Multiple Dispatch for Ambient Intelligence

Sebastián González Wolfgang De Meuter Kim Mens

Departement d'Ingeniérie Informatique
Université catholique de Louvain
Belgium

{sgm,km}@info.ucl.ac.be

Programming Technology Lab
Vrije Universiteit Brussel
Belgium

wdmeuter@vub.ac.be

29th May 2005

Abstract

The paper promotes multiple dispatch as an advantageous feature for object technology geared towards Ambient Intelligence. Some of the design decisions and technical challenges that arise are discussed.

1 Introduction

Developing object technology for Ambient Intelligence is not going to be easy. AmI poses hard conditions like frequent disconnection and low-resource computing (just to mention a few), and also it poses hard requirements, like unanticipated context-aware adaptation and unanticipated interaction of applications. We believe that object orientation can provide the right answers, but those answers are certainly not based on the kind of OO technology currently being deployed in industry and mainly being researched in the scientific world (namely class-based, statically typed languages). Looking for better models, we have come across a computation model which combines prototype-based programming and multiple dispatch. While not entirely new, the combination has recently reincarnated into the so-called PMD model [7]. PMD seems to us like a good start for developing AmI-geared object technology. In this paper we briefly present the model (section 2), describe its advantages (section 3) and show the challenges that we think will be faced in its adaptation to AmI (section 4). If short of time, the reader should focus attention on this last part.

2 PMD in a Nutshell

PMD is a computation model based on the prototype-based model of Self [9], thus it is purely object-oriented, classless, with multiple dynamic inheritance and very uniform semantics. The distinguishing characteristic of PMD is that it mixes prototypes with multiple dispatch (hence the acronym PMD). The only current implementation is the Slate programming language.[1] [7] In this short presentation we put the emphasis on the multiple dispatch part, since that is the focus of the paper.

In PMD, a method specifies the kinds of arguments for which its code is designed to work. These specifications are called *argument specialisers*, and arguments with such restrictions are called *specialised arguments*. As an example, consider the following method definition (using Slate syntax, which is Smalltalk-like):

```
a@Set add: element
  [ "implementation here" ]
```

[1] The Cecil language [2] matches very closely PMD, save that it restricts prototype delegation to be fixed at compile time, greatly restricting the benefits of prototypes.

The a@Set notation specialises the a formal parameter on Set, implying that the method is intended to work correctly with any actual argument object that is equal to or a descendant of the Set object.[2] An unspecialised formal argument (one lacking a @ suffix such as the element parameter above) can be considered to be specialised on the most general object, e.g. Object; consequently an unspecialised formal can accept any argument object. Unlike singly dispatched languages, there is no implicit self formal; all formal parameters are listed explicitly.

Methods may be overloaded, i.e., there may be many methods with the same name, as long as the methods with the same name and number of arguments differ in their argument specialisers. In the following example the method as: is overloaded:

```
x@SmallInteger as: i@BigInteger
  [ "implementation A" ]

x@BigInteger as: i@SmallInteger
  [ "implementation B" ]
```

When sending a message of a particular name with a certain number of arguments, the method lookup system will resolve the overloaded methods to a single most-specific applicable method based on the dynamic values of the actual argument objects and the corresponding formal argument specialisers of the methods. In cases where two methods are equally specific, languages differ. Cecil considers the ambiguity an error, while CLOS and Slate choose a method by giving the arguments priority from left to right. In the context of the discussion below we need not concern ourselves with the specific strategy used.

3 Advantages of (P)MD

To the extent of our knowledge, the possibilities of multiple dispatch in a distributed setting are unexplored, let alone the combination of multiple dispatch and prototypes (i.e. PMD) in the context of AmI. A discussion on the advantages of prototypes for AmI was held last year at an ECOOP workshop [4], and it is not our focus here. A good account of the general advantages of prototypes is available in [6]. This section is focused on multiple dispatch, as implanted in the PMD model. The goal is to motivate the use of multiple dispatch in AmI through general arguments and some more more AmI-specific.

3.1 Declarativeness

The PMD model makes programming more declarative. As an example, consider the following excerpt (adapted from the boolean system of Slate):

```
_@True and: _@True [True].
_@Boolean and: _@Boolean [False].
_@False or: _@False [False].
_@Boolean or: _@Boolean [True].
```

The code reads naturally: the conjunction of two True-like objects results in True. The conjunction of any other Boolean results in False. The disjunction operation is implemented in a symmetric way.

Declarativeness implies simplified control flow. In a non-declarative language, the programmer has to manually code the decisions about the run-time type of arguments. In some cases it is possible to use the double dispatch technique to avoid cluttering the code with if statements, but double dispatch adds to the architectural complexity of the application, and it forces the programmer to manually write all the necessary dispatch code.

[2] An object A *descends* from another object B if B is a parent of A in the prototype-based sense, i.e. if B can be found by looking up the inheritance graph starting from A.

3.2 No Receiver-Centricity

Multiple dispatch avoids the weirdness of implementing in the receiver behaviour which does not naturally belong to one particular participant. The typical example are arithmetical operators. In the singly-dispatched case we see code like the following:

```
Integer>>+ b
  self primitiveAdd: b
```

The question arises, why the first operand in the addition operation is doing the job? why not the second one? In single dispatch, the "story" told by methods is written in first person. The code is written in terms of self, which for some situations is natural, but for some others, as illustrated above, is too "egocentric" (receiver-centric). In multiple dispatch the example above is implemented as:

```
a@Integer + b@Integer
  a primitiveAdd: b
```

The method is seen as a collaboration on an equal basis between the method arguments, with no distinguished receiver. The "story" is told from a neutral point of view, analogous to third person. Drawing from the Actors model metaphor [1], a multimethod can be seen as a theatre script with well-defined roles, the argument specialisers. Actual actors fulfil these roles when they come into scene, i.e. when the script is executed. This analogy is not gratuitous: we believe that the Actors model will provide many answers to the challenges posed by AmI [3]. The adaptation of the Actors model to multimethods is one of our main interest foci.

3.3 Generality

Multiple dispatch subsumes procedures and single dispatch. Zero, one, or several of a method's arguments may be specialised, thus enabling multi-methods to emulate normal undispatched procedures and singly-dispatched methods as well as true multi-methods.

3.4 Non-Intrusive Specialisation of Behaviour

In PMD, a given message can initially be implemented with e.g. a single unspecialised procedure and then later be easily extended with several specialised implementations, by just adding new methods. Clients of the original method are unaffected. Specialisation is likely to be used intensively in AmI, since a same system will contain code about many domains, and specialisation is a way of (a) describing specific behaviour for the different "niches" in those domains and (b) avoiding incorrect interactions (i.e. using too general methods, or methods that are not prepared to handle specific cases).

3.5 Multimethods as Fine-Grained Knowledge

In multiple dispatch, the problem of method dispatch can be seen as the question "do I *know how* to perform the requested functionality for the given arguments? and if I do, what is the best (most specific) option I have?". Seen this way, methods effectively represent the know-how a system has. When devices interchange individual methods, they are interchanging knowledge at a very fine-grained level about how to perform tasks in specific contexts (i.e. for specific kinds of arguments). This exchange occurs more naturally if methods are not monopolised by one single object. In multiple dispatch, the unit of deployment of knowledge is not objects but rather individual methods.

3.6 Subjective Dispatch

Multiple dispatch allows the introduction of subjective dispatch, a technique pioneered by the Self extension Us [8], and smoothly incorporated into Slate. In this view of subjective dispatch, a subject is merely an extra implicit participant in the dispatch process. An implicit *layer* argument is prepended to argument lists of message invocations, so that methods can be specialised according to different points of view (i.e. different layers). The point of view, held in the thread or interpreter state of the client, is passed implicitly on each message send, so that the actual implementation chosen by the method dispatch algorithm might not be the same for two different clients (if they hold two different points of view). Subjective dispatch brings in the advantages of subject-oriented programming [5]. The essential characteristic of subject-oriented programming is that different subjects[3] can separately define and operate upon shared objects, without any subject needing to know the details associated with those objects by other subjects. Only object identity is necessarily shared. We think that this *non-interference* property of subject-oriented programming will be of capital importance in AmI, also from a security standpoint.

4 Technical Challenges

Hoping to have convinced the reader of the worthiness of multiple dispatch as realised in the PMD model, we would like to turn attention to implementation issues in a distributed setting. These issues are key in the assessment of the viability of the PMD model for AmI. Our analysis is driven by the different steps involved in method invocation:

1. send, receive and schedule a message for execution
2. lookup the actual method to be executed
3. execute the method

We have identified a number of issues when these steps are considered in an AmI setting, which we describe in this section. The section contains basically guesswork since there is no implementation available. It is our goal for the workshop to gather ideas, remarks and criticism from the participants, in order to obtain in the near future as good a design of distributed PMD as possible, if possible at all. Our first impression is that distributed PMD is not only feasible, but also the analysis below reveals further advantages of multiple dispatch apart from those mentioned in section 3.

4.1 Message sending, receiving and scheduling

Because of the frequency of disconnections, our hypothesis is that AmI object-oriented technology needs to use asynchronous message passing. With asynchronous communication there is no correlation between the time of sending and the time of receiving a message. This decouples the availability of communication partners in time and makes it appropriate in wireless network environments [3]. We are not aware of any OO language supporting multiple dispatch and asynchronous message passing. This is a technical challenge to be explored. Firstly, sent messages do not have a particular sender, because at any given point there is no special "self" object. Secondly, it is not possible to append received messages to the message queue of a receiver, because in multiple dispatch there is no distinguished receiver. Our answer to these two points is that message queues should be global to the VM; the VM schedules the messages by managing a priority queue and decides which message in the queue to process next. A message processing mechanism with priorities allows for real-time constraints to be taken into account, which are also important in AmI.

[3] A *subject* is a collection of state and behaviour specifications reflecting a particular *gestalt* or perception of the world at large, such as is seen by a particular application or tool.

4.2 Method Lookup

Suppose a message "A with: B and: C" is sent in some machine L, where A is an object local to L and B, C are remote objects located at machines M and N respectively. In order to find the applicable methods, it is necessary to find all methods with selector #with:and: whose formal specialisers are compatible with the actuals A, B and C, and then choose the most specific method. The question is, where to search? In machine L, where the message is sent? or in M and N also, since B and C belong there? or in other machines present in the current environment? It might be that the know-how (i.e. an applicable method) is not present in L, M or N, but it is available in some other device. This latter approach (querying the devices present in the environment) corresponds to the idea that the sum of devices is more than each one considered in isolation, i.e. that there is *emerging behaviour* or *synergism* in the union of different devices. This is a very desirable characteristic for AmI. However, searching the whole environment for each message send is too high a cost to pay. A possibility is to incrementally resort to other machines only when applicable methods are not found locally. In a first stage, the machine where the message is sent is queried, then the different machines where the arguments of the message reside, and finally a broadcast query is sent to every device in the current environment. This way, locally available behaviour can be utilised without the additional overhead due to distribution. Most chances are that the majority of the behaviour accessed by applications is available locally (e.g. library code), while the cost of distributed queries is paid only when it is really needed. The worst case is non-understood messages, where the three stages are executed without success.

The 3-stages approach just described is not optimal in every sense. While it tries to avoid the cost of frequent distributed querying, it uses the available knowledge (i.e. functionality) suboptimally. For example, if two applicable methods are available at machines L and M, the one of L will be chosen because it is local, even if the method in M is more specific (thus the technique is "suboptimal" in this sense). There is a variation which could improve the utilisation of distributed functionality by (paradoxically) considering non-functional requirements. Suppose that some kind of meta-data is associated with methods, e.g. estimates of resource consumption (processor, memory) and special-purpose metrics. The method dispatch process can take that meta-data into consideration and decide to proceed with a distributed query even if it already found a local applicable method. Furthermore, it might be that the local method *is* the most specific, but that given the current machine execution conditions (e.g. too little memory available) it cannot be invoked. This adaptive 3-stages approach models the real world: an organisation (or person) does not utilise its local know-how only if it believes that it is insufficient for the task at hand. Then more appropriate knowledge is consulted externally.

Finally, we note that security concerns should be born in mind while thinking about method lookup mechanisms: sometimes a device might hold an applicable method but the method should be concealed from other devices. We think that the solution lies in subject-oriented programming (see section 3.6), where method visibility is determined by the subject (i.e. groups of methods are accessible from certain client perspectives only).

4.3 Method execution

Assume that a most "appropriate" method, whether most specific or not, has been found. The next question is, where should it be executed? If it was found locally, it might seem natural to execute its code in the local machine. If it was found in a machine hosting one of the message arguments (like M or N in the previous example), it could be executed there. If it is available elsewhere, should the device hosting the method pay the price of executing it, given that it is an outsider to the interaction taking place? or should the method be transmitted to some of the involved parties? There is a plethora of possibilities. A device with enough storage space and processing power (like a static computer making part of the network infrastructure) might very well execute the method even if it is not involved in the interaction. On the other hand, transmitting the method to another device has the advantage that knowledge (i.e. functionality) is gained by that device. In case of disconnection from the network the knowledge will still be available to the device. In analogy

with real world organisations, the situation resembles the decision between doing outsourcing or training personnel.

5 Conclusion

We have proposed a language feature, multiple dispatch, from which AmI could benefit. Since multiple dispatch has not been explored in a distributed setting, nor under the optics of context-aware application adaptation and interaction, we have hinted at some initial intuitive questions and possible answers. It is in our near future plans to deploy a language with multiple dispatch on a mobile device (a Smart Phone) and gather experience about the language pragmatics, and shortcomings and potentialities of the approach. Our goal for the workshop is to get further ideas, insight, remarks and criticism from the participants, and also to know about related work, so as to assess the viability of the approach.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[3] J. Dedecker and W. Van Belle. Actors for mobile ad-hoc networks. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, volume 3207, pages 482–494. Springer-Verlag, 2004.

[4] S. González, W. De Meuter, P. Constanza, S. Ducasse, R. Gabriel, and T. D'Hondt. Report from the ECOOP 2004 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity. In J. Malenfant and B. M. Østvold, editors, *ECOOP 2004 Workshop Reader*, volume 3344 of *Lecture Notes in Computer Science*, pages 49–61. Springer-Verlag, 2004.

[5] W. H. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA '93: Proceedings of the 8th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM Press.

[6] J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications.* Springer-Verlag, 1999.

[7] L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *Proceedings ECOOP 2005 (European Conference on Object-Oriented Programming)*, 2005. To appear.

[8] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems (TAPOS)*, 2(3):161–178, 1996.

[9] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.