Conversations for Ambient Intelligence

Stijn Mostinckx[⋆], Jessie Dedecker[⋆⋆], Tom Van Cutsem[⋆⋆], and
Wolfgang De Meuter
{smostinc, jededeck, tvcutsem, wdmeuter}@vub.ac.be

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel

Abstract. The development of programs for mobile devices inhabiting Ambient Intelligent network constellations is notoriously difficult. These difficulties stem from the fact that the limitations imposed by the hardware need to be dealt with in an ad hoc fashion. In this paper we advocate the use of *conversations* as a general programming model to develop applications for Ambient Intelligence. We illustrate how conversations tackle the different concerns of ambient-oriented software development.

1 Introduction

The past few years Ambient Intelligence (AmI) has begun to seep into society. Whereas the vision as a whole remains futuristic, the introduction of ever cheaper, smaller and more powerful mobile devices – such as cellular phones and PDAs – cannot go unnoticed. These devices also harbour the necessary wireless network provisions that allow them to escape their isolation and collaborate in open, highly dynamic network settings. Whereas technically feasible, collaboration between different devices remains cumbersome due to the sheer complexity of the software that governs such collaborations. One particularly challenging problem consists of finding abstraction mechanisms for the collaboration between different mobile devices, with respect to the different hardware limitations overshadowing the interaction.

This paper advocates the use of *conversations* [Ran75] – a well-known exception handling technique – as a programming model for developing AmI applications. Upon conducting a careful examination of the hardware limitations imposed by mobile devices, we identify the main concerns to be addressed in *ambient-oriented* software. Subsequently we investigate how conversations can be employed to address these concerns and which extensions and modifications are needed to obtain a full-fledged programming model for ambient-oriented applications.

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

2 Motivation

This section will highlight the main differences between traditional distributed systems and the AmI setting of collaborating devices investigated in this paper. These differences will provide us with a means to evaluate the applicability of conversations in an AmI environment. At present, mobile devices are often characterised by scarce resources such as lower CPU speed, available memory and battery lifespan. However, given the rapid evolution of these devices and their growing resemblance to full-blown computers such as laptops, we consider these issues not to be fundamental characteristics of AmI devices.

The characteristics that we *do* consider to be fundamental are directly related to the peculiarities of the wireless networks that allow the mobile devices to connect with one another. These wireless networks come in two distinct flavours: *nomadic* and *ad hoc* networks. The former network type implies that mobile devices roam while remaining connected through the use of dependable infrastructure. Ad hoc networks on the other hand can be characterised by the absolute lack of central infrastructure to support the interaction. To both network constellations the following observations can be applied:

Volatile Connections A first important difference between traditional distributed systems and an AmI setting of collaborating devices is that the latter can no longer rely on stable network connections. For ad hoc networks with no infrastructure available, connectivity is limited by the range of the wireless facilities. Thus connections may be broken as users move about. When infrastructure is available, roaming users in a nomadic network may still choose to use network facilities periodically, for example to minimise the cost[1] or the battery consumption of upholding a network link.

Ambient Resources In contrast to their counterparts in stationary networks, ambient-oriented applications should not rely on explicit knowledge of the available resources. Instead the availability of resources needs to be discovered dynamically, as the open network dynamically evolves due to the unheralded joining and leaving of devices (which provide specific services).

No Presumed Infrastructure Whereas servers – reliable nodes providing a fixed set of services for their clients – are commonplace in traditional distributed systems, ambient-oriented applications should be able to function without them. Of course, one cannot prohibit software developers to make use of servers in their applications, but the underlying network layer of a programming model for ambient-oriented software should rely only on peer-to-peer networks to accommodate ad hoc network constellations.

Natural Concurrency The need for concurrency naturally arises in a setting populated by mobile devices. It is inconceivable to consider applications that use the dynamics of a network of these devices as a single-threaded application. If this were the case, the disconnection of whatever device that currently holds the running thread would freeze an entire network of devices.

[1] Typically, access to reliable network infrastructure – such as a GPRS-network for SmartPhones – requires payment based on the time one remains connected.

We have explored these hardware characteristics in previous work, to isolate some characteristic features of an ambient-oriented programming language [DVM$^+$05]. A result of this experiment was the development of AmbientTalk[2], a minimal, yet realistic[3], programming language kernel for developing ambient-oriented programs. AmbientTalk introduces actors [Agh86] to encode processes which communicate with one another using asynchronous messages. Such asynchronous communication limits the effects of failing communication links. AmbientTalk additionally provides a basic service discovery mechanism that is based unification of patterns published by the providers and potential users of a service. A thorough introduction of the language is clearly outside the scope of this position paper and can be found in the aforementioned article [DVM$^+$05]. The Ambient Actor model, which is the formal basis for AmbientTalk is detailed in [DV04].

While developing some examples in AmbientTalk, we have come across four concerns that programmers encounter during the development of an ambient-oriented application. The first two concerns are directly related to the exchange of messages between different parties, whereas the latter two are related to the particular network constellations under consideration.

Synchronisation In response to the hardware phenomena described above, ambient-oriented languages require the use of asynchronous, non-blocking communication primitives. However these primitives place a cognitive burden on the programmer, who has to manually encode the synchronisation points in his application using call-backs. This style of programming is akin to continuation-passing style, which is traditionally considered cumbersome to program in. Therefore an important aspect of an ambient-oriented model is the ease with which one can express synchronisation points in the software.

Exception Handling Present-day applications typically use exceptions extensively as a means to signal exceptional events. The need for exception handling becomes obvious in distributed systems, where failing network connections are signalled through the use of exceptions as well. Given the dynamic networks we are investigating, such network exceptions will occur frequently. The well-known try-catch block cannot be aligned with asynchronous message passing, since exceptions may be signalled to the calling device, long after the try-catch block was exited. Because ambient-oriented programming languages are *obliged* to use asynchronous communication, new means for exception handling need to be explored.

Decentralised Distribution Since one can in principle not rely on the availability of any infrastructure whatsoever, it is important that none of the protocols relies on the use of a reliable server infrastructure. In particular the protocols that facilitate the synchronisation and exception handling issues detailed above, should avoid being dependant on a designated leader. Such decentralisation avoids upheaval when the leader becomes unreachable.

[2] More information on the language as well as access to the experimental virtual machine is available at: `http://prog.vub.ac.be/~jededeck/research/ambienttalk/`

[3] The AmbientTalk virtual machine is developed in pure Java and is currently deployed on QTek 9090 SmartPhones.

Service Discovery Since ambient-oriented applications inhabit inherently dynamic open networks, one cannot encode general explicit knowledge that describes where an object may encounter available services. Similarly, the requirement for decentralised distribution also prohibits the use of a name server-based architecture. Thus, when developing an ambient-oriented program, the ability to "sense" services in one's environment is crucial.

The concerns mentioned above will be recapitulated in section 4 to analyse the suitability of the conversation model as a programming model for ambient-oriented software. First, the next section provides a basic introduction to the original conversation model and its more recent offspring the Coordinated Atomic Action model.

3 The Art of Conversation

Conversations were introduced by Randell as an abstraction to control concurrency and communication between collaborating processes [Ran75]. He observed that dependencies are created between processes as they exchange information with one another. Consequently, the effect of a software failure in a single process easily spreads over all dependent processes, since to restart the faulty process, all dependent processes need to be restarted as well. To alleviate this problem, conversations isolate a group of processes, as illustrated in figure 1.

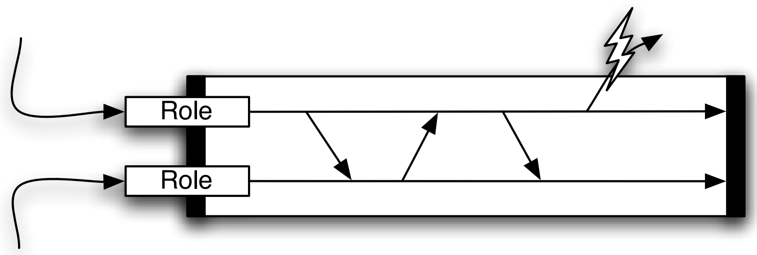

Fig. 1. A traditional conversation isolates its participants from external processes

Figure 1 shows that processes may become *participants* of a conversation by triggering a *role* of the conversation. This role prescribes the behaviour exhibited by the process while it is confined in the conversation's borders. Upon triggering a role – and crossing the left-most barrier or recovery line – the conversation creates a checkpoint of the participant's state. Throughout the conversation, participants may freely communicate with one another, but not with external processes. Such *information smuggling* is prevented to avoid the creation of dependent processes outside of the control of the conversation. Finally, the right-most barrier or test line is used to detect faulty participants. If needed, the conversation restores the checkpoints and restarts itself.

3.1 Coordinated Actomic Actions

Coordinated atomic actions (CA actions) are an object-oriented extension of the traditional conversations described above [XRR$^+$95]. Whereas a thorough discussion of all aspects of the CA action model will lead us too far, we do highlight some of the additional concepts the model introduces.

Forward Error Recovery Whereas traditional conversations respond to errors uniformly by retrying the conversation, the CA action model allows the programmer to specify handlers for exceptions raised by its participants. Upon successfully completing a handler, the CA action terminates normally. Since exceptions may be raised concurrently by different participants, a CA action introduces the notion of a *resolution graph* which maps combinations of exceptions to a corresponding handler. Gathering the different raised exceptions and determining the handler to be triggered by the participants is performed at run-time by the *action manager*.

External Objects Participants of CA actions *may* communicate with external objects, provided that these objects can guarantee transactional semantics. In particular, such external objects should prevent *information smuggling*, by ensuring that the effects of the conversation can be rolled back if necessary. In other words, external objects *"must be atomic and individually responsible for their own integrity"* [VG00].

4 Conversations for Ambient Intelligence

The CAA model described in the previous section has already been applied to a variety of real-world problems [XRR$^+$99,BRR$^+$00,VGR00,ZPR03]. These examples clearly illustrate that conversations are a powerful means of abstraction to structure distributed systems. This section will evaluate conversations with respect to the four concerns of ambient-oriented programming we have identified in section 2. To make this analysis more concrete we outline a minimal AmI scenario to illustrate the different concerns.

> Alan pulls up in front of the restaurant where he has booked a table. As Alan switches off the engine – by retracting his eKey from his dashboard computer to his PDA – a valet approaches offering to park Alan's car. Through his PDA, the valet offers a certificate, which is both a proof that he is employed by the restaurant and a ticket to reclaim the car after the meal. When Alan's PDA validates the certificate, it hands over the car's eKey to the valet.

This scenario, albeit minimal, bears many of the essential characteristics of ambient-oriented software. First of all, the scenario features mobile devices, which connect with one another in an ad hoc fashion. Second, the car parking service offered by the valet is an ambient resource – it becomes available when

Alan pulls up outside the restaurant – which needs to be discovered. Third, the scenario is also interesting since the protocol of interaction between the PDAs of Alan and the valet can be automated. In the remainder of this section, we will illustrate how conversations can be adapted to make this typical AmI scenario possible. Finally, we present some pseudo-code for the resulting conversation.

4.1 Synchronisation

The car parking scenario has two points where the two processes need to synchronise and exchange information, respectively the certificate and the eKey. Conversations provide an apt mechanism to synchronise collaborating processes, since the conversation typically only starts when all roles are attributed to participants. Semantically, this is a very useful property, especially if a participant may supply data, which can be used inside the conversation by all participants, such as the certificate supplied by the valet role[4]:

```
role valet(certificate) {
    // Role body
}
```

Whereas semantically, synchronisation at the start of a conversation is a useful feature, one must take into consideration that technically the actual start of a conversation is determined by the availability of resources. Since an AmI setting is characterised by its dynamic open network settings in which the availability of resources cannot be predicted, the conversation may start an indefinite amount of time after the process has signalled it wants to participate. Since concurrency is a natural phenomenon of our setting, we propose to ensure that the process will only be confined to the boundaries of the conversation once it effectively starts executing its role. As such, the process may still answer asynchronous requests while the other participants are not available yet.

4.2 Exception Handling

The car parking scenario exemplifies the use of exceptions to report on unexpected events, *i.c.* someone may pose as a valet, and present a false certificate to steal Alan's car. Conversations were explicitly designed to handle exceptions in distributed systems, so it should not be surprising that conversations perform this task adequately for ambient-oriented programs as well.

Unlike when using try-catch, it is impossible that an exception is *signalled* when the context in which it was to be handled is already abandoned. This behaviour is due to the fact that processes may not leave a conversation until all participants have completed their role. This logical synchronisation at the end of a conversation can be upheld equally well in an AmI setting. However, since in ambient-oriented software network connections are inherently volatile, blocking participants at the end of a conversation renders the conversations

[4] Other role bodies may access the certificate using valet.certificate

fragile. Therefore, non-functional exceptions [CG03]– *e.g.* signalling that another participant is no longer reachable – should not be handled automatically by the virtual machine, but rather be propagated to the conversation which can then incorporate them in its resolution.

4.3 Decentralised Distribution

Since conversations have been widely used to tackle exception handling in distributed systems, different possibilities to distribute a conversation have already been explored [RZ97]. Whereas *Romanovsky and Zorzo* suggest that both the roles and the action manager of a conversation can be distributed, the solutions discussed in their paper only support distribution of roles. The COALA framework implements distribution of action managers, but still enforces the use of a *leader manager* for coordination. Such a higher authority cannot be reconciled with the AmI criterion that imposes *no presumed infrastructure.* Currently we are exploring different possibilities to relax the reliance on a single leader manager to alert all participants of raised exceptions.

Apart from finding a reliable peer-to-peer protocol to notify all parties of failures, we are also concerned with how the distribution of a conversation-based ambient-oriented program is achieved. We envision the development of such applications as follows: All relevant entities in the program are defined as (ambient) actors. Subsequently, conversations are used to express the collaborations that may occur between these exemplar actors. In the car parking scenario, the role of a valet could thus be specified as follows:

```
role valet(certificate)@CPEmployeeActor {
    // Role body
}
```

The @ syntax shown in the above example confines the participants who wish to take on the role of a valet to the CPEmployeeActor, an exemplar actor representing an employee of the car parking service. The exemplar actor is then cloned to instantiate multiple employees, which can all perform the task of a valet. These clones can then be distributed to the PDA of each employee, which is able to trigger the conversation, if it meets a customer.

In general this mechanism implies that after specifying the different conversations which the exemplar actors may participate in, copies of these exemplar actors can be distributed to all relevant devices. When distributing these cloned actors, one should not be aware of the possible conversations these actors may engage in. The distribution model that we hint at in this section holds the promise of being able to substantiate this vision, without requiring centralised support.

This seemingly oblivious way to obtain distributed conversations, is achieved by introducing *role slots* to represent the roles a conversation attributes to the exemplar actors[5]. These role slots are then used to embody the relation between

[5] The use of role slots is inspired by Slate which introduces them to inform objects of their roles in multimethod invocations [RS05].

actors and the conversations they can participate in. Copies of the exemplar actors contain copies of these role slots to ensure that they can engage in precisely the same conversations as their original.

When distributing ambient actors, the marshalling algorithm will use the role slots to ensure that all conversations in which the actor can participate are made available at the destination node. As such the marshalling algorithm of the language is used to ensure that both the roles and action managers of a conversation are distributed across all nodes that may host participants.

4.4 Service Discovery

Conversations are traditionally not regarded as a means to perform service discovery, yet in the scenario two people meet, exactly because they may interact in a meaningful way. Generalising from this example, we identify conversations as a means to perform *participant discovery.* This implies that processes signal their interest in performing a particular task by participating in a conversation. The conversation is thus conceived as a go-between which brings the process in contact with other potential participants.

Due to the distribution model outlined above, actors are co-located at all times with the different conversations they can participate in. Consequently, participating processes are handled locally without using central infrastructure. Nevertheless, the system should be aware of processes wanting to collaborate. The service discovery system sketched below can be used to fulfil this role.

At the level of the AmbientTalk virtual machine, every device periodically broadcasts its presence, allowing other devices to discover its availability. The service discovery algorithm we propose, reacts to such a notification of presence by transmitting the device's list of active conversations – conversations with at least one role filled in. When receiving such a list of active conversations, the service discovery algorithm will attempt to unify the provided conversations with the ones it hosts on it's own device. Such a unification may produce three different results :

- Match : The unification succeeded and all the conversation's roles have been assigned to participants. Upon encountering a match, the conversation is started.
- Underspecified : The unification succeeded, yet some roles are not filled in. The service discovery algorithm will query all reachable devices, to check whether they can contribute in the conversation. This is necessary to allow for conversations spanning more that two devices. If the conversation, cannot be completed, no matching occurs. No intermediate matching is performed to avoid problems should the partner become unreachable before the remaining roles are filled in.
- Overspecified : The unification failed, for example because both devices filled in the same role. Neither conversation can be started.

Concretely in the scenario, both Alan and the valet will fulfil one role in the conversation: respectively `customer` and `valet`. When Alan then pulls up in front of the restaurant, the ranges of both PDAs intersect, triggering the service discovery algorithm. This unification will indicate a MATCH, and thus initiate the interaction between the devices.

4.5 Scenario revisited

We have analysed how conversations could be conceived as a programming model for ambient-oriented software. Conversations currently do not address all the concerns we have identified as crucial to ambient-oriented software development. However, the ideas we have presented to tackle these problems only involve the support for conversations provided by the underlying virtual machine. Given that the proposed changes are largely invisible to the programmer, we conclude that *conversations provide a potent programming model for ambient-oriented applications.*

In conclusion of this section, we return to the scenario that has guided our analysis. The entire interaction between Alan and the valet may be captured in a single conversation, which is outlined below. The CarParkService conversation essentially establishes the first synchronisation point between the valet and the customer roles.

```
1  conversation CarParkService {
2
3      exception InvalidCertificate();
4
5      method validate(certificate) throws InvalidCertificate{
6          // check the certificate
7      }
8
9      role valet(certificate)@CPEmployeeActor {
10         Parking.valet(certificate);
11     }
12
13     role customer()@PersonActor {
14         validate(valet.certificate);
15         Parking.customer(self.car, self.carkeys)
16     }
```

As can be witnessed the valet at this point makes a certificate available to all participants of the conversation (line 9). While executing its role, the customer will attempt to validate the provided certificate (line 14). Provided that this validation does not throw an exception, another synchronisation point is established between both processes when they join the nested Parking conversation.

```
17     conversation Parking {
18
19         role valet(certificate)@CPEmployeeActor {
20             self.park(customer.car, customer.eKey);
21             self.store("CarRetrieveService", certificate, customer.eKey);
22              }
23
24              role customer(car, eKey)@PersonActor {
25                  self.store("CarRetrieveService", valet.certificate, eKey);
26              }
27
28          }
```

In the nested Parking conversation, the customer hands over his car and eKey to the valet (line 24), which uses them to park the car (line 20). Furthermore, both participants record the relation between the certificate and the eKey (lines 21 and 25) so that the customer can retrieve his car later on.

```
29          resolve(exceptions) {
30              case :
31                  exceptions.contains(InvalidCertificate@customer):
32                      abort();
33                  exceptions.contains(TimedOut) :
34                      retry();
35          }
36      }
```

Finally, the resolve function is called by the local action managers after all roles have terminated and when at least one role has raised an exception. The exceptions parameter contains all raised exceptions in a collection which can be subsequently queried. Also note the use of the @ syntax which allows one to distinguish exceptions based on the role of the participant that raised them.

For this particular example, we have employed default exception handlers. Clearly, the programmer should be permitted to implement his own **handler** functions to incorporate different application-specific exception handling.

5 Position Statement

This paper first identified the main differences between traditional distributed systems and the new emerging field of Ambient Intelligence. These differences are all related to the essentially different characteristics of the network constellations encountered by ambient-oriented software. First of all, ambient-oriented programs are targeted towards networks populated by mobile devices, between which only *volatile connections* can exist. Furthermore the dynamic nature of such networks, implies that the availability of resources cannot be provided up front, such that the program is required to *discover available resources.* Finally, the dynamic networks under consideration may be formed entirely ad hoc, which implies that an ambient-oriented programming language should assume *no infrastructure.*

The impact of these criteria on the development of software is profound. In this paper we have conjectured that – given an appropriate programming language for ambient-oriented programming – the main concerns when developing an ambient-oriented application are :

1. Expressing synchronisation points between the different processes involved.
2. Consequently handle exceptions raised by either the processes themselves, or the non-functional exceptions (*e.g.* to signal disconnection of a partner process) raised by the distribution layer.
3. How easy is it to deploy the application on a network of mobile devices.
4. How the application can become aware of its dynamic environment.

Based on this conjecture, we have proposed the introduction of *conversations* as a programming model for ambient-oriented software development. Through the use of a simple scenario we have analysed how conversations can address the different concerns outlined above. On the other hand, this analysis also indicated room for improvement of the model. In particular, we explored possible ways to tackle both the scattering of conversations over the available devices, as well as a service discovery mechanism, which allows the scattered conversations to reconnect. Since these improvements only involve the level of the virtual machine, we claim that conversations should be considered as a viable programming model for ambient-oriented software.

References

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[BRR+00] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow, and R. J. Stroud. An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *SIGOPS Oper. Syst. Rev.*, 34(4):21–31, 2000.

[CG03] Denis Caromel and Alexandre Genoud. Non-functional exceptions for distributed and mobile objects. In *Workshop on Exception Handling in Object-Oriented Systems (ECOOP 2003)*, 2003.

[DV04] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. In L. Yang, M. Guo, J. Gao, and N. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, August 2004.

[DVM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, and Theo D'Hondt. Ambienttalk : A small reflective kernel for programming mobile network applications. Technical report, Vrije Universiteit Brussel, 2005.

[Ran75] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[RS05] Brian T. Rice and Lee Salzman. The Slate programmer's reference manual, 2005.

[RZ97] A. Romanovsky and A. F. Zorzo. On distribution of coordinated atomic actions. *SIGOPS Oper. Syst. Rev.*, 31(4):63–71, 1997.

[VG00] Julie Vachon and Nicolas Guelfi. Coala: a design language for reliable distributed system engineering. In *Proceedings of the Workshop on Software Engineering and Petri Nets*, Dep. Of Computer Science, University of Aarhus, Denmark, June 2000. 135–154, DAIMI.

[VGR00] Julie Vachon, Nicolas Guelfi, and Alexander B. Romanovsky. Using coala to develop a distributed object-based application. In *International Symposium on Distributed Objects and Applications (DOA)*, pages 195–208, September 2000.

[XRR$^+$95] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS*, pages 499–508, 1995.

[XRR$^+$99] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 68, Washington, DC, USA, 1999. IEEE Computer Society.

[ZPR03] A.F. Zorzo, P. Periorellis, and A. Romanovsky. Using coordinated atomic actions for building complex web applications: a learning experience. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-time Dependable Systems (WORDS 2003)*, pages 288–295. Guadelajara, Mexico, 2003.