v}) \neq \emptyset$, and for each $i \in \mathbf{NGI}(\mathbf{m})$, we have $\mathrm{sub}(i) \cap \mathbf{NGI}(\mathbf{v}) \neq \emptyset$.

2.6. Micro-agent execution

We now describe how a micro-agent is executed on a virtual machine that is a host for the micro-agent. There are two basic processes involved: invoking event handlers and executing instructions. We cover events first. An event handler is invoked in two situations: (a) when a generic event occurs, and the micro-agent has a handler for it, and (b) when a non-generic event e occurs ($e \in \mathbf{NGE}(\mathbf{v})$), and the micro-agent has a handler for a super-event of e (which may be e itself). In the latter case we have $\mathrm{super}(e) \cap \mathbf{NGE}(\mathbf{m}) \neq \emptyset$.

Actually, when a non-generic event e occurs, there may be more than one super-event of e, for which there is a handler (i.e., the set $\text{super}(e) \cap \mathbf{NGE}(\mathbf{m})$ may have more than one element). In that case, the handlers for all the super-events of e are invoked. The handlers for more-specific super-events of e are invoked before those for less specific ones.

The event handling mechanism leads to noteworthy "playing with the levels of abstraction." Let for some $e \in \mathbf{NGE}(\mathbf{m})$, the set $\text{sub}(e) \cap \mathbf{NGE}(\mathbf{v})$ contain more than one element, as in Fig. 10. There, whichever of the three sub-events occurs, it is handled by a single handler – that of the super-event `LightSourceEvent`. It can be said that the micro-agent does not distinguish between the three sub-events; at the micro-agent's level of abstraction, all of them are handled as a single event.

Now, let for some non-generic event $e \in \mathbf{NGE}(\mathbf{v})$, the set $\text{super}(e) \cap \mathbf{NGE}(\mathbf{m})$ contain more than one element, as in Fig. 11. There, whenever `Ceiling-LampInKitchenSwitchedOnEvent` occurs, it is handled by three different handlers – those of its super-events. It can be said that the micro-agent handles the same event at three different levels of abstraction.

As for the micro-agent's instructions, generic ones are simply executed. For a non-generic instruction $i \in \mathbf{NGI}(\mathbf{m})$, its sub-instruction supported by the virtual machine (i.e., an element of $\text{sub}(i) \cap \mathbf{NGI}(\mathbf{v})$), which could be i itself, is executed. If there is more then one sub-instruction (i.e., $\text{sub}(i) \cap \mathbf{NGI}(\mathbf{v})$ contains more than one element), it is up to the virtual machine to pick the sub-instruction to be executed. For example, in a case of a more advanced ontology than the one shown in Fig. 5, $\mathbf{NGI}(\mathbf{v})$ could include the `alertSoundLowPitch` and `alertSoundHighPitch` instructions. If a micro-agent used `alertSound`, then it would be up to the virtual machine to alert with a low pitch sound or a high pitch sound.

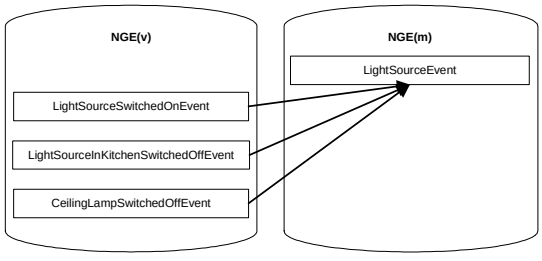

Figure 10. Multiple events handled at a higher level of abstraction as a common super-event

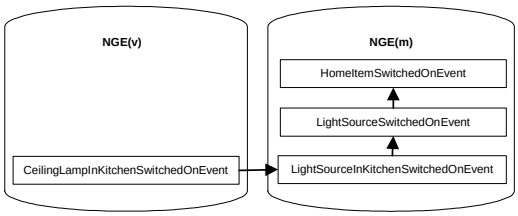

Figure 11. Single event handled at different levels of abstraction

2.7. Implementation issues

The technique of domain-oriented virtual machines is quite lightweight and can be implemented on tiny sensor nodes (e.g., of the TinyOS class). Only an implementation of the non-generic part of a virtual machine needs to be considered. A simple approach is as follows (we focus on events for the sake of example). Let each event $e \in \mathbf{NGE}$ have a unique numerical *event code*, which can be stored as a binary string (the codes can be generated by the ontology preprocessor). Then for a node's virtual machine $\mathbf{v}$ and for each event $e \in \mathbf{NGE}(\mathbf{v})$, the node should store the codes of all the super-events of e (i.e., the elements of $\text{super}(e)$). As can easily be seen, this information and a simple search suffices to check if, for a given micro-agent $\mathbf{m}$, $\mathbf{NGE}(\mathbf{m}) \subset \bigcup_{e \in \mathbf{NGE}(\mathbf{v})} \text{super}(e)$ holds (this is done to determine if the virtual machine is a host for the micro-agent). It is equally simple to find the set $\text{super}(e) \cap \mathbf{NGE}(\mathbf{m})$ (this is done to implement event handling). Essentially the same approach can be applied for non-generic instructions.

3. Prospective middleware layers

The technique of domain-oriented virtual machines is of use in a whole class of middleware layers for heterogeneous sensor-actuator networks.

A prospective middleware, in which the technique is applicable, satisfies the following basic architectural assumptions: (1) each node of the network has a domain-oriented, node-specific virtual machine; (2) a pervasive computing application (or at least a part of it) consists of mobile agents, whose architecture matches that of the domain-oriented virtual machine (one example of such a mobile agent architecture, the micro-agent, has been presented); (3) the application interacts with the environment by means of the mobile agents, through non-generic events and instructions.

A further architectural assumption, given below, seems to ensure that the technique of domain-oriented

virtual machines offers most value. The assumption goes beyond the approach presented so far in the paper.

The middleware layer preferably offers a primitive enabling a mobile agent to dispatch another mobile agent. The dispatching primitive is in fact a request that an instance of the dispatched agent be created on some of its hosts. The primitive may have modes, e.g., a broadcast mode (instantiate on all available hosts) and an anycast mode (instantiate on a single host). If no host is found, then no instance is created. The dispatching service is implemented by the middleware through appropriate host-finding routing protocols. A part of the implementation is a module (present in the middleware at each node) that evaluates the host(**v**, **m**) predicate for a node's virtual machine **v** and a micro-agent **m**.

If non-generic events and instructions (or the underlying sensors and actuators) are thought of as services offered by nodes, then the agent dispatching primitive entails an implicit (transparent to the programmer) service discovery.

Note that we do not assume anything about how a prospective middleware layer takes care of communications and coordination between the mobile agents. One approach is presented in [2]. However, many others are possible.

4. Applications to opportunistic computing

Domain-oriented virtual machines seem to be a convenient facility in opportunistic pervasive computing. In this section we provide some arguments to justify the claim. We assume that a middleware layer incorporating domain-oriented virtual machines satisfies the assumptions presented above.

The key advantage of resource (service) discovery done by dispatching a mobile agent is that sought resources can be described (and requested) at different levels of abstraction; this offers the programmer great flexibility in defining her needs. It is possible to specify different aspects of a resource with different specificity. For example, the embedding object can be described in quite general terms, while the location – quite precisely (or vice versa).

One approach to an unknown sensor-actuator network is to use a high level of abstraction (i.e., to dispatch agents, which are not very demanding in terms of specificity). Some matching resources will probably be found at that level, although at the expense of low "resolution" of the working of an application.

If the above approach is not good enough, an opportunistic pervasive computing application can keep dispatching mobile agents that are successively less demanding in terms of specificity. At the beginning, very demanding (specific) agents are dispatched to achieve full functionality. If no hosts for those are found (because required resources are not available), somewhat less demanding agents are dispatched. The process is repeated until the right level of abstraction is achieved.

A high level of abstraction is not necessarily something to be avoided; sometimes it may be used on purpose to refer to great many resources in a concise way. Assume that an event inheritance graph contains two high level events, `HumanEvent` and `MachineEvent`, and that each more specific event is identified with either one or the other. Then, to detect all events generated by a human being, it is sufficient to have a very simple agent with just one handler (for `HumanEvent`) and to dispatch the agent in the broadcast mode. A quite abstract piece of context information ("somebody is at home") can be obtained this way, without any explicit context synthesis.

Another case of using a high level of abstraction on purpose is when the programmer does not care how a given piece of functionality is delivered to the user (e.g., an alert can be displayed on whatever device happens to be available).

A useful feature of domain-oriented virtual machines is that an agent can work on a new node that offers non-generic items unknown at the time the agent was developed. This is the case as long as the new non-generic items are sub-events (sub-instructions) of those used by the agent.

Some of the issues raised in this section have a flavor of object-oriented programming (which also has to do varying levels of abstraction). The unique feature of domain-oriented virtual machines is that domain abstractions can be "embedded" into severely constrained sensor-actuator nodes.

5. Related work

We could find very little work directly related to domain-oriented virtual machines, as presented in this paper. Node-specific virtual machines in pervasive computing are presented in [4]. However, the heterogeneity there has to do with computing power of underlying nodes. In our approach, the network is homogeneous in terms of computing power, but heterogeneous as to available sensors and actuators.

A programming paradigm that fits the node-specific virtual machine approach is the *prototype-based programming*, as referenced by the Self language [5]. It can be characterized as a classless object-oriented programming model. Objects are not instantiated from classes but derived from other objects. Their methods can be freely modified (added, removed from the

object). This is similar to our micro-agents freely composed of instructions and events.

Below we highlight some work that, in our opinion, could be used in opportunistic pervasive computing, but is *not* directly linked to the domain-oriented virtual machines approach. A common trait here is programming for an unknown and changing environment.

One of the problems is how to refer to an unknown number of nodes present in the environment. Naturally, a middleware layer could enable the programmer to "manually" discover the nodes and manage individual node references (which would need to be periodically updated). A much more programmer-friendly approach is discussed in [6] and [7]. In both approaches, special references, each pointing to multiple objects, are used. In [6], they are called *multi-references* (as opposed to *mono-references*). In [7], they are called *omnihandles* (as opposed to *unihandles*). In either approach, the programmer can handle multiple nodes by using a single reference.

Dynamically changing references to resources are the source of another problem: even if the reference has been "acquired," it can quickly become out-dated once the resource disappears. A programming model to facilitate this problem is based on *strong* and *weak* references [6]. Strong references always point to the same object. Weak references can point to different objects during their lifetime and are automatically associated with available objects by the middleware. (Note that a similar mechanism is used in some programming languages, e.g., Java, but its purpose is to help in garbage collection).

Another level of programming abstraction, which in some way can be used in opportunistic pervasive computing, is offered by so-called *macroprogramming* (exemplified by the Kairos system [8] and the Regiment language [9]). Compared to multi-references and omnihandles, the macroprogramming approach goes a step further. The goal is to program the whole network as a single entity. New abstractions, such as *regions*, *streams*, or *areas* have been introduced. They group multiple nodes and sensor readings together.

A holistic approach to programming unknown, changing environments, called ambient-oriented programming (AmOP), is discussed in [10]. AmOP exploits an object-oriented model with non-blocking communications primitives and combines the prototype-based programming with the actor model.

Neighborhood abstractions, which are of great use in opportunistic pervasive computing, have been discussed in [11] and [12]. Both systems use special abstractions of the node's neighborhood, which is defined as nodes sharing some attributes. Once defined, the neighborhood is maintained by the middleware, which will automatically update the list of nodes belonging to the neighborhood.

6. Conclusion

The concept of opportunistic pervasive computing has been briefly described, and the technique of domain-oriented virtual machines has been presented in detail. Possible ways to apply domain-oriented virtual machines to opportunistic pervasive computing have been outlined; we plan to study them further. Clearly, much more work is needed to realize the vision of opportunistic pervasive computing.

7. References

[1] McGee, D.R. and Cohen, P.R.: "Use what you've got: Steps toward opportunistic computing", Technical Report, 2000, School of Science and Engineering, Oregon Health and Science University.

[2] Domaszewicz, J., et al.: "ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-Actuator Networks", Mobile Distributed Computing (MDC'06), 2006, Niagara-Falls, NY, USA.

[3] Domaszewicz, J. and Rój, M.: "Lightweight Ontology-driven Representations in Pervasive Computing", Network Centric Ubiquit. Systems (NCUS'05), 2005, Nagasaki, Japan

[4] Palmer, D.: "A Virtual Machine Generator for Heterogeneous Smart Spaces", Virtual Machine Research and Technology Symposium (VM'04). 2004. San Jose, USA

[5] Ungar, D. and Smith R. B.: "Self: The Power of Simplicity", Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87), 1987, Orlando, USA

[6] Van Cutsem, T., Dedecker, J., Mostinckx, S., and De Meuter, W.: "Abstractions for Context-aware Object References", Building Software for Pervasive Computing, OOPSLA'05 Workshop. 2005. San Diego, CA, USA.

[7] Bischof, H.-P. and Kaminsky, A.: "Many-to-Many Invocation: A new framework for building collaborative applications in ad hoc networks". Ad Hoc Communication and Collaboration in Ubiquitous Computing Environments (CSCW'02), 2002, New Orleans, USA.

[8] Gummadi, R., Gnawali, O., and Govindan, R.: "Macro-programming Wireless Sensor Networks Using Kairos", Distr. Computing in Sensor Systems (DCOSS'05), 2005

[9] Newton, R. and Welsh, M.: "Region Streams: Functional Macroprogramming for Sensor Networks", Data Management for Sensor Networks, 2004, Toronto, Canada.

[10] Dedecker, J., et al.: "Ambient-Oriented Programming", Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05), 2005, San Diego, CA, USA.

[11] Whitehouse, K., Sharp, C., Brewer, E., and Culler, D.: "Hood: A Neighborhood Abstraction for Sensor Networks", Mobile Systems, Applications and Services (MobiSys'04), 2004, Boston, MA, USA.

[12] Welsh, M. and Mainland, G.: Programming sensor networks using abstract regions, Networked Systems Design and Implementation (NSDI'04), 2004, San Francisco, USA.