Multimethods for Context-Awareness*

Sebastián González[1], Stijn Mostinckx[2],
Pascal Costanza[2], Kim Mens[1], and Wolfgang De Meuter[2]

[1]Département d'Ingeniérie Informatique
Université catholique de Louvain, Belgium
{s.gonzalez,kim.mens}@uclouvain.be

[2]Programming Technology Lab
Vrije Universiteit Brussel, Belgium
{smostinc,pcostanz,wdmeuter}@vub.ac.be

Abstract The increasing availability of mobile devices and the wireless networks that connect them sets the stage for the introduction of context-aware applications. Current context-aware applications typically achieve adaptation to context by hard-coding decisions. In this paper we present an alternative approach based on classless objects in combination with distributed multimethods to develop more flexible context-aware applications.

1 Motivation

The introduction of mobile devices equipped with wireless network provisions, allows for present-day applications to become aware of their environment and to interact with it. Unfortunately, the incorporation of context information into running applications is currently often achieved using *ad hoc* mechanisms. To allow for an application to behave differently in a given context, this context-specific behaviour is typically hard-wired in the application under the form of `if`-statements scattered in method bodies or by using design patterns [7] (*e.g.* the Decorator, State and Strategy patterns). As an alternative solution, in this paper we explore the PMD object model [10] – which features prototypes and multiple dispatch – in the context of distributed systems. Our distributed extension provides a structured mechanism to deal with contextual information in an extensible, flexible and high-level manner.

Context-aware distributed applications rely on a context architecture that represents the input from sensors (and possibly other applications) in a way that is meaningful to the application. The architecture used in this paper is akin to that of the Context Toolkit [6], using objects to aggregate the context derived from different (interpretations of) sensor data. The chief difference with our approach lies in the way the context architecture will be employed by the applications that rely on it. To avoid hard-wiring context-related behaviour inside the application, the aggregator directly influences the dispatch of methods. In other words, the programming model provides direct support for *Context-Oriented Programming* [4].

* S. González is funded by the Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture (FRIA, Belgium). S. Mostinckx is funded by the *Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT, Belgium).*

2 Distributed Prototypes

The PMD model – which is extended in this paper to develop context-aware distributed applications – relies on a flexible object model inspired on the programming language Self [13]. In the PMD model, objects are entirely self-sufficient such that they can properly function without requiring a class definition. Hence, each object contains its own variable and method slots. New objects are created by cloning existing *prototypes*, objects that act as representative examples of domain entities. However, prototypes do not have a special status in the language, as any object can be cloned and therefore serve as a prototype.

Objects in a prototype-based language can extend other objects by *delegating* to them [9]. This allows the extending object (traditionally termed *child* object) to reuse both the methods and the variables from the extended object (traditionally termed *parent* object). In Self, the relation between child and parent is established using special *parent slots*. Since parent slots may change at run-time, this extension scheme is called *dynamic inheritance*, in contrast with standard inheritance in class-based languages, where the inheritance relationships are defined statically.

Currently, we are developing a distributed version of PMD which allows for objects to be distributed across different hosts. We ensure that all (transitively reachable) parents of an object are collocated with the child. Although it is theoretically possible to distribute the delegation relationship across different machines [12], it remains unclear how a multimethod dispatching algorithm should respond when a particular parent is not available.

3 Distributed Multimethods

The prototype-based object model presented in the previous section is complemented with *multimethods* to avoid hard-wired "dispatch" code on the provided context information. The introduction of multimethods – methods which specify the combination of arguments for which they are intended to work – allow specifying the influence of context information in a declarative and extensible fashion. In multi-dispatched languages such as CLOS, Cecil, Dylan, Slate and MultiJava, messages are not sent to a single receiver. Rather, the implementation which is best suited for the particular combination of message arguments is searched for. A multimethod declares which set of arguments it expects by using *argument specialisers*. The following example defines a simple multimethod describing a default discovery protocol for bluetooth devices (in a Smalltalk-like syntax):

```
personal@BTDevice discovers: other@BTDevice
  [ personal show: (other name). ]
```

The personal@BTDevice notation specialises the personal formal parameter on BTDevice, implying that the method is intended to work correctly with any actual argument which extends or is a clone of BTDevice. An unspecialised formal argument (one lacking an @ suffix) can accept any argument object. Unlike singly

dispatched languages, there is no implicit receiver, and thus no implicit this (or self) formal parameter.

The declarative power of multimethods stems from the fact that they can 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. Since the method that is chosen depends on the dynamic type of all passed arguments, patterns such as Visitor [7] and Double Dispatch [8] can be specified directly in the language [2]. The programmer can add special cases by simply adding a new multimethod specialised on the right set of arguments. The example below prescribes that objects representing paired bluetooth devices, which naturally extend BTDevice objects, should attempt to synchronise upon discovering one another:

```
personal@PairedDevice discovers: other@PairedDevice
  [ (personal pairedDevices contains: other) ifTrue:
      [other synchronize: personal]. ]
```

Multimethods can be considered a dynamic alternative to Java's overloading mechanism, such that the method dispatch always takes into account the dynamic type of all arguments it is being passed. This multiple dispatch mechanism provides a flexible and declarative mechanism to describe the interaction of different objects.

To allow for this mechanism to be employed in a distributed setting, our system reuses Slate's roles [10] to internalise multimethods into objects. Roles are tuples (s, i, m) where s is the method selector (the method name), i is the position at which the object is used as an argument specialiser, and m the multimethod itself. Upon dispatching a message, the roles of argument objects are searched for applicable multimethods, without resorting to a global method table. In the remainder of this section, we illustrate how this distributed handling of multimethods can be extended to incorporate context information into the dispatch process.

Subjective Dispatch for Context Adaptation

Section 1 has already suggested that the influence of context on a running system is achieved by having the context aggregator influence the method dispatch process. To this end, a *perspective* object is implicitly prepended to the argument list of method definitions and message sends, so that methods are applicable only if the current perspective is equal to or extends the perspective where the method was defined. By means of such an implicit perspective argument, the PMD multiple-dispatch mechanism is exploited to obtain *subjective behaviour* [11], behaviour that depends on the perspective from which it is seen. Thanks to dynamic inheritance, the aggregation of perspective objects can be altered at run-time to reflect changes in the context. Context-specific multimethods can be defined as follows:

```
[ personal@BTDevice discover: other@BTDevice
  [ personal setIcon: getBusyIcon.
    resend. ]
] seenFrom: userOccupied
```

In the above example the discover: multimethod is refined to attach context information – showing that the user of this device is occupied and should not be disturbed – to the visualisation of the personal device. The seenFrom: call executes the code block within the userOccupied context. To invoke the method defined above, the call would thus have to originate from a context that extends or is equal to userOccupied.

Our distributed PMD system exhibits the characteristics of *Context-Oriented Programming* [4]: 1) the influence of context is described declaratively, 2) context information is passed implicitly through the dynamic invocation chain, 3) layer activations (i.e. seenFrom messages) can be nested, and 4) perspective objects are aligned with threads, such that different layers may be activated in different threads. The main differences between ContextL (the implementation of Context-Oriented Programming in Common Lisp) and our distributed version of PMD are that we employ a prototype-based object model, such that layers can be switched on dynamically by manipulating the delegation hierarchy of the context object. A final important difference with ContextL is that the perspective is reified in the system as an ordinary object, which can be handed out to other threads. Therefore one thread may in fact add a layer to the context of another thread (e.g. an agenda may use this mechanism to notify the discovery application that the user currently is occupied).

4 Work in progress

At present, the programming experience we have with the distributed PMD model remains limited. Although promising so far, the model currently raises more questions than it answers.

4.1 Distribution Semantics

To the extent of our knowledge, the possibilities of multiple dispatch in a distributed setting – let aside a mobile computing setting – are largely unexplored. The approach documented here illustrates some of the interesting properties that a system featuring distributed multimethods may offer. On the other hand, some questions regarding how to distribute multimethods remain to be answered. We list some of these issues (as well as how they are handled in distributed PMD today).

At present, our distributed PMD model uses a thread-based concurrency model, where multimethods are always executed by the same thread that invoked them. This semantics prohibit the scheduling of remote execution of multimethods and thus introduce a large network overhead when all arguments to an invocation reside on another machine.

An alternative that aligns well with the notion of self-sufficient objects is to employ an actor-based concurrency model [1], where objects are equipped with their own thread. However, since a multi-dispatched language does not distinguish a single receiver, this approach introduces problems of its own: namely how to select the actor to execute the message from the set of all suitable candidates. Such a selection of a particular active object for execution of the message can be based on load-balancing criteria, on the reliability of the connection to remote devices, etc. Furthermore, it is not sufficient to select an active object to run the method, since when multiple active objects need to collaborate, the multimethod should act as a mediating entity for the different asynchronous calls inside the body of the multimethod.

4.2 Object Encapsulation

Traditionally multimethods are criticised since they allow access to the state of all objects on which they are specialised. Conceptually, this is defendable since multimethods should be considered internal to each of the objects on which their arguments are specialised [2]. However, without any additional mechanism in place, this design implies that introducing a multimethod specialised on an object would negate its encapsulation. A similar critique can be voiced regarding the malleable prototype-based object model that is employed in distributed PMD. Since objects can simply designate any referable object as their parent (using special-purpose parent slots) any object may therefore gain access to the `protected` slots of an object.

We seek to replace the traditional module-based approach [3] to retain encapsulation in the presence of multimethods by a system that also tackles the unrestricted extension in prototype-based systems. However, it is critical that, while ensuring encapsulation, the flexibility of the PMD model is maintained. We aim to achieve this by translating the different language operators on objects (e.g. extension, specialisation of a multimethod) to patterns of message sending. In previous work, we have successfully applied this technique to obtain an encapsulation mechanism for singly-dispatched prototype-based languages [5].

5 Conclusion

In this short paper we have introduced a distributed object model model based on prototypes and multimethods. Subsequently, a number of reasons were presented to illustrate how this particular object model can aid in the development of context-aware applications.

Firstly, multimethods allow declarative specifications of the interaction between objects. If new interaction cases are to be handled, they should not be encoded using e.g. `if` statements; rather, they can be handled by adding new multimethods, so that previously deployed methods remain unchanged.

Secondly, the context object that aggregates the underlying context information is a malleable prototype that can adapt its delegation hierarchy at runtime to reflect contextual changes in the environment.

Thirdly, the use of subjective dispatch allows one to dynamically adapt the behaviour of the system (i.e. the choice of methods which actually get executed) according to the current configuration of context objects. The need to hard-code context-aware behaviour in the body of methods is thereby nullified.

Finally, the use of multimethods does not necessitate a distributed infrastructure that stores all multimethods. Instead, multimethods can be made internal to the objects on which they specialise, which guarantees that dispatching can be performed using local information only.

References

1. G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. C. Chambers. Object-oriented multi-methods in cecil. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.
3. C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17(6):805–843, 1995.
4. P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming. In *Dynamic Languages Symposium at OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.
5. W. De Meuter, E. Tanter, S. Mostinckx, T. Van Cutsem, and J. Dedecker. Flexible object encapsulation for Ambient-Oriented Programming. In *Dynamic Languages Symposium at OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.
6. A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16, 2001.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 347–349, New York, NY, USA, 1986. ACM Press.
9. H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.
10. L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. *Lecture Notes in Computer Science*, 3586:312–336, 2005.
11. 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.
12. R. Tolksdorf and K. Knubben. dself - a distributed self. KIT-Report 144, TU Berlin, 2001.
13. 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.