Electronic Communications of the EASST
Volume 28 (2010)

Proceedings of the Third International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2010)

Ambient Contracts

Dries Harnie, Christophe Scholliers and Wolfgang De Meuter

6 pages

Guest Editors: Sonia Ben Mokhtar, Romain Rouvoy, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Ambient Contracts

Dries Harnie*, Christophe Scholliers† and Wolfgang De Meuter

Software Languages Lab
Vrije Universiteit Brussel, Belgium
{dharnie,cfscholl,wdmeuter}@vub.ac.be

Abstract: With current programming languages programmers have to manually keep track of device connectivity state changes while interacting with multiple partners in an ambient environment. This leads to complex code which is hard to evolve and maintain. We propose ambient contracts, a novel programming abstraction which tracks connectivity states in order to react appropriately when failure occurs. With ambient contracts the programmer no longer needs to be concerned about connectivity state changes during interaction, which leads to cleaner code.

Keywords: contracts, ambient-oriented programming, multi-party interaction

1 Introduction

With the growing popularity of mobile network technology, we are also witnessing an explosive growth in applications for mobile platforms. A new type of mobile application allows interactions with other devices in the proximity; we call these type of applications *ambient applications*. Due to the environment in which these applications are deployed, the required software engineering abstractions are significantly different from those needed for traditional applications:

The networks utilized, *mobile ad hoc networks*, are spontaneously formed when mobile devices are collocated. As these networks are constantly in flux, ambient applications need device and service discovery at the core of their programming model. The devices running such applications only have a limited wireless communication range, which causes frequent disconnections and reconnections due to user mobility. Therefore device disconnections should be considered the rule rather than the exception.

These properties do not map well to regular programming languages (like Java [Dow98]) which treat disconnections as fatal errors and assume communication references are stable [DVM+05]. This mismatch becomes even more pronounced when programmers attempt to communicate with several partners at the same time in a *multi-party interaction* [HYC08].

Even current approaches for writing ambient applications do not support multi-party interaction well: first of all, there are no facilities for discovering multiple services at once, the programmer has to write code to discover the multiple services separately. Moreover, relationships between services (i.e. two services provided by the same device) have to be computed manually. Finally, current programming languages can only track the connectivity of individual objects. This lack of abstractions for dealing with multi-party interactions results in complex and unmaintainable code. Before presenting our solution under the form of ambient contracts, we show current shortcomings and derive requirements for ambient multi-party interaction abstractions.

* Funded by the Prospective Research for Brussels program of IWOIB-IRSIB, Belgium
† Funded by a doctoral scholarship of the IWT-Flanders, Belgium

2 Scenario

In this section the issues present in multi-party interaction are shown by means of a *smart home* environment [HME+05]. Bob has a television and a sound system in his living room; as he is also an important business person, he receives a lot of phone calls. This however, poses no problems because his living room is a smart environment. When Bob receives a phone call, his digital television will pause *and/or* his sound system will pause (he could be just listening to his sound system). While this example is an extreme simplification of a multi-user interaction pattern it already shows the difficulties of implementing multi-party interactions. Pseudo code for implementing the example scenario in a high-level ambient programming language incorporating single service discovery is shown in Figure 1.

```
1  state := [ nil , nil ]
2
3  discoveredTV(tv) {
4      state [0] := tv
5      when tv disconnects: { state [0] := nil }
6      when tv reconnects: { state [0] := tv }
7  }
8
9  discoveredSoundSystem(s) {
10     state [1] := s
11     when s disconnects: { state [1] := nil }
12     when s reconnects: { state [1] := s }
13 }
14
15 phoneRings() {
16     if ( state [0] == nil && state[1] == nil ) { /* do nothing */ }
17     else if ( state [0] != nil && state[1] == nil ) { state [0]. send("pause") }
18     else if ...
19 }
```

Figure 1: Pseudo code implementing the smart environment scenario.

The implementation first creates two handlers (lines 3–13) which are called when a television or a sound system is discovered. Each handler keeps track of the discovered objects by storing a reference in the *state* array (lines 4 and 10). If the connection is lost the discovered object is removed from the state array, and it is reinserted when the connection is reestablished (lines 5–6 and 11–12). The third handler (lines 15–19) handles the phone ring event: it uses the contents of *state* and a chain of if-else-if statements to determine the appropriate reaction.

The issues mentioned in the introduction become apparent in this scenario: there is no support for discovering and maintaining multiple services at once (lines 3–13). The programmer also has to track their connectivity manually (lines 5–6 and 11–12). A large portion of the code is dedicated to discovery and tracking state changes, while only a small portion of the code is dedicated to the actual base functionality. In current systems it is not possible to specify constraints on discovered objects, therefore Bob's phone could start controlling a television in another room by mistake. Moreover devices cannot refuse service once they have been offered into the environment: if Alice is watching television and Bob receives a call in the office, the television should be able to refuse Bob's *pause* command.

3 Requirements

Now that the issues of multi-party interaction in an ambient environment are made apparent by means of the smart environment scenario, we will present requirements for an appropriate abstraction dealing with these issues.

R1: Discovering multiple objects at once

Currently, there is no language construct for declaratively discovering multiple objects at the same time. As the scenario shows, the programmer has to address this by discovering objects one by one and keeping track of their connectivity state. The complexity of this kind of *stateful discovery* increases exponentially as the number of objects increases. Just as programming languages for writing ambient programs abstract the discovery of single objects, we need a language construct which abstracts the discovery of multiple objects at the same time.

R2: Defining the impact of disconnections and reconnections

Not all disconnections are fatal: sometimes objects are not deemed "essential" for the continuation of a multi-party interaction. This is the case in the scenario: if the TV disconnects we still want the sound system to be paused. As the number of participants in a multi-party interaction grows, the code for properly handling disconnections and reconnections grows as well. Unlike the discovery aspect disconnections and reconnections cannot be anticipated, so code for handling them has to be repeated at each interaction point in the program. If this number is variable, the set of essential objects must be able to grow or shrink dynamically. We need a way of making this set explicit in multi-party interactions and handle disconnections and reconnections accordingly.

R3: Describing relationships between objects in the environment

No object is an island: most objects are related to other objects. Programmers need to be able to make relationships between objects explicit: for example, demanding that two objects are in the same room or from the same device. In the scenario, if Bob has a home cinema system which serves as both a sound system and a TV, it could receive the *pause* message twice. We cannot express this kind of constraints with current programming languages because all object discovery happens independently.

R4: Allowing objects to exercise access control

If an agent exports an object to the outside world, everyone can discover and use it. However, there are a number of situations where controlling the access to exported objects is necessary. For example, a resource-constrained system could allow only a certain amount of users simultaneously and refuse service to new clients. In our scenario, this would give the TV set the means to refuse Bob's commands if Alice is already watching it.

Currently no programming language meets all of these requirements. Given a system that *does* meet the requirements, a programmer can express multi-party interactions without writing stateful discovery code himself and without managing disconnections and reconnections manually. This would greatly improve the clarity and the maintainability of the code.

4 The Ambient Contract Model

In this section we formulate our solution under the form of a novel model called **ambient contracts**. This model is inspired upon previous work called contracts [HHG90]. These contracts,

however, assume a non-distributed object-oriented setting: they do not meet the requirements distilled above because they do not take into account the properties of an ambient environment. Our model extends contracts in order to meet these requirements.

A contract describes a cooperation between a number of participants, where each participant fulfills a well-defined role. A role is an abstract description of the operations a participant should support and which constraints it should satisfy before it is *added* to the contract. Further, a contract describes how to initially set up the roles and which invariants it should maintain once the contract is *initialized* (for example, requiring that a participant is always present).

The two main themes in the requirements for multi-party interaction are object discovery and handling connectivity state changes. With these themes in mind, we will explain how the ambient contract model operates.

In Figure 2, we show the different phases of the lifetime of an ambient contract: discovery, initialization, maintenance and termination. An ambient contract starts in the discovery phase, searching for remote services which can fulfill the roles specified in the contract. When the user's device (a phone in this example) discovers a remote service, it is put in a pool of *connected* services (**1**). Its interface is first compared to the interfaces required by the various roles in the ambient contract. If the service "fits" in a certain role, the ambient contract verifies the constraints which need to be satisfied by this role. Any relationship constraints (**2**) are also verified in this phase (**R3**). If all the constraints are met, the newly discovered service service is requested (**R4**) to join the contract fulfilling this specific role.

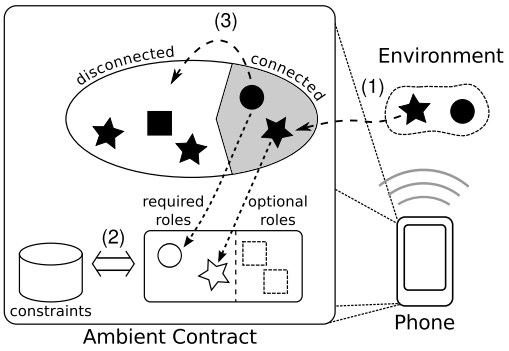

Figure 2: Diagram of ambient contracts

The discovery phase lasts until all essential roles are filled in, at which point the contract enters the initialization phase. In this phase, all participants are informed that the contract has started and the implementation ensures that all invariants are satisfied. The contract then enters the maintenance phase which allows the core logic of the contract to run. From the programmer's point of view, these phase changes all happen simultaneously (**R1**).

During the discovery and initialization phases, disconnections are not treated as errors. Once the contract has entered the maintenance phase however, participants are notified if another participant disconnects. The participant is then marked as disconnected (**3**). If it was part of the "essential" set, the contract is said to be *broken* (**R2**) until the participant reconnects. A broken contract does not allow any message to be sent to any of its participants until it is restored. This restoration occurs if the original participant reconnects, but contract writers could also specify that an equivalent object suffices or that no restoration is possible. If restoration is not possible the contract enters the final termination phase, notifying participants and destroying the contract.

In the ambient contract model, the programmer no longer needs to be concerned about device disconnections and reconnections: he can declaratively specify which services are required and which ones are optional.

5 DEAL: an Ambient Contract Framework

In this section we present DEAL: a prototype implementation of the ambient contract model using AmbientTalk [DVM+05], a high level ambient oriented programming language. To show how ambient contracts enable multi-party interaction, the implementation of the scenario from Section 2 using DEAL is shown in Figure 3.

```
1  def MuteWhenPhoneRings := contract: {
2      role: Phone supports: { ring() }
3      role: AudioDevice supports: { pause() }
4
5      def phone := required: one(Phone)
6      def devices := optional: many(AudioDevice).where( {|ad| ad.room == phone.room} )
7
8      invariant { after phone.ring() { devices.send("pause") } }
9  }
```

Figure 3: Implementing the scenario in Section 2 using DEAL

In the *MuteWhenPhoneRings* contract we define two roles (lines 2–3), each with a set of operations that must be supported. We then use these roles to define the participants in the contract: the first (line 5) states that exactly one *Phone* is necessary. The next (line 6) binds all *AudioDevice*s discovered in the environment to the variable *devices*, as long as they are in the same room as the phone. This constraint is verified by the *where* clause, which evaluates the passed function every time an *AudioDevice* is discovered (the discovered device is bound to the **ad** variable). Finally, line 8 describes the invariant: after sending the *ring()* message to the phone, all audio devices must receive the *pause()* message. Once the contract is defined, the DEAL framework uses the given participant definitions and invariants to set up the appropriate discovery, disconnection and reconnection handles.

All of the issues we described earlier are addressed by ambient contracts: first of all, ambient contracts track state changes automatically, the programmer does not have to write discovery code in a stateful way. Secondly, the *required* and *optional* statements on lines 5–6 explicitly declare which objects are essential to the contract. If any of the audio devices disconnects, it is removed from the *devices* set and the contract can continue. However, when the phone disconnects the contract is broken and all interaction stops until the phone is back in communication range. The third issue (imposing constraints on communication partners) is addressed by line 6: the implementation uses the *where* syntax to express a relationship between the phone and the audio devices. This relationship also implies an order in which the participants of the contract must be discovered; the DEAL framework takes care of this by not adding audio devices to the *devices* set until the phone has entered the contract. Finally, it is not possible to start controlling the wrong device by accident because the implementation negotiates with objects in the environment before adding them to a contract.

Using DEAL, the programmer can incrementally develop ambient applications by evolving contracts along with his applications. For example, he could start with a naive ambient contract which assumes all participants always stay in range, then he could start shrinking the set of essential participants and adding code to maintain the invariants.

6 Conclusion

In this paper we argument that current programming languages do not offer abstractions for dealing with multi-party interaction in mobile ad hoc networks. Some of the issues arising when dealing with multi-party interactions also arise when communicating with a single partner in the ambient, like discovering an object in the environment and managing its disconnections and reconnections. Solutions for single-party interactions in existing languages like AmbientTalk [DVM+05] do not translate straightforwardly to multiple partners. In certain cases it is possible to transform a multi-party interaction into a series of single-party interactions but, as Honda et al. note [HYC08], this is not always possible. Moreover expressing such multi-party interaction as a series of single-party interactions is often extremely difficult, even more so when the parties communicate over unreliable networks.

We propose ambient contracts: a novel language construct which allows programmers to declaratively specify a heterogenous group of objects which are discovered in the ambient environment, enforce invariants, and intelligently handle disconnections and reconnections. Ambient contracts are modular and extensible thereby encouraging code reuse and separation of concerns.

We consider ambient contracts and its prototype implementation DEAL as a first step towards a solution for dealing with multi-party interactions. Future work will focus on refining our solution. Currently it is not possible to respond differently to disconnections depending on the contract's state. Inspiration will be drawn from context-aware application frameworks where the behavior of the application is modified depending on the context [HHS+02]. This work is currently being validated by comparing the use of ambient contracts to equivalent manual implementations.

Bibliography

[Dow98] T. Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc. Foster City, CA, USA, 1998.

[DVM+05] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.

[HHG90] R. Helm, I. Holland, D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *ACM SIGPLAN Notices* 25(10):169–180, 1990.

[HHS+02] A. Harter, A. Hopper, P. Steggles, A. Ward, P. Webster. The anatomy of a context-aware application. *Wireless Networks* 8(2):187–197, 2002.

[HME+05] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, E. Jansen. The gator tech smart house: A programmable pervasive space. *Computer*, pp. 50–60, 2005.

[HYC08] K. Honda, N. Yoshida, M. Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices* 43(1):273–284, 2008.