Handbook of Research on Mobile Software Engineering:

Design, Implementation, and Emergent Applications

Paulo Alencar
University of Waterloo, Canada

Donald Cowan
University of Waterloo, Canada

Volume I

ENGINEERING
SCIENCE REFERENCE

Managing Director:	Lindsay Johnston
Senior Editorial Director:	Heather A. Probst
Book Production Manager:	Sean Woznicki
Development Manager:	Joel Gamon
Development Editor:	Hannah Abelbeck
Acquisitions Editor:	Erika Gallagher
Typesetter:	Jennifer Romanchak
Cover Design:	Nick Newcomer, Lisandro Gonzalez

Published in the United States of America by
Engineering Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Handbook of research on mobile software engineering: design, implementation, and emergent applications / Paulo Alencar and Donald Cowan, editors.
p. cm.
Includes bibliographical references and index.
Summary: “This book highlights state-of-the-art research concerning the key issues surrounding current and future challenges associated with the software engineering of mobile systems and related emergent applications”--Provided by publisher.
ISBN 978-1-61520-655-1 (hardcover) -- ISBN 978-1-61520-656-8 (ebook) -- ISBN 978-1-4666-1612-7 (print & perpetual access) 1. Mobile computing--Handbooks, manuals, etc. 2. Mobile computing--Research--Handbooks, manuals, etc. 3. Software engineering--Handbooks, manuals, etc. I. Alencar, Paulo, 1957- II. Cowan, D. D. (Donald D.), 1938-
QA76.59.H34 2012
004--dc23
2012002505

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 10
Language Engineering for Mobile Software

Engineer Bainomugisha
Vrije Universiteit Brussel, Belgium

Alfredo Cádiz
Université catholique de Louvain, Belgium

Pascal Costanza
Vrije Universiteit Brussel, Belgium

Wolfgang De Meuter
Vrije Universiteit Brussel, Belgium

Sebastián González
Université catholique de Louvain, Belgium

Kim Mens
Université catholique de Louvain, Belgium

Jorge Vallejos
Vrije Universiteit Brussel, Belgium

Tom Van Cutsem
Vrije Universiteit Brussel, Belgium

ABSTRACT

Mobile systems offer the possibility of delivering software services that tightly match user needs, thanks to their availability right at the moment and place where they are needed, and their ability to take advantage of local resources and self-adapt to their environment of use. Alas, writing software for mobile systems is not an easy endeavour. Mobile software construction imposes a number challenges that render existing programming technology insufficient to write such software conveniently. To improve this situation, the authors have taken a language engineering approach. In this chapter they identify the main challenges encountered in mobile software construction and the requirements that rise in the design of programming languages. By way of illustration, the authors present the result of their language engineering experiments —four programming models to ease the construction of software that can cope gracefully with the challenges brought about by mobility.

INTRODUCTION

Hardware technology is ripe for the construction of mobile applications that run ubiquitously, foster peer-to-peer communication and seamlessly adapt their services to changing environments. With respect to these contemporary hardware phenomena, we observe that programming technology is lagging behind in enabling the construction of applications that naturally support concurrency, decentralised and unreliable distribution, context-awareness and dynamic self-adaptability. These

DOI: 10.4018/978-1-61520-655-1.ch010

aspects are key to fully exploiting the potential that will set mobile systems apart from traditional desktop and server systems. To consolidate the emerging field of mobile software engineering, new programming languages, methodologies and development tools need to be co-designed. In this chapter we concentrate on one piece of the puzzle, namely language engineering for mobile applications. We give a rendition of the experience we have gathered so far in shaping the design space of mobile languages, and describe possible designs along our two principal axes of expertise: Ambient-Oriented Programming (AmOP) (Dedecker et al., 2006; Van Cutsem et al., 2007; Vallejos et al., 2009) and Context-Oriented Programming (COP) (Costanza, 2008; González et al., 2008). AmOP languages ease the construction of software deployed in mobile ad hoc networks because they provide dedicated language features that help the programmer in dealing with the hardware characteristics inherent to those networks. Example features include connections that tolerate temporary network failures (required because of the volatility of wireless network links) and primitives to spontaneously discover nearby services in the local network. COP languages ease the construction of adaptive applications by providing features to support context-dependent behavioural variations. COP treats context explicitly, and provides mechanisms to dynamically adapt application behaviour in reaction to changes in context at run time. The context encompasses all computationally accessible information that describes the current situation, such as device location, battery charge level, and user activity. We use the most representative languages we have developed –AmbientTalk, Ambience, ContextL and Lambic– as case studies on language design, highlighting their strong points and discussing their pitfalls by way of illustration.

CHALLENGES IN MOBILE LANGUAGE ENGINEERING

The hardware properties of the devices constituting a mobile network engender a number of phenomena that have to be dealt with when constructing mobile software. Next we summarise these phenomena, which are inherent to mobile networks and which shape the design space of mobile programming languages.

Volatile Connections

Mobile devices featuring wireless connectivity possess only a limited communication range, such that two communicating devices may move out of earshot unannounced. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Quite often, such transient network partitions should not affect an application, allowing both parties to continue their collaboration where they left off. These more frequent transient disconnections expose applications to a much higher rate of partial failure than that for which most distributed languages have been designed. In mobile networks, disconnections become so omnipresent that they should be considered the rule, rather than an exceptional case.

Zero Infrastructure

In a mobile network, devices that offer services spontaneously join and leave the network. Moreover, a mobile ad hoc network is often not manually administered. As a result, in contrast to stationary networks where applications usually know where to find collaborating services via URLs or similar designators, applications in mobile networks have to find their required services dynamically in the environment. Services have to be discovered on proximate devices, possibly without the help of shared infrastructure. This lack of infrastructure requires a peer-to-peer communication model, where services can be directly advertised and discovered.

Further, remote references to services may become unavailable as the mobile device user roams around; lost references to particular kinds of services should be regained when available. This is in contrast with stationary networks in which references to remote resources are obtained based on the explicit knowledge of the availability of the resource, and are relatively stable. In the context of mobile networks, resources are said to be *ambient.*

Dynamic Execution Context

Due to mobility, the execution context of applications changes constantly, both at physical and logical levels. Contextual information plays an increasingly important role in mobile applications, ranging from those that are location-based to those that are situation-dependent or even deeply personalised. All these context properties that can influence the behaviour of applications are much more variable than in stationary networks.

Run-Time Behavioural Adaptation

Mobile platforms heighten the expectation that applications running on them will turn from mere isolated programs to smart software that can interact with its environment. Thanks to real-time availability of information coming from their physical and logical environment, mobile systems have the potential to adapt swiftly to changing running conditions. Mobile systems should be aware of their execution context and should adapt dynamically and autonomously to such context so that they can provide services that fulfil user needs to the best extent possible.

To the best of our knowledge, no programming language has been designed that deals with the characteristics of mobile systems just described. Languages like Emerald (Jul et al., 1987) and Obliq (Cardelli, 1995) are based on synchronous communication, which is irreconcilable with the connection volatility and the zero infrastructure characteristics. Languages like ABCL/f (Yonezawa et al., 1995) and Argus (Liskov and Shrira, 1988) that are based on futures (Halstead, 1985) partially solve this issue, but their objects block when accessing unresolved futures. Other languages based on the actor model, such as Janus (Kahn and Saraswat, 1990), Salsa (Varela and Agha, 2001) and E (Miller et al., 2005) use pure asynchronous communication. However, these languages offer no support to discover ambient resources or to coordinate interactions among autonomous computing units in the face of volatile connections.

Languages that treat procedures or methods as first-class values allow programs to be structured in a way that they can exhibit different behaviour under different circumstances. In pure functional programming languages like Haskell (Hudak et al., 1992), they can be passed as parameters to higher-order functions, and in languages like Scheme (Dybvig, 1996) or ML (Ullman, 1994), they can be assigned to variables which can be side-effected later on to change the behaviour of a program. However, the availability of first-class procedures alone does not offer sufficient means to coordinate changes of several collaborating procedures. Piccola (Achermann et al., 1999) is an experimental language for specifying applications as compositions of software components. A key feature of Piccola is the *form* –a first-class environment which is used to model, amongst others, objects, components, modules and dynamic contexts (Achermann and Nierstrasz, 2000). Although environments can be manipulated in Piccola, expressions are statically bound to the environment they are evaluated in, so dynamic activation is not supported directly.

An alternative to programming languages is middleware. Over the past few years, many middleware platforms to support mobile computing have been proposed (Mascolo et al., 2002). In RPC-based middleware like Rover toolkit (Joseph et al., 1997) attempt to tackle the issue of connection volatility by supporting temporary queuing of RPCs. However, these approaches remain variations of synchronous communication and are thus irreconcilable with the autonomy and connection volatility phenomena. Communication by means of shared *tuple spaces*, as originally proposed in Linda (Gelernter, 1985), has proven to be particularly good communication model for mobile networks. This is witnessed by middleware such as LIME and TOTA, which are based on distributed variants of the original, shared memory tuple space model. Even though tuple spaces are an interesting communication paradigm for mobile computing, they do not integrate well with the object-oriented paradigm because communication is achieved by placing data in a tuple space as opposed to sending messages to objects. Another kind of existing middleware is publish-subscribe middleware that adapts the publish-subscribe paradigm (Eugster et al., 2003) to cope with the characteristics of mobile computing (Cugola and Arno, 2002; Caporuscio et al., 2003). Such middleware allows asynchronous communication, but has the disadvantage of requiring manual callbacks to handle communication results, which severely clutters object-oriented code.

REQUIREMENTS FOR MOBILE COMPUTING LANGUAGES

This section presents main requirements for the design of mobile computing languages. These requirements are derived from the challenges described in the previous section.

Non-Blocking Communication

In a mobile computing language, all distributed communication should be non-blocking, i.e. asynchronous. The main reason behind this strict asynchrony is that communicating parties remain loosely coupled. It is this loose coupling that significantly reduces the impact of volatile connections on a distributed application. With respect to communication, two degrees of coupling between communicating parties can be distinguished:

- **Decoupling in time.** The communicating parties do not need to be online at the same time (Eugster et al., 2003). Decoupling in time implies that a sender may send a message to a recipient that is offline, and a recipient may receive and process a message from a sender that is offline. This makes it possible for communicating parties to interact across volatile connections. Decoupling in time is directly inspired by the need to deal with the intermittent disconnections inherent to mobile ad hoc networks.
- **Synchronisation decoupling.** The control flow of communicating parties is not blocked upon sending or receiving (Eugster et al., 2003). Synchronisation decoupling implies that a sending party can employ a form of asynchronous message passing, such that the act of message sending becomes decoupled from the act of message transmission. Likewise, allowing recipient parties to process messages asynchronously decouples the act of message reception from the act of message processing. Message transmission and reception require a connection between sender and receiver, but message sending and processing can be decoupled, allowing commu-

nicating parties to abstract over the availability of the other party. This requirement is again directly derived from the volatile connections phenomenon in mobile networks. It allows parties to perform useful work while being disconnected.

Ambient Acquaintance Management

A mobile computing language should have built-in support for ambient acquaintance management: the discovery and management of nearby devices and their hosted services. However, the way in which communicating parties can discover one another reveals yet another degree of coupling with important repercussions in mobile ad hoc networks:

- **Decoupling in space.** The communicating parties do not need to know each other beforehand (Eugster et al., 2003), not their address nor their location. However, this means that communicating parties must rely on some mechanism other than precise addresses or URLs to get to know one another. Decoupling in space is an important property in mobile ad hoc networks because these networks have a minimum of shared infrastructure, making reliance on servers to mediate collaborations impractical. Ambient acquaintance management implies more than simply the discovery of new parties. It also implies that communicating parties must be able to keep an updated view of which participants are connected or disconnected.

Runtime Context-Dependent Behavioural Adaptations

A mobile computing language should provide support to dynamically adapt system behaviour to the current context of use. Any information that is computationally accessible may form part of the context upon which system behavioural variations depend. We identify the following essential language properties to support run-time behavioural system adaptation:

- **A means to specify behavioural variations.** Variations typically consist of new or modified behaviour, but may also comprise removed behaviour. They can be expressed as partial definitions of modules in the underlying programming model such as procedures or classes, with complete definitions representing just a special case. Variations may also be expressed as edits, wrappers, or even general refactorings.
- **A means to group variations into entities.** Entities group related context-dependent behavioural variations. These entities are first-class in that they can be explicitly referred to in the underlying programming model. Entities are composed in reaction to contextual information. Based on information available in the current execution context, specific program entities may be activated or deactivated.
- **Dynamic activation and deactivation of entities based on context.** Entities aggregating context-dependent behavioural variations can be activated and deactivated dynamically at runtime. Code can decide to enable or disable entities of aggregate behavioural variations based on the current context.
- **A means to explicitly and dynamically control the scope of entities.** The scope within which entities are activated or deactivated can be controlled explicitly. The same variations may be simultaneously active or not within different scopes of the same running application.

The first two features just presented (i.e. *non-blocking communication* and *ambient acquaintance management*) are designed with the *volatile* *connections* and *zero infrastructure* phenomena in mind, respectively. We will henceforth refer to programming languages that adhere to these features as Ambient-Oriented Programming (AmOP) languages. AmOP languages incorporate transient disconnections and evolving acquaintance relationships in the heart of their computation model. The *run-time activation and deactivation of context-dependent behaviour* language features are designed for the *changing execution context* and runtime *behavioural adaptation* phenomena. We will refer to the programming languages that adhere to these features as Context-Oriented Programming (COP) languages. COP languages deal with context explicitly and make it accessible and manipulable by software. In COP, programs can be partitioned into behavioural variations that can be freely activated and combined at runtime with well-defined scopes.

PROGRAMMING LANGUAGES FOR MOBILE COMPUTING

In this section we present a number of programming language approaches we have developed to tackle the challenges of mobile networks. We briefly introduce these approaches and explain how they address the requirements put forward in the previous section. We illustrate the solutions provided by each language by using code snippets from the implementation of two common examples in mobile computing: an instant messenger and a context-aware mobile phone.

AmbientTalk

AmbientTalk (Van Cutsem et al., 2007) is an object-oriented programming language specially designed to satisfy the AmOP characteristics presented in the previous section, non-blocking communication and ambient acquaintance management. This language features a concurrency and distribution model based on the communicating event loops model of the E programming language (Miller et al., 2005), which is itself an adaptation of the well-known actor model (Agha, 1986).

Non-Bocking Communication

In AmbientTalk, actors spawn concurrency: one AmbientTalk virtual machine may host multiple actors which execute concurrently. Actors define boundaries of concurrent execution around groups of objects. Two objects owned by the same actor can communicate synchronously, by means of traditional message passing. However, objects may refer to objects owned by other actors. Object references that span different actor boundaries are named *far references* and only allow asynchronous access to the referenced object. Any message sent to a receiver object via a far reference is enqueued in the mailbox of the actor that owns the receiver object and processed by the owner itself. Actors are event loops: they take messages one by one (i.e. sequentially) from their mailbox and dispatch them to the receiver object by invoking its appropriate method.

AmbientTalk allows asynchronous message sends to return values by means of *futures* (Halstead, 1985). A future is a placeholder for the return value of an asynchronous message send. Once the return value is computed, it replaces the future object; the future is then said to be resolved with the value. To make the discussion more concrete, consider the following example. Assume messenger represents a far reference to a remote object that represents an instant messenger application (further explained at the end of this section). The following code shows how to query this instant messenger for its user's username:

```
def f := messenger<-getName();
when: f becomes: { |val| display(val) }
```

The <- operator denotes an asynchronous send of the message getName to the remote messenger object. This operation returns a future f. In AmbientTalk, an object may react to a future becoming resolved by registering an observer (a closure) that will be called with the resolved value (val) when the future has become resolved. To be able to respond to a getName message, it suffices for the messenger object to define a getName method as follows:

```
def createMessenger(name) {
   object: {
    def getName() { name }
  }
}
```

Note that the return value of the getName method is used to resolve the future that was created as a result of the messenger<-getName() message send.

Ambient Acquaintance Management

AmbientTalk employs a publish/subscribe service discovery protocol. A publication corresponds to exporting an object by means of a type tag. The type tag serves as a topic known to both publishers and subscribers (Eugster et al., 2003). A subscription takes the form of the registration of an event handler on a type tag, which is triggered whenever an object exported under that tag has become available in the ad hoc network. In the instant messenger example, the user can discover other messenger applications available in the surroundings as follows:

```
whenever: IM discovered: { |messenger|
    def fut:= messenger<-getName();
    // notify user of new buddy
};
```

The whenever:discovered: function takes as arguments a type tag and a closure that serves as an event handler. Whenever an actor is encountered in the ad hoc network that exports a matching object, the closure is scheduled for execution in the message queue of the owning actor. An object matches if its exported type tag is a subtype of the type tag argument of whenever:discovered:. The messenger parameter of the closure is bound to a far reference to the exported item object of another actor. The closure can then start sending asynchronous messages via this far reference to communicate with the remote object. Similar to the export:as: function, the discovery mechanism returns an object whose cancel() method cancels the registration of the closure.

In AmbientTalk, objects can acquire far references to objects by means of parameter-passing or return values from inter-actor message sends. Additionally, an actor can explicitly export objects to make their services available to remote actors and their objects. Service objects are exported by means of a type tag. Type tags are a lightweight classification mechanism, used to categorise objects explicitly by means of a nominal type. One use of type tags in AmbientTalk is to provide an intensional description of what kinds of services an object provides to remote objects. In AmbientTalk, a type tag can be a subtype of one or more other type tags, and one object may be tagged with multiple type tags. Although type tags are not used for static type checking, they are best compared with empty Java interface types, like the typical "marker" interfaces used to merely tag objects (e.g. java.io.Serializable and java.lang.Cloneable). In the instant messenger example, the user can announce the presence of the messenger application on the network by means of IM type tag as follows:

```
deftype IM;
def createIM() {
   def pub:= export: self as: IM;
   // this object can be used to
      cancel the advertisement
   pub;
}
```

From the moment an object is exported, it is discoverable by objects owned by other actors by means of its associated type tag. The export:as: function returns an object which can be used to take the exported object offline again, by invoking pub.cancel().

Last but not least, by admitting far references to cross virtual machine boundaries, we must specify their semantics in the face of partial failures. AmbientTalk's far references are by default resilient to network disconnections. When a network failure occurs, a far reference to a disconnected object starts buffering all messages sent to it. When the network partition is restored at a later point in time, the far reference flushes all accumulated messages to the remote object in the same order as they were originally sent. Hence, messages sent to far references are never lost, regardless of the internal connection state of the reference. Making far references resilient to network failures by default is one of the key design decisions that make AmbientTalk's distribution model suitable for mobile ad hoc networks, because temporary network failures have no immediate impact on the application's control flow. Far references have been intentionally made resilient to transient partial failures. This behaviour is desirable in mobile networks because temporary network partitions can provoke many partial failures.

Lambic

Lambic (Vallejos et al., 2009) is an actor-based extension to Common Lisp also designed for the AmOP paradigm. Lambic features a variation of the AmbientTalk's communicating event loops model that reconciles event-driven programming with the generic function invocation style of Common Lisp (object-oriented programs are written in terms of function invocations rather than messages exchanged between objects). Similar to AmbientTalk, Lambic takes the requirements of non-blocking communication and ambient acquaintance management into account. The main property of this language is that the event loops model is integrated in the method execution process. As such, Lambic enables programs to be written in a sequential style and with a uniform syntax for local and distributed computations, while still providing event-driven program execution and support to deal with mobile computing issues such as connection volatility and zero infrastructure.

Non-Blocking Communication

In Lambic (as in AmbientTalk), actors are containers defining boundaries of concurrent execution for a group of objects. Event notifications are modelled as asynchronous generic function invocations, which are sequentially processed by the actor's event loop, dispatching to the appropriate generic functions. Events are then handled by the corresponding methods in the generic function. Standard (synchronous) Common Lisp generic function invocations are allowed only if they occur within the actor that owns all the objects used as arguments. Inter-actor computations are possible by means of asynchronous generic function invocations. A generic function is asynchronously invoked by designating an actor as the responsible for its processing. This results in scheduling the function invocation in the event queue of the actor. The following expression illustrates an asynchronous function call in Lambic:

```
(in-actor-of messenger (get-username
messenger))
```

In Lambic, neither actors nor objects can receive messages directly. In order to select the actor that should process an asynchronous function invocation, a programmer has to supply a reference to an object contained in the targeted actor. Thus, an asynchronous function invocation can be read as "process this function invocation in the actor of this object". The example above corresponds to the translation of the AmbientTalk message that queries for the name of the instant messenger's user, described in the previous section. In this asynchronous function invocation, messenger is a far reference to the remote object representing an instant messenger application. By passing this far reference as the first argument to the in-actor-of form, we ensure that the invocation to the get-username accessor method is processed by the actor that contains the remote object.

By default, an asynchronous generic function invocation returns a future as result whose result can be handled by means of an observer similar to the when:becomes: construct in AmbientTalk:

```
(when-resolved (in-actor-of messenger
(get-username messenger))
    (lambda (name)
        (display "Buddy name: " name)))
```

Lambic integrates the features of the communicating events loop model into the method execution process, which makes it possible for concurrent and distributed programs with asynchronous style (e.g. when-resolved) and non-uniform syntax (e.g. in-actor-of) to be turned back into a sequential style with uniform syntax. This is ensured by two design principles. First, in Lambic there is no syntactic distinction to indicate whether a method invocation should be processed synchronously or asynchronously. This is an implicit decision that is based on the location of the receiver object. Invoking a method on a local object leads to a standard (synchronous) method execution, whereas invoking a method on a remote object leads to an asynchronous method execution at the object's remote location. Second, method invocations immediately return a future as result, which can be passed as argument to other invocations. As such, methods can be defined using a sequential style. No special construct is required to receive results of asynchronous remote invocations. Standard programming structures (e.g. control and conditional expressions) preserve their sequential semantics even if they are used in combination with futures. An invoked method is executed only when all the argument futures are resolved. The future created by a method invocation is asynchronously resolved with the result of the method execution.

Asynchronous function invocations follow the same syntactic pattern of Common Lisp synchronous invocations. For instance, the asynchronous method invocation described above can be replaced by:

```
(get-username messenger)
```

In this case, messenger is both the receiver argument and the indicator of the actor in which the method should be processed. The remote invocation of get-username implicitly returns a future and no special callback is required to receive the results Thus, the example of the when-resolved construct described previously can be replaced by the following nested expression:

```
(display "Buddy name: " (get-username
messenger))
```

Ambient Acquaintance Management

Lambic presents the same language support for dealing with the issues of distribution (discovery and failure handling) that the one of AmbientTalk. Each of these abstractions has been properly aligned to the semantics of generic functions. For the sake of conciseness, we do not discuss these constructs in this chapter and refer the reader to (Vallejos et al., 2009) for a complete explanation.

ContextL

ContextL (Costanza, 2008) is an extension to Common Lisp that enables context-oriented programming. It provides a means to model runtime context-dependent adaptations of a software system as *layers*, which are modular increments to the underlying program definition.

Runtime Context-Dependent Behavioural Adaptations

In ContextL, the context-dependent behavioural variations of a program are represented as a number of partial class and method definitions which are associated with layers. A layer is a first-class entity that can be dynamically activated and deactivated. When a layer is activated, the partial definitions contained in this layer become part of the program until it is deactivated. The effect is that the behaviour of a program can be modified according to its context of use without the need to mention such context dependencies in the affected program. All layer (de)activations have a delimited scope of action which ensures that the behavioural variations are only effective for well-defined parts of a program, and for well-defined durations.

ContextL layers basically consist of only a name and no further properties of their own. However, other constructs of ContextL can explicitly refer to such layers and add definitions to them accordingly. There is a predefined layer named t that denotes the root or default layer, in which all definitions are automatically placed when they do not explicitly name a layer.

The following example shows the definition in ContextL of a context-aware mobile phone that needs to adapt its behaviour between two phone tariffs based on dynamic properties such as time, flat rates, and so on. The interface for making phone calls can be defined as follows:

```
(define-layered-function start-phone-
call (number))
(define-layered-function end-phone-
call ())
```

In the above code snippet two generic functions are defined: start-phone-call which takes a phone number as parameter and end-phone-call which takes no parameters. Layers are defined with the deflayer construct. For example, the two phone tariff layers used to determine the cost of phone calls using different methods can be defined as follows in Exhibit 1.

In Exhibit 1, the start-phone-call and end-phone-call layered methods specify their containment layer using the:in-layer specification. By

Exhibit 1.

```
 (deflayer phone-tariff-a)
(define-layered-method start-phone-call
   :in-layer phone-tariff-a  (number)
   ... record start time...)
 (define-layered-method end-phone-call
   :in-layer phone-tariff-a  (number)
   ... record end time & determine cost a...)
 (deflayer phone-tariff-b)
(define-layered-method start-phone-call
   :in-layer phone-tariff-b  (number)
   ... record start time...)
 (define-layered-method end-phone-call
   :in-layer phone-tariff-b  (number)
   ... record end time & determine cost b...)
```

default, only the root layer is active at runtime, which means that only definitions associated with the root layer affect the behaviour of a program. Other layers can be activated at runtime by way of the with-active-layers language construct. Layer activation is dynamically scoped, ensuring that all the named layers affect the program's behaviour for the dynamic extent of the enclosed program code. Layer activation in ContextL is expressed as follows:

```
(with-active-layers (phone-tarrif-a)
    (start-phone-call...))
```

In the above code fragment, the phone-tarrif-a layer is activated, meaning means that start-phone-call and end-phone-call function calls will execute the call behaviour defined in phone-tarrif-a. Layers can be deactivated at runtime by way of a similar construct with-inactive-layers as follows:

```
(with-inactive-layers (phone-tarrif-a)
    ... contained code...)
```

Such a layer deactivation ensures that none of the named layers affect the program behaviour for the dynamic extent of the enclosed program code. Layer activation and deactivation is restricted to the current thread of execution in multithreaded Common Lisp implementations, to avoid race conditions and interferences between different contexts. Furthermore, layer activations and de-activations can be nested arbitrarily in the control flow of the program.

Ambience

Ambience (González et al., 2008) is a context-oriented language that borrows the Prototypes with Multiple Dispatch computation model from Slate (Salzman and Aldrich, 2005) and is also inspired by the similar object system of Cecil (Chambers, 1992). Ambience's main features are the support of first-class contexts and dynamic behaviour adaptation to such contexts. Ambience is implemented on top of Common Lisp, therefore sharing the same syntax. However, Ambience does not rely on CLOS; rather, it implements its own object model from the ground up. Applications in Ambience are created in terms of objects which are cloned from prototypical objects. Prototypes are objects that act as representative examples of domain entities. Prototypes do not have a special status in the language other than being meaningful exemplars (Lieberman, 1986). By convention, prototype names are prefixed with the @ symbol. For example, in the case of a smartphone, we can define a prototypical @smartphone object from which other smartphone objects can be cloned:

```
(defproto @smartphone (clone @object))
(add-slot @smartphone 'number nil)
(add-slot @smartphone 'ringtone nil)
```

Methods describe prototypical interactions among objects. Every method has a selector that identifies the particular interaction it implements, and a list of prototypical arguments that take part in the interaction. The method is said to be specialised on those particular arguments, and each prototypical argument is called an *argument specialiser*. In Ambience, argument specialisers are plain objects, in contrast with the multimethods of class-based languages such as CLOS (Bobrow et al., 1989) and MultiJava (Clifton et al., 2006), which use classes as argument specialisers. A method defining a call between two smartphones has the following form:

```
(defmethod call ((origin @smartphone)
(target @smartphone))
   (play (ringtone target))
   (format t "Call from ~d to ~d~%"
   (number from) (number to)))
```

Ambient Acquaintance Management

In Ambience, contexts are first-class representations of situations. Contexts reify the physical and logical circumstances in which the system is running. For every relevant situation there is an associated context object that represents such situation computationally. For instance, being inside a car can be associated to a prototypical @ car context; whether it is currently day or night can be represented by @morning, @afternoon and @evening contexts; running with low battery charge can correspond to the @low-power context. The current activities or state of the user can also be reified if needed by contexts such as @ meeting, @programming, @sleeping, and so on. In an Ambience application, a context is defined programmatically as follows:

```
(defcontext @car)
(defcontext @low-power)
(defcontext @meeting)
```

A context object, as any normal object, can delegate part of its behaviour to other objects. These delegate objects can be seen as representing more general situations. General context objects can delegate further to coarser-grained contexts as needed. For instance, the @meeting context can be associated to a more general @silent context. The intention is that a meeting situation is expected to take place in a silent environment. To model this in Ambience, we have the particular context delegate to the more general one:

```
(add-delegation @meeting @silent)
```

Another use of delegation among context objects arises when two or more situations are valid simultaneously. Ambience will produce a *combined context* object representing the joint occurrence of the individual situations. This combined context object delegates to each subcontext object corresponding to each individual situation.

There is a special context combination, the *current context* object, that represents the current situation as a whole, the perceived state of affairs both inside and outside the device, at the physical and logical levels. This representation is subdivided by way of delegation in a number of domain-specific subcontexts. These contexts that are currently reachable in the delegation graph starting from the current context are said to be active. The current context thus serves as a handle to all currently active subcontexts. Note that by definition the current context is always active. Furthermore, it is the most specific context that can possibly be active at any given time. The reciprocal of the active status is of course inactive: any context object that is not linked to the current context delegation graph is inactive. The activation and deactivation of context is controlled at runtime as follows:

```
 (activate-context @car)
   -> @car context is active
(deactivate-context @car)
   -> @car context is inactive
```

Further, the activation of a context implies the activation of its delegate (more general) subcontexts. The effect of activating the @meeting context shown before is therefore as follows:

```
 (activate-context @meeting)
   -> @meeting context is active
   -> @silent context is active
```

Runtime Context-Dependent Behavioural Adaptations

In Ambience, object behaviour exhibited in response to a message send does not only depend on the message arguments, but also on the context from which the message is sent. That is, the context of the caller affects behaviour selection (Harrison and Ossher, 1993). Hence, the behaviour that is exhibited by objects is intrinsically bound to the current (changing) circumstances in which they are used.

Retaking the example shown previously, we will add to our smartphone specialised behaviour when the device is in silent environments. For doing that, we just need to define the specialised implementation in the @silent context (see Exhibit 2).

In this way, Ambience allows modularising behaviour not only according to method specialisers, but also considering the current situation of the running application. At run time our example will behave differently depending of the contexts that are currently active (see Exhibit 3).

When the context is updated by activating @ meeting,

```
(activate-context @meeting)
  -> @meeting context is active
```

the same message shown previously will result in different behaviour:

```
(call bob-phone alice-phone)
  -> Activating vibrator
  -> Silent call from bob-phone to
     alice-phone
```

This way, Ambience offers run-time behavioural adaptation to changing running conditions.

FUTURE WORK

The AmOP and COP languages we have shown have been developed in relative independence, and even though their abstractions have been validated to varying degrees, open questions remain and more experience is still needed in medium- and large-scale applications. More practical experience will help us develop not only new programming abstractions, but also new accompanying methodologies to analyze, design and implement mobile software. Regarding this methodological aspect, a promising line of research we are exploring is the adaptation of Feature-Oriented Domain Analysis, and more particularly the extension of the Feature Description Language, to the case of Context-Oriented Programming (Costanza and D'Hondt, 2008). We are also starting to explore the use of COP

Exhibit 2.

```
(with-context (@silent)
  (defmethod call ((source @smartphone) (target @smartphone))
    (activate (vibrator @target)
    (format t "Silent call from ~d to ~d~%"
              (number from) (number to)))
```

Exhibit 3.

```
(defparameter bob-phone (clone @smartphone))
(defparameter alice-phone (clone @smartphone))
(call bob-phone alice-phone)
   -> Playing ringtone
   -> Call from bob-phone to alice-phone
```

programming abstractions to facilitate the implementation of advanced mechanisms that depend on logical states or modes of operation, such as lightweight memory transactions (Costanza et al., 2009; Gonzalez et al., 2009). Such modes of operation can be regarded as a particular case of execution context. With respect to AmOP, part of our future work focuses on scaling up the number of devices involved (going from tens of devices to thousands of computationally lightweight devices, such as sensor nodes or active RFID tags) (Lombide et al., 2008).

Currently, none of our COP languages incorporates dedicated concurrency and distribution abstractions (although they can use the underlying language's facilities in an ad hoc fashion); conversely, none of our AmOP languages have features that are specifically designed for dynamic behaviour adaptation to context. Therefore, a major research direction we plan to follow in the future is the combination of both AmOP and COP features into one unified computation model. Mobile systems are at the crossroads of concurrent, distributed, adaptable and autonomous systems —a mixture that renders this field unique and open to cross-fertilisation.

CONCLUSION

Mobile computing offers the opportunity to assist people in their everyday activities, right at the place and moment in which software services are needed the most. The usefulness and quality of delivered services can be improved considerably if the software is able to cope seamlessly with mobility and adapt its behaviour according to sensed changes in the environment surrounding the host device. Given that existing programming languages and middleware do not address satisfactorily some of the main challenges we identified for mobile systems programming, we have set out to design our own programming abstractions. The language engineering experiments we have carried out were driven by the need for abstractions that explicitly support the construction of adaptable software running on mobile ad hoc networks. As a result, two fields have emerged: Ambient-Oriented Programming and Context-Oriented Programming. The abstractions they propose are aimed at streamlining mobile software development by supporting adaptation and mobility with less hard-coded and cross-cutting infrastructural code. The use of dedicated programming abstractions helps avoiding the recurring use of special libraries, design patterns and software architectures to support volatile connections, peer-to-peer communication, service discovery, and dynamic behavioural adaptation to context. These characteristics are commonplace in mobile systems, rather than the exception. Provided with adequate abstractions, programmers can concentrate on the core logic of applications, with software designs that match more closely their domains of expertise, thus avoiding a bias towards the non-functional challenges imposed by mobility.

ACKNOWLEDGMENT

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels, the MoVES project of the

Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, and the SAFE-IS project in the context of the Research Foundation - Flanders (FWO). Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

REFERENCES

Achermann, F., Lumpe, M., Schneider, J., & Nierstrasz, O. (1999). *Piccola - A small composition language.*

Achermann, F., & Nierstrasz, O. (2000). Explicit namespaces. In *Modular Programming Languages, LNCS 1897*, (pp. 77–89). Springer-Verlag.

Agha, G. (1986). *Actors: A model of concurrent computation in distributed systems*. Cambridge, MA: MIT Press.

Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., & Moon, D. (1989). Common Lisp object system specification. *Lisp and Symbolic Computation*, *1*(3/4), 245–394.

Caporuscio, M., Carzaniga, A., Wolf, A. L., & Wolf, E. L. (2003). Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, *29*, 1059–1071. doi:10.1109/TSE.2003.1265521

Cardelli, L. (1995). A language with distributed scope. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (pp. 286–297). New York, NY: ACM.

Carreton, A. L., Van Cutsem, T., & De Meuter, W. (2008). Reactive queries in mobile ad hoc networks. In *Proceedings of the 6th International Workshop on Middleware for Pervasive and Ad-Hoc Computing, MPAC '08* (Leuven, Belgium, December 01 - 05, 2008) (pp. 13-18). New York, NY: ACM.

Chambers, C. (1992). Object-oriented multi-methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming, LNCS 615* (pp. 33–56). Springer-Verlag. Clifton, C., Millstein, T., Leavens, G. T., & Chambers, C. (2006). MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems, 28*(3).

Costanza, P. (2008). Context-oriented programming in ContextL: State of the art. In *LISP50: Celebrating the 50th Anniversary of Lisp* (pp. 1–5). New York, NY: ACM. doi:10.1145/1529966.1529970

Costanza, P., & D'Hondt, T. (2008). Feature descriptions for context-oriented programming. In S. Thiel& K. Pohl (Eds.), *12th International Conference on Software Product Lines, Second Volume (Workshops),* (pp. 9–14). Lero Int. Science Centre, University of Limerick, Ireland.

Costanza, P., Herzeel, C., & D'Hondt, T. (2009). Context-oriented software transactional memory in Common Lisp. In *Proceedings of the 5th Symposium on Dynamic Languages*, (pp. 59–68). ACM Press.

Cugola, G., & Arno (2002). Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mobile Computing and Communication Review, 6*(4), 25–33.

Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., & De Meuter, W. (2006). Ambient-oriented programming in AmbientTalk. In *20th European Conference on Object-Oriented Programming*, (pp. 230–254).

Dybvig, R. K. (1996). *The Scheme programming language: ANSI scheme*. Upper Saddle River, NJ: Prentice Hall PTR.

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, *35*, 114–131. doi:10.1145/857076.857078

González, S., Denker, M., & Mens, K. (2009). Transactional contexts: Harnessing the power of context-oriented reflection. In *International Workshop on Context-Oriented Programming*, (pp. 1–6). ACM Press.

González, S., Mens, K., & Cádiz, A. (2008). Context-oriented programming with the ambient object system. *Journal of Universal Computer Science*, *14*(20), 3307–3332.

Halstead, R. H. (1985). Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, *7*, 501–538. doi:10.1145/4472.4478

Harrison, W., & Ossher, H. (1993). Subject-oriented programming: A critique of pure objects. *ACM SIGPLAN Notices*, *28*(10), 411–428. doi:10.1145/167962.165932

Hudak, P., Wadler, P., Brian, A., Fairbairn, B. J., Fasel, J., Hammond, K.,... Young, J. (1992). Report on the programming language haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices, 27.*

Joseph, A. D., Tauber, J. A., & Kaashoek, M. F. (1997). Mobile computing with the rover toolkit. *IEEE Transactions on Computers*, *46*, 337–352. doi:10.1109/12.580429

Jul, E., Levy, H., Hutchinson, N., & Black, A. (1987). Fine-grained mobility in the emerald system. In *SOSP '87: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, (pp. 105–106). New York, NY: ACM.

Kahn, K., & Saraswat, V. A. (1990). Actors as a special case of concurrent constraint (logic) programming. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, (pp. 57–66). New York, NY: ACM.

Lieberman, H. (1986). Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz (Ed.), *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), Vol. 21, (pp. 214–223). ACM Press.

Liskov, B., & Shrira, L. (1988). Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language design and Implementation*, (pp. 260–267). New York, NY: ACM.

Mascolo, C., Capra, L., & Emmerich, W. (2002). Mobile computing middleware. In *Advanced Lectures on Networking* (pp. 20–58). New York, NY: Springer-Verlag, Inc. doi:10.1007/3-540-36162-6_2

Miller, M., Tribble, E. D., & Shapiro, J. (2005). Concurrency among strangers: Programming in E as plan coordination. In R. De Nicola & D. Sangiorgi (Eds.), *Symposium on Trustworthy Global Computing, LNCS 3705*, (pp. 195–229). Springer.

Salzman, L., & Aldrich, J. (2005). Prototypes with multiple dispatch: An expressive and dynamic object model. In *Proceedings of the European Conference on Object-Oriented Programming, LNCS 3586,* (pp. 312–336). Springer-Verlag.

Ullman, J. D. (1994). *Elements of ML programming*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Vallejos, J., Costanza, P., Van Cutsem, T., Meuter, W. D., & D'Hondt, T. (2009). Reconciling generic functions with actors: Generic function-driven object coordination in mobile computing. In *ILC 2009: Proceedings of the International Lisp Conference 2009*, ACM.

Van Cutsem, T., & Mostinckx, S. Gonzalez, Dedecker, J., & De Meuter, W. (2007). AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Computer Science Society*, (pp. 222–248).

Varela, C., & Agha, G. (2001). Programming dynamically reconfigurable open systems with salsa. In *ACM SIGPLAN Notices, OOPSLA'2001 Intriguing Technology Track Proceedings*, (p. 2001).

Yonezawa, A., Briot, J.-P., & Shibayama, E. (1995). *Object-oriented concurrent programming in ABCL/1* (pp. 158–168). Los Alamitos, CA: IEEE Computer Society Press.

KEY TERMS AND DEFINITIONS

Ambient Acquaintance Management: The discovery and management of nearby devices and their hosted services.

Ambient-Oriented Programming (AmOP): An emerging programming model aimed at easing the construction of software deployed in mobile ad hoc networks, by means of dedicated language features that help the programmer in dealing with the hardware characteristics inherent to those networks.

Behavioural Variation: Modularised definition of new, modified and removed behaviour, by means of the underlying programming model's constructs, such as procedures or classes. A variation is therefore expressed as a group of partial definitions, with complete definitions being a particular case.

Context: All computationally accessible information that describes the current situation, such as device location, user activities, people and objects in the vicinity, environmental properties such as lighting and noise, device status such as battery charge and network signal strength, available network peers and the services they offer, and so on.

Context-Oriented Programming (COP): An emerging programming model aimed at easing the construction of adaptive applications by providing features to support context-dependent behavioural variations. COP treats context explicitly, and provides mechanisms to dynamically adapt application behaviour in reaction to context changes.

Layers: Grouping of related behavioural variations. Layers are first-class entities that can be referred to explicitly at runtime, and whose composition can be dynamically controlled on demand. Layers are composed in reaction to contextual information.

Mobile Ad Hoc Network (MANET): A self-configuring open network. Self configuration means that there is no centralised infrastructure. Openness means that hosts freely enter and leave the network at any point in time. Generally, hosts are mobile and thus use wireless network links.

Non-Blocking Communication: Communication that requires neither sender or receiver to be suspended while data is being transmitted, using asynchronous send and receive operations.

Scope: The dynamic extent in which behavioural variations are simultaneously active or inactive within a running application. The scope delimits the execution time span in which variations are effectively applied.