The Context-Dependent Role Model

Jorge Vallejos, Peter Ebraert*, Brecht Desmet,
Tom Van Cutsem**, Stijn Mostinckx*, and Pascal Costanza***

Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
{jvallejo, pebraert, bdesmet, tvcutsem,
smostinc, pascal.costanza}@vub.ac.be

Abstract. Implementing context-dependent behaviour of pervasive computing applications puts a great burden on programmers: Devices need to continuously adapt not only to their own context, but also to the context of other devices they interact with. We present an approach that modularises behavioural adaptations into roles. Role selection takes the context of all the devices involved in an interaction into account, ensures an unambiguous scope of adaptation even in the presence of concurrency, and protects the privacy of the devices. Thus, our context-dependent role (CDR) model facilitates expressing interactions between applications in different, possibly conflicting contexts.

1 Introduction

Context-awareness is commonly defined as the ability of an application to *perceive* and *dynamically adapt* its behaviour to the *surrounding environment* [20]. This definition, however, only seems halfway correct, especially in the presence of distribution. Context-dependent adaptations have particular effects on the interactions between devices, and thus are more difficult to coordinate in pervasive computing systems.

Consider the scenario of the context-aware cellphone, in which a person attending an important meeting does not want to be disturbed by incoming calls. Therefore, his cellphone should, for example, automatically signal incoming calls in a discreet way only. The definition of context-awareness given above suffices for this scenario since the cellphone may adapt its behaviour based on information inferred from its surroundings by means of sensors, like the user's location. However, assume further that this person has a relative who is currently in the hospital, and that he wants to be sure that he does not miss any call from the hospital although he is in an important meeting. The issue here is that he may not know what is the phone number or even the identity of the person who

* Author funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).
** Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.).
*** Author funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

J. Indulska and K. Raymond (Eds.): DAIS 2007, LNCS 4531, pp. 1–16, 2007.

would call him from the hospital. In this case, his cellphone cannot derive the necessary information to decide the kind of signal required for such a special call. The only information it can actually rely on is found in the context of the calling device, which is the fact that the call originates from the hospital. Conversely, if the adaptation were decided at the calling device, it would not only inhibit the callee's ability to discern the calls he wants to receive. It would probably even conflict with the requirements at the callee's phone, as the caller may not, and probably should not, be aware of the callee's context.

The scenario above reveals the problems of distribution for context-dependent adaptations. First, the behavioural adaptation of a device (i.e. signalling calls loud or discreetly) may not only depend on its own context (i.e. "user is in meeting room") but also on the context of all the participants of an interaction (i.e. "call originates from the hospital"). Second, different interactions require adaptations that do not necessarily fit together, such as the different call signals in the cellphone (i.e. loud and discreet signals cannot be combined). Last but not least, external decisions of adaptations can be more vulnerable to context changes and hamper the privacy of the devices.

We propose a role-based object-oriented programming model, called the *context-dependent role (CDR) model*, to facilitate the development of context-dependent adaptations in mobile distributed systems. In this model, (1) roles represent the different behavioural adaptations a software application can dynamically adopt according to the context, (2) an application autonomously decides on an appropriate role based on the context of all the participants, and (3) an adaptation is strictly delimited by the scope of an interaction.

We validate the CDR model by implementing it as an extension to AmbientTalk [16], a programming language especially designed for pervasive computing applications. We use this extension for implementing the scenario of the context-aware cellphone described above.

2 Context-Dependent Adaptations in Mobile Distributed Systems

We now briefly discuss the main properties of context-dependent adaptations in pervasive computing environments to later introduce the specific requirements of distribution for such adaptations. We derive these requirements from the analysis of the scenario of the context-aware cellphone.

Since we focus on context-dependent adaptations in this paper, we do not explicitly deal with the context acquisition, i.e. the way in which software systems obtain information from their surroundings. We rather assume that every application has the necessary support to derive unambiguous context information from potentially unreliable low-level sensor data, like the Context Toolkit framework [24]. We also require that all participants in a mobile distributed system agree on a common representation of the particular context information of interest, like for example the ontology introduced in [22].

2.1 Context-Dependent Adaptations

A context-aware application has to be able to deal with dynamic changes that often lack any periodicity or predictability. The presence of unexpected context changes may lead us to additionally presume that adaptations have to be applied at arbitrary unanticipated points in time. However, adaptations do not necessarily have to happen right after a context change. The context-aware cellphone, for instance, needs to adapt signalling calls only when receiving an incoming call, not necessarily when the user enters the meeting room. This means that an adaptation has a *delimited scope of action*: The adaptation is only required for a specific operation (e.g. a method execution in an object-oriented system) and thus its impact can be limited to the execution of this operation.

In most cases, a context-dependent adaptation affects only parts of the program. The example of the context-aware cellphone illustrates this partial adaptation of behaviour: The adaptation required in this scenario involves exclusively the signals for incoming calls, but leaves other functionality intact. An important requirement for using partial adaptations is that the resulting behaviour of the application should be a *consistent composition* of its default behaviour and the adaptations.

The dependency of the application behaviour on its context does not imply that the code required to reason about the context should get entangled with the rest of the application program. Reasoning about the context inside of the program would lead to undesirable situations such as scattered context-dependent if-statements, resulting in cluttered code that is hard to maintain [12]. Context-dependent adaptations, as well as the reasoning process that they require, should be *modularised* to avoid their entanglement and scattering in the program.

To summarise, context-dependent adaptations:

- occur **dynamically**, with arbitrary frequency, and within a **delimited scope** of action.
- generally affect only part of the program. In this case a **consistent composition** with the rest of the program should be ensured.
- should be **modularised** in such a way that they do not get entangled with the base program.

2.2 Distribution Conditions for Context-Dependent Adaptations

In this section, we analyse the implications for context-dependent adaptations of the distributed nature of pervasive computing environments.

Multiple Influence of Context. The context is not a monolithic and homogeneous set of information for all the participants of a pervasive computing system. It can vary with time and from one device to another. This variability implies that applications might be interacting with others in completely different contexts. The question concerning our focus on context-dependent adaptations is thus how this context heterogeneity may influence the behavioural adaptations of such applications. In the scenario of the context-aware cellphone, for instance,

we observe that the behavioural adaptation on the user's cellphone is not only influenced by its location, but also by the location of the calling device.

Conflicting Adaptations. The problem of conflicting adaptations stems from applications that may be involved in several interactions with different remote applications at the same time. Since presumably these interactions require also different adaptations, there is a high probability that applications end up with adaptations that conflict with each other. The context-aware cellphone, for example, cannot adopt two different call signals at the same time, even if the signals are the appropriate adaptations for two different incoming calls. Part of this problem is directly related to the natural concurrency of the mobile devices. Therefore, adaptations must be circumscribed to a delimited scope of action that is unambiguous even in the presence of concurrent interactions.

Privacy Issues. In a distributed system, the decision whether and how to adapt its components can be made at different physical locations. In pervasive computing systems, however, this condition may raise privacy issues. A context-dependent adaptation decided in a different device from the one affected by the adaptation is neither always possible nor always desirable for the users of mobile devices. In the scenario of the context-aware cellphone, for instance, if the caller could decide that the callee's cellphone should be switched to loud signalling mode, the person at the meeting would lose the possibility to discern the calls he wants to receive.

The same argument can be used to rule out centralised adaptation and decision schemes, developed to coordinate the adaptations of collaborative applications [11,9]. If such a cooperation scheme is required, it should also take into account the privacy of each device involved in a common task. We call this the *non-intrusiveness* principle.

Summary. The distribution requirements for context-dependent adaptations introduced in this section are listed below.

- The behavioural adaptation of an application may depend on **multiple contexts**, especially in the case of interactions with other applications.
- Context-dependent adaptations should be circumscribed to a **delimited scope** of action, in a way that is **consistent with concurrency** to avoid conflicting adaptations.
- Context-dependent adaptations should comply with the **non-intrusiveness** principle to preserve the privacy of mobile applications. This principle is also valid for cooperation schemes of adaptations.

To the best of our knowledge, no existing middleware or programming language offers a solution to deal with all of the properties and which satisfies all distribution requirements for context-dependent adaptations presented in this section. We further discuss the related work in Section 5.

3 The Context-Dependent Role (CDR) Model

To address the requirements discussed in the previous section, we now introduce the *CDR model* for context-dependent adaptations in mobile distributed systems. It extends the actor model [1] of concurrency and distribution with the notion of context-dependent roles. In this section, we describe the semantics for creating, selecting, and adopting context-dependent roles.

We use the context-aware cellphone application identified in this paper to illustrate the different components of our model. We implement this application in AmbientTalk [16], an actor-based programming language especially designed for pervasive computing in which we have developed our model. For the sake of conciseness, we do not present an in-depth discussion of AmbientTalk itself. Instead, we introduce specific features as necessary in the course of this section and refer the reader to dedicated publications [14,15] for more information about this language.

3.1 Flexible Composition of Behavioural Adaptations

In the CDR model, a context-aware application is represented as an actor whose behaviour encapsulates the default functionality of the application and all of its context-dependent adaptations. The default behaviour and the adaptations are modelled as objects and organised in a *delegation* hierarchy. Such a hierarchical delegation structure, originally presented in [21], enables the adaptations to extend the default behaviour of the application – placed at the root of the hierarchy – or any other more general adaptation situated higher up the delegation tree. Figure 1 shows the behaviour of the actor that implements the context-aware cellphone application, specifically its feature to receive incoming calls.

In a delegation hierarchy, an object can either override or share behaviour with its parent. This is especially beneficial for modelling *partial* adaptations. In the context-aware cellphone, for instance, the `loud` and `discreet` adaptation objects each have a specialised implementation of the `signal` method, while they share the behaviour of the `call` method, which is defined in the default

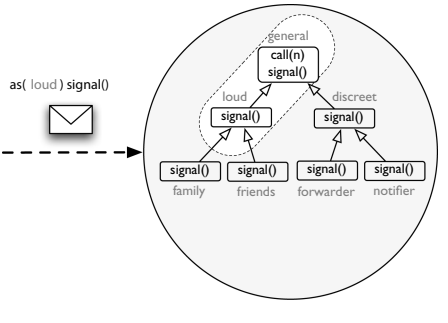

Fig. 1. Context-dependent behaviour of the cellphone actor

behaviour (the `general` object). At the same time, the delegation semantics ensures a consistent interaction between objects that delegate to each other [21]. The following listing presents a definition of the behaviour of the context-aware cellphone actor:

```
contextCellphone: contextActor({
  general: object({
    signal():: {
      playTone("normal-tone")
    };
    playTone(tone):: {...};
    blinkLights():: {...};
    ...
  });
  loud: extend(general,{
    signal():: {
      playTone("noisy-tone");
      blinkLights()}
  });
  discreet: extend(general,{
    signal():: {blinkLights()}
  });
  ...
})
```

Different from normal AmbientTalk actors, whose behaviour is represented by a unique object, actors in our model (created using the dedicated `contextActor` construct) contain multiple behaviour objects. In the code above, the adaptations of the cellphone actor are represented by the `discreet` and `loud` objects which extend the `general` object, overriding the `signal` method.

3.2 Dynamic Adaptation Based on Roles

In the CDR model, the behaviour objects cannot receive messages directly because these objects correspond to the internal state of an actor and, as such, they should only be accessed by the actor. Instead, actors receive messages and respond to them by first selecting the appropriate role and then executing the corresponding method in the adaptation object of that role. The adaptation required by a role-specific message not only involves the object that denotes this role, but also its delegation chain. In the context-aware cellphone, for example, if the `loud` role is specified in an incoming message, the application will respond according to the delegation chain composed of the `loud` and `general` objects (marked by the dotted line in Figure 1).

3.3 Context-Dependent Role Selection

The selection of which role an actor has to adopt to respond to a message is a decision made autonomously by the actor receiving the message but based on the context of both the message sender and receiver. This means that the sender must not indicate the role required for the message execution but rather passes part of its own context information along with the message. The part of the context included in the message is autonomously chosen by the message sending

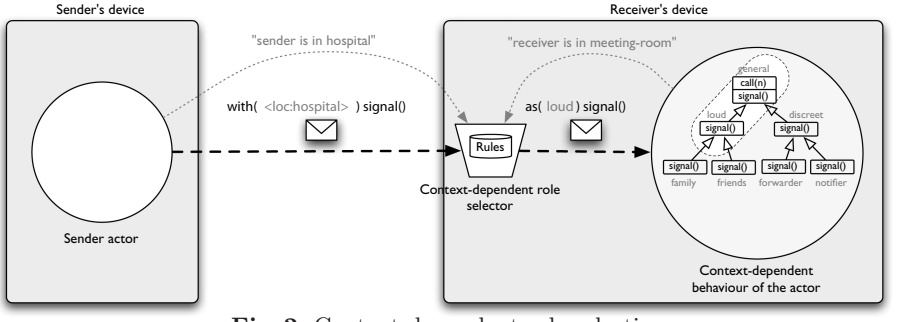

Fig. 2. Context-dependent role selection

actor (discussed later in Section 3.5). Figure 2 illustrates the context-dependent role selection process in the scenario of the context-aware cellphone.

The role selection process is supported by a dedicated entity within the actor, called the *context-dependent role selector*. This role selector is a logic reasoning engine that takes as input the context information of the sender and receiver, together with programmer-defined rules that describe under which conditions a given role can be selected for an incoming message. The output, then, corresponds to the message provided with the role that is most appropriate for the context conditions.

The advantage of designing the role selector as a logic engine is that it offers the developer the expressiveness of the logic programming paradigm. Using a logic programming language, the developer can declaratively specify when a role is applicable, rather than having to specify imperatively when roles become active or inactive by tracking changes in the context.

The rules that the role selector uses to decide on a role have the following structure:

```
role aRole for receiver, message if
   condition1 & ... & conditionN
```

The role selector chooses the role indicated in the head of a rule only if all its conditions are accomplished. The information that these conditions require is retrieved from the logic variables `receiver` and `message`, which are bound to the receiver actor and the message respectively. For instance, we add the following rule to the definition of the context-aware cellphone actor presented in Section 3.1 (by using the `addRule` primitive), indicating that this actor should adopt the `discreet` role when it is at the meeting room:

```
contextCellphone: contextActor({
  general: object({ ... });
  discreet: extend(general,{ ... });
  ...
  // Don't signal calls at the meeting room.
  addRule({role discreet for receiver, message if
              receiver.getContext("location") = "meeting-room"})
})
```

There can be cases where more than one rule matches the context conditions of the receiver and the message that was sent. It means that several roles could be adopted for the same message execution. In our model, however, actors can adopt only one role at the same time, and for this reason, the rules in the context-dependent role selector have a priority order. Only the role that corresponds to the rule with the highest priority is returned as a result. This is the way, for instance, in which the context-aware cellphone application can determine that, although it is in the meeting room, the calls from the hospital should be signalled loudly. In this case, the programmer should add an extra rule for such a new condition with a higher priority than the rule about the receiver in the meeting room described above. The new rule looks as follows:

```
// Signal the calls from the hospital loud.
addRule({role loud for receiver, message if
            message.getContext("location") = "hospital"});
```

In the current implementation of the CDR model, the priority of the rules is determined by the order in which they are defined in the actor, as in the Prolog logic programming language (in Section 4, we propose some alternatives to this basic way of defining priorities). Conversely, it could be that none of the rules matches the current sender's and receiver's context. For this specific case we have defined a default rule without any condition that returns the role corresponding to the default behaviour of the application (the `general` role).

3.4 Delimited Scope of Adaptations

A behavioural adaptation is delimited by the scope of the execution of a message, which means that an actor adopts the role indicated in a message exclusively to process that message. This delimited scope is ensured by the asynchronous message passing mechanism of actors [2] which explicitly separates the message reception from the message execution by using a message queue. This separation enables the actor to adapt its behaviour to individual messages without the risk of affecting or being affected by the adaptations required for other message executions. This separation also enables actors to include the context-dependent role selection as an extra step between the reception and the execution of a message. Figure 3 illustrates the actor with all its components: the context-dependent behaviour, the context-dependent role selector, and the message queue.

Within the scope of the message execution, we can find other information in addition to the sender and receiver's contexts. The receiver's state and the message itself are also part of the context of the communication and hence they can also be used in the definition of the rules and the methods. For instance, assume that the user in the meeting room wants to send back an explanation about why he is not answering the call to all the callers that have an entry in his address book. We define a caller to be a buddy if the caller corresponds to an entry in the user's address book. So then the rule would also be implemented in terms

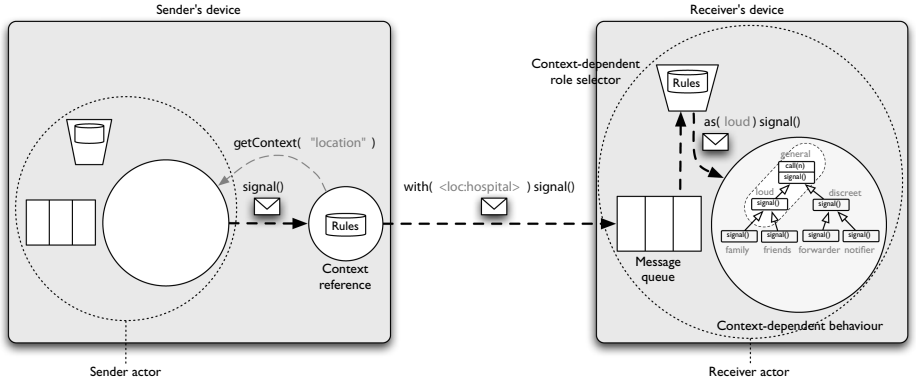

Fig. 3. The implementation of the CDR model in AmbientTalk

of the "inAddressBook" method. The implementation of such a requirement in the actor is as follows:

```
contextCellphone: contextActor({
  ...
  notifier:: extend(discreet,{
    signal():: {
      buddy: getContact(thisMessage.getContext("name"));
      // Send the explanation in a text message.
      buddy<-receiveText("I'm in a meeting until 11:30");
      // Use the signal method defined in the discreet role
      // (the parent of this role).
      super.signal()}
  });

  // Notify only to my buddies that I am in a meeting.
  addRule({role notifier for receiver, message if
              receiver.getContext("location") = "meeting-room" &
              senderName: message.getContext("name") &
              receiver.inAddressBook(senderName)});
  ...
})
```

We benefit from the visibility of the information contained in the message (accessed via the pseudovariable `thisMessage`) in the implementation of the `notifier` role to reply to the message.

3.5 Context Selection

The CDR model preserves the privacy of the receiver actor by allowing it to autonomously decide its adaptations. To also protect the privacy of the message sender we should enable this actor to autonomously select the context that it sends to the message receiver. We introduce *context references* for this purpose. A context reference is a dedicated proxy for a remote actor, whose main responsibility is to get sender's context information and include it in the message sent to the remote actor. Figure 3 illustrates the place of the context reference in the interaction between the two context-aware cellphone applications.

The following listing shows the use of a context reference in the implementation of the context-aware cellphone application, now at the calling device:

```
contextCellphone: contextActor({
  general: object({
    addressBook : makeHashmap();
    addContact(nickname,visibleCtx):: {
      addressBook.put(nickname,makeContextRef(nickname, visibleCtx))
    };
    callContact(nickname):: {
      buddy: getContact(nickname);
      buddy<-signal()
    };
  ... };
... })
```

A context reference is defined using the `makeContextRef` construct. Its definition comprises the identification of the remote actor[1] (represented by the nickname of the cellphone's user in our example), and the context accessible for the reference. Similar to the context-dependent role selector presented in Section 3.3, a context reference enables developers to declaratively specify the part of the context that will be sent to the remote actor. Using the implementation above, for instance, the cellphone's user can add a contact to his address book and reveal his location only during working hours, as follows:

```
contextCellphone<-addContact("Tom",
                             {addRule({send location of sender if
                                       time: sender.getContext("time") &
                                       time > 8.00 & time < 17.00})})
```

The information that the conditions of the rules for the context reference require, is retrieved from the logic variable `sender` which is bound to the sender actor[2]. These rules are evaluated each time the sender actor sends a message through the context reference. The context reference takes into account the context information indicated in the head of all the rules that accomplish their conditions. If none of the rules succeeds, the context reference does not add any context information to the message.

Another benefit of the context references is that they enable the sender actor to abstract from the passing of the context required in the CDR model. This means that programmers do not have to manually include the context information whenever they send messages (e.g. see the `signal` message sent inside the `call` method). The inclusion of the context information occurs transparently in the context reference. At the same time, a context reference is a central place for configuring what context information to expose to the remote actor.

4 Discussion and Future Work

In summary, in the CDR model: an actor encapsulates a delegation hierarchy composed of a default behaviour and its different context-dependent adaptations,

[1] In AmbientTalk such identification corresponds to an intensional description of the service provided by the actor in terms of its properties [16].

[2] We do not consider the receiver for these rules at this stage. In Section 4, we discuss some extensions to this context selection process.

all of them represented as roles; actors respond to messages by first selecting the appropriate role and then executing the corresponding method in the adaptation object of that role; the role required for the execution of a message is autonomously selected by the actor that receives the message, using the context-dependent role selector and based on the context of both the sender and receiver of the message; adaptations have a delimited scope of action which is defined by the execution of a message; and a context reference enables the message sender to be aware of the part of the context exposed to the message receiver.

This model accomplishes the properties of context-dependent adaptations identified in Section 2.1 as follows:

Dynamic adaptations. Dynamic adaptations of behaviour occur transparently for the programmer as a result of the selection of a context-dependent role. This role indicates the behavioural adaptation (object) to which the actor has to address the message.

Delimited scope. Behavioural adaptations are only active within the scope of a message execution. This means that an actor only adopts a certain role to process a single message.

Consistent composition. The composition of behavioural adaptations is defined by the delegation hierarchy. This hierarchy is a flexible structure in which the adaptations can specialise and consistently share behaviour.

Modularisation. Behavioural adaptations in this model are modular since they are encapsulated in objects whose only interaction with the other adaptations is regulated by the semantics of the delegation mechanism. The context reasoning is also concentrated in a single entity called the context-dependent role selector.

The CDR model also copes with the distribution requirements presented in Section 2.2:

Multiple context influence. This model takes the context of all the entities involved in a common task explicitly into account. The execution of a message does not only depend on the application that receives the message (its context and state), but also on the context information of the message sender that is passed along with the message.

Delimited scope and concurrency. Actors communicate by asynchronous message passing which enables them to adapt their behaviour to a message without conflicting with other interactions.

Non-intrusiveness. An actor autonomously decides on the role that it will adopt to process a certain incoming message. This decision is made by its context-dependent role selector. The actor that sends the message also decides autonomously the context information that is passed along with the message.

Although the CDR model can help in tackling some of the challenges for context-dependent adaptations faced in pervasive computing, a number of challenging issues needs to be further explored. For instance, in this model the default

behaviour of a context-aware application is represented as a single object. We are currently investigating an extension to this model that enables programmers to also deal with application behaviours composed of multiple objects. The roles of an application, in this case, do not represent the adaptations of one object but rather *modules* of adaptations for several objects. The essence of these modules of adaptation can be found in the notions of class families in CaesarJ [3], classboxes [6], or layers in ContextL [12]. So far, none of these approaches provide support to deal with pervasive computing systems.

In this paper, we illustrate the benefits of delimiting the scope of an adaptation to the execution of one message. We are currently also investigating how the adaptation scope needs to be propagated in case of having interactions that involve more than one message.

We also propose in this work the use of a delegation hierarchy to model compositions of context-dependent adaptations. In this structure, the adaptations have a predefined location which gives clarity to the behaviour composition, but at the same time restricts the possibilities of adaptations to those denoted by the delegation chains in the hierarchy. A possible alternative to this delegation hierarchy is to have a set of *unwired* behavioural adaptations, similar to mixins [8] or traits [26], which can be dynamically composed whenever they are used.

In the logic reasoning process of context-dependent role selection, we need to ensure that only one role is chosen. For this reason, we establish priorities between the rules that in the current implementation of the CDR model rely on their order of definition. We are currently exploring Choice Logic [29], Ordered Logic [19] and dynamic preferences in Extended Logic programming [10], as more expressive and dynamic ways of defining priorities.

In the CDR model, we define context references as the entities that centralise the context selection process at the message sender's device. A context reference decides the part of the context that is sent to the message receiver based on the sender's context conditions. We are currently working on an extension that also considers the message receiver in the context selection process, e.g. to reason about the context conditions of the receiver or to enable it to prompt the sender for specific pieces of context.

Finally, we are exploring different ways of optimising the logic reasoning process required in the CDR model (context and role selection). We investigate some techniques for caching information [18] and therefore avoiding the recalculation of the role in every message reception.

5 Related Work

Context-Aware Frameworks. There is a huge amount of research on frameworks that support the development and deployment of context-aware systems like WildCAT [13], ContextToolkit [25] or Java Context Awareness Framework [5]. The aim of these frameworks is to provide a generic programming infrastructure that deals with common functionalities like uniform interfaces to access sensor data, event-based system to signal context changes, and reasoning mechanism to aggregate context information. Context-aware frameworks are useful for both

pro-active and reactive systems. In the former case, callback methods are used, as part of an event-driven system, to automatically invoke some behaviour in response to relevant context changes. Additionally, framework solutions also provide the ability to query for actual context information such that reactive systems can adopt their behaviour accordingly. Developers have to rely on traditional dispatching constructs like conditional statements or polymorphism to establish the behavioural adaptation. As soon as context-dependent behaviour appears to be the rule rather than the exception, these language constructs become unmanageable. We therefore argue that context-aware frameworks and our role-based model are actually complementary. Whereas the framework solutions provide the required functionalities to develop context-aware systems, our role-based programming model focuses on how context-dependent adaptations can be decently modelled inside of software systems. The synergy between both proposals supports the development of pervasive systems.

The CORTEX [27] is a middleware architecture that exploits the sentient object paradigm: so-called sentient objects receive events as input (from other sentient objects or sensors), process the events by means of an inference engine and generate further events as output. The communication between sentient objects happens asynchronously via an event layer which hides the network and the transformation process of real-world events. Although our model incorporates concepts that also appear in CORTEX, like asynchronous communication and a reasoning system, both approaches address different application domains. The sentient object model of CORTEX is intended for pro-active context-aware systems that autonomously invoke some action in response to relevant context changes. In contrast, our model deals with reactive systems. That is, upon the reception of a message, the behaviour of the most appropriate role is executed.

Actors in Open Distributed and Pervasive Systems. There exists a number of research proposals that extend the actor model to address the software development issues found in open distributed and pervasive environments. Although so far these approaches do not directly deal with context-dependent behavioural adaptations, some of their coordination and adaptation mechanisms may be useful for developing context-aware applications. SALSA [28], for instance, is an actor-based programming language designed for internet and grid computing. This language enables application's adaptation by means of reflection. The basic operations of the actors (communication and message processing) can be freely manipulated at the meta-level of SALSA. These reflective capabilities are mainly used to fulfil a set of default policies like resource profiling, secure communication and coordination, but new policies can also be defined.

ARC [23] is a role-based coordination model for open, distributed and embedded systems. This model also uses a meta-level but in this case to map quality of service (QoS) requirements to coordination constraints. These constraints are transparently imposed to the actors through message manipulation. Unlike roles in the CDR model, roles in the ARC model are totally independent entities (meta-actors) that provide abstractions for actor functional behaviours and that

can be shared by multiple actors. The local coordination between actors is conducted by the roles whereas the distributed coordination is conducted by other meta-actors called *coordinators*.

Models of Composition and Conditional Selection of Behaviour. The CDR model also shares a number of properties with some object models of composition and conditional behaviour selection. Split objects [4] is a programming model that uses the delegation mechanism of prototype-based languages for role modelling. Similar to an actor in the CDR model, a split object encapsulates a collection of objects structured in a delegation hierarchy that hold part of the description of the split object (state and behaviour) and represent the different roles the split object can adopt to respond to a message. The difference with the actor is that the split object does require that the messages sent to it indicate a role. This means that the split object cannot autonomously decide its adaptation which contradicts the non-intrusiveness principle defined in Section 2.2.

Composition filters [7] is a composition model that enables programmers to modify the behaviour of object-based components through the manipulation of incoming and outgoing messages. As in the CDR model, object behaviours are fully encapsulated, and the behavioural adaptations are exclusively performed inside the component (by using *filters*).

Predicate dispatching [17] generalises a diversity of method dispatching proposals into a unified theory of dispatch. This is established by permitting arbitrary predicates to control the applicability of methods. The authors paid special attention to static typechecking to ensure that there always exists a single most-specific method. Our model can be regarded as a specific application of predicate dispatch in which predicates are associated with roles.

6 Conclusion

Within the domain of pervasive computing, we focus on the capacity of software applications to adapt to their dynamically reconfigurable environments. We describe a number of properties for context-dependent adaptations and then establish some specific requirements of distribution for such adaptations, derived from the analysis of a concrete scenario of context-aware cellphone applications. We observe that context-dependent adaptations occur dynamically and within a delimited scope of action. In addition, these adaptations should be consistently combined with the default behaviour of the application, and clearly modularised to avoid the entanglement between the adaptations and the application behaviour. To cope with the effects of distribution on context-dependent adaptations, an adaptation should take into account the context of all the applications involved in an interaction, have an unambiguous scope of action even in the presence of concurrent interactions, and finally protect the privacy of the interacting applications.

In this paper, we propose the context-dependent model to deal with the properties and distribution requirements described above. In this model, context-aware applications are represented as actors provided with a set of behavioural adaptations organised in a delegation hierarchy. Each adaptation is represented as a role that an actor can adopt to respond to a message. The actor autonomously selects a role for each message based on the context of the message sender and receiver. The context information of the sender that is used for this selection, is passed along with the message and is also autonomously chosen by the sender.

Currently, we are investigating different extensions of our model, like increasing the units of adaptations, making more flexible composition structures of adaptations, propagating the adaptation scope for multiple-actor interactions, and enhancing the expressiveness and efficiency of the context and role selection process.

References

1. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Agha, G., Hewitt, C.: Concurrent programming using actors. Object-oriented concurrent programming, pp. 37–53 (1987)
3. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of caesarj. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
4. Bardou, D., Dony, C.: Split objects: a disciplined use of delegation within objects. In: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 122–137. ACM Press, New York (1996)
5. Bardram, J.E.: The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In: Pervasive, pp. 98–115 (2005)
6. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: controlling visibility of class extensions. Computer Languages, Systems and Structures 31(3-4), pp. 107–126 (2004)
7. Bergmans, L.: The composition filters object model. Technical report, Dept. of Computer Science, University of Twente (1994)
8. Bracha, G., Cook, W.: Mixin-based inheritance. In: Meyrowitz, N. (ed.) Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming, pp. 303–311, ACM Press, Ottawa, Canada (1990)
9. Brewer, E.A., Katz, R.H., Amir, E., Balakrishnan, H., Chawathe, Y., Fox, A., Gribble, S.D., Hodes, T., Nguyen, G., Padmanabhan, V.N., Stemm, M., Seshan,S., Henderson, T.: A network architecture for heterogeneous mobile computing. Personal Communications, IEEE (1998)
10. Brewka, G.: Well-founded semantics for extended logic programs with dynamic preferences. Journal of Artificial Intelligence Research 4, 19 (1996)
11. Correa, C.D., Marsic, I.: A flexible architecture to support awareness in heterogeneous collaborative environments. In: Fourth International Symposium on Collaborative Technologies and Systems (CTS 2003), pp. 109–116 (November 2003)

12. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming - An overview of ContextL. In: Dynamic Languages Symposium (2005)
13. David, P.-C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: MPAC '05. Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, pp. 1–7. ACM Press, New York (2005)
14. Dedecker, J.: Ambient-Oriented Programming. PhD thesis, Vrije Universiteit Brussel (2006)
15. Dedecker, J., Van Belle, W.: Actors for Mobile Ad-hoc Networks. In: International Conference on Embedded and Ubiquitous Computing EUC2004 (2004)
16. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-Oriented Programming in Ambienttalk. In: Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP) Nantes, France (2006)
17. Ernst, M.D., Kaplan, C.S., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: ECOOP '98, the 12th European Conference on Object-Oriented Programming, pp. 186–211, Brussels, Belgium (July 20-24, 1998)
18. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)
19. Gabbay, D., Laenens, E., Vermeir, D.: Credulous vs. sceptical semantics for ordered logic programs. In: Kaufmann, M. (ed.) Second International Conference on Principles of Knowledge Representation and Reasoning, pp. 208–217 (1991)
20. I.A. Group. Ambient intelligence: from vision to reality (September 2003)
21. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Conference proceedings on Object-oriented Programming Systems, Languages and Applications, pp. 214–223. ACM Press, New York (1986)
22. Preuveneers, D., Van den Bergh, J., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V., De Bosschere, K.: Towards an extensible context ontology for ambient intelligence. In: Ambient Intelligence, pp. 148–159 (2004)
23. Ren, S., Yu, Y., Chen, N., Marth, K., Poirot, P.-E., Shen, L.: Actors, roles and coordinators - a coordination model for open distributed and embedded systems. In: COORDINATION, pp. 247–265 (2006)
24. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: aiding the development of context-enabled applications. In: A. Press (ed.) CHI 99: Proceedings of the SIGCHI conference on Humon factors in computing systems, pp. 434–441. New York, USA (1999)
25. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: aiding the development of context-enabled applications. In: CHI '99. Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 434–441. ACM Press, New York (1999)
26. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: ECOOP 2003 – Object-Oriented Programming, LNCS, vol. 2743, pp. 248–274, Springer, Heidelberg (2003)
27. Sørensen, C.-F., Wu, M., Sivaharan, T., Blair, G.S., Okanda, P., Friday, A., Duran-Limon, H.: A context-aware middleware for applications in mobile ad hoc environments. In: MPAC '04. Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pp. 107–110. ACM Press, New York (2004)
28. Varela, C.A., Agha, G.: A hierarchical model for coordination of concurrent activities. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 166–182. Springer, Heidelberg (1999)
29. Vos, M.D., Vermeir, D.: Choice logic programs and nash equilibria in strategic games. In: Flum, J., Rodriguez-Artalejo, M. (eds.) Computer Science Logic, vol. 1683, pp. 266–276. Springer, Heidelberg (1999)