Pers Ubiquit Comput
DOI 10.1007/s00779-010-0355-z

ORIGINAL ARTICLE

Ambient contracts: verifying and enforcing ambient object compositions à la carte

Christophe Scholliers · Dries Harnie · Éric Tanter · Wolfgang De Meuter · Theo D'Hondt

Received: 29 September 2010 / Accepted: 4 December 2010

Abstract Current programming languages do not offer adequate abstractions to discover and compose heterogenous objects over unreliable networks. This forces programmers to discover objects *one by one*, compose them manually, and keep track of their individual connectivity state at all times. In this paper we propose Ambient Contracts, a novel programming abstraction to deal with the difficulties of composing objects connected over unreliable networks. Ambient Contracts provide declarative heterogenous group discovery and composition while dealing with the unreliability of the network. An ambient contract allows runtime verification *and* enforcement of the messages sent between the participants in the contract. The use of our abstraction significantly reduces the code base and allows programmers to focus on the core functionality of their application. Our claims are reinforced by comparing the implementation of an example scenario in our contracts with a Java implementation using M2MI.

C. Scholliers (✉) · D. Harnie · W. De Meuter · T. D'Hondt
Software Languages Lab, DINF, Vrije Unversiteit Brussel, Brussel, Belgium
e-mail: cfscholl@vub.ac.be

D. Harnie
e-mail: dries.harnie@vub.ac.be

W. De Meuter
e-mail: wdmeuter@vub.ac.be

T. D'Hondt
e-mail: tjdhondt@vub.ac.be

É. Tanter
PLEIAD Laboratory Computer Science Dept (DCC), University of Chile, Santiago, Chile
e-mail: etanter@dcc.uchile.cl

Keywords Contracts · Programming languages · Concurrency

1 Introduction

Developing applications which make use of ambient services is substantially different from developing applications for fixed computer networks because of two important reasons [12]: nodes in the network only have intermittent connectivity (due to the limited communication range of wireless technology combined with the mobility of the devices) and applications need to discover and compose services without relying on a additional infrastructure such as a centralized server. These properties do not map well to regular programming languages [4] which treat disconnections as fatal errors and assume that all communication references are stable. As a result, it is currently extremely hard to program and debug applications that are deployed in such a highly dynamic environment.

The lack of abstractions for *verifying* and *enforcing* ambient service composition in current systems results in complex and unmaintainable code [11]. In this paper we present a novel programming abstraction called *Ambient Contracts* to deal with these difficulties. The four main novelties of our abstraction are:

1. Runtime deployable pre- and post-conditions that verify the interactions of ambient objects;
2. Declarative heterogenous group service discovery;
3. Disconnection strategies to deal with service disconnection in a composition;
4. Ambient access modifiers to shield the access to services from the ambient.

Before presenting our solution in depth, we use an example scenario to show how current approaches fall short and derive requirements for ambient service composition abstractions. Next we present DEAL, a prototype implementation of our ambient contracts and show its effectiveness by comparing the implementation of our example scenario with an implementation using state of the art ambient technology. From micro benchmarks of our prototype we can conclude that the overhead for verification is less than 20%. We finish by listing the related work and the conclusion.

2 Motivation

In this section we present the issues programmers face when composing ambient services by means of a *smart home* environment. Bob has a television and a sound system in his living room; he receives a lot of phone calls because he is an important business person. As his living room is a smart environment watching television and receiving phone calls does not pose any problem. When Bob receives a phone call, his digital television and/or his sound system will pause to not disturb the conversation. While this example is an extreme simplification of an ambient service composition, it already shows the difficulties in implementing such compositions with current software engineering abstractions.

Pseudo code for implementing the example scenario in a high-level ambient programming language incorporating single service discovery is shown in Fig. 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 incomingCall() {
16     if (state[0] == nil && state[1] == nil) then {
17     /* do nothing */
18     }
19     else if (state[0] != nil && state[1] == nil) then{
20         try {
21             state[0].send("pause")
22         } catch(Exception e) {
23             //action
24         }
25     }
26     else if ...
27 }
```

Fig. 1 Manual service composition

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 services by storing a reference in the `state` array (lines 4 and 10). If the connection is lost, the discovered service 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 15–27) is invoked when the phone rings: it uses the contents of `state` and a chain of if-then-else statements to determine the appropriate course of action.

As we can see, in this example a large portion of the code is dedicated to discovery and tracking state changes, while only a small portion is dedicated to the actual base functionality (pausing devices when the phone rings). This is because the programmer does not have appropriate abstractions for dealing with compositions.

First of all, there is no support for discovering and maintaining multiple services at once, so programmers have to discover them separately (lines 3–13) and track their connectivity manually (lines 5–6 and 11–12). Programmers have to manually account for the discovery of a group of heterogenous services. This is particularly difficult when composing a group of objects from the ambient which can disconnect during the discovery process.

Secondly, devices can disconnect at any time so programmers have to write appropriate failure handling and resumption handlers for all code which communicates with external services (try-catch blocks, lines 20–24). Programmers have to manually keep track of the connectivity state of the individual services at all times in a composition, this forces them to scatter the connectivity concern across the rest of the code. This phenomenon has also been seen in the context-oriented programming community: composing context-specific behavior (much like discovery and state handling here) with application logic results in context-related conditionals (if statements) being scattered over the program [2].

In current languages [9], once a service is exported to the ambient environment, *all* devices within reach can access it without limit or control: Bob's phone could start controlling a television in another room by mistake. Services cannot refuse access (from within the model) once they have been exported to 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 smart environment scenario has demonstrated the issues programmers face when composing ambient services, we present requirements that an abstraction for ambient service composition should meet.

3.1 R1: Multi-service discovery

Currently, there is no language construct for declaratively discovering multiple services at the same time. As shown in the scenario, programmers have to emulate this by discovering services one by one and keeping track of their connectivity state. The complexity of this kind of *statefull discovery* increases exponentially with the number of services (in the example, the `incomingCall()` handler has to be extended for the additional states). Just like ambient oriented programming languages provide abstractions for the discovery of a single service, we need abstractions for the discovery of multiple heterogeneous services.

3.2 R2: Managing connectivity

Not all disconnections are fatal: sometimes services are not deemed "essential" for the continuation of an ambient service composition. 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 composition grows, the code for properly handling disconnections and reconnections grows as well. If there is a variable number of participants, the set of essential services must additionally be able to grow or shrink dynamically. An abstraction for service composition needs a way of making the set of essential services in a composition explicit and handle disconnections and reconnections accordingly.

3.3 R3: Ambient composition verification

Programmers need abstractions to express verification constraints over the services they want to compose. For example, a programmer could demand that two services are in the same room or reside on the same device, at discovery time but also throughout the whole interaction. In the scenario, if Bob is in his office and Alice is watching television, a phone call should not pause the television as Bob is not in the same room as the TV. We cannot easily express this kind of constraints with current programming languages.

3.4 R4: Service access control

If a device offers a service to the outside world, everyone can discover it and start using it. However, there are a number of situations where controlling the access to exported services is necessary. For example, a resource-constrained system could allow only a limited amount of users simultaneous access and refuse service to additional clients. In our scenario, this would allow the TV to refuse Bob's commands if Alice is already watching it.

Currently no programming language meets all of these requirements. A system that *does* meet these requirements allows programmers to express ambient service composition without having to write statefull discovery and without managing disconnections and reconnections manually. This makes the developed programs more reusable and evolvable.

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 [8]. The contracts in this previous work, however, assume a non-distributed object-oriented setting: they do not meet the requirements distilled above because they were not designed with an ambient environment in mind. Our model extends contracts in order to meet these requirements.

4.1 Ambient-oriented programming

Before giving an operational description of our ambient contracts we show the object-oriented paradigm in which we have defined our model.

In ambient-oriented programming (AmOP), all distributed communication is *non-blocking*. This allows communicating parties to deal with the impact of intermittent connectivity of devices on the application as their control flow is not blocked upon sending or receiving. In this paper we consider an ambient-oriented concurrency model based on the model of the E language's communicating event loops, which is itself an adaptation of the well-known actor model. In this model, actors are represented as containers of regular objects encapsulating a single thread of execution (*an event loop*) which perpetually take a message from their message queue and invoke the corresponding method of the object denoted as the receiver of the message. The method is then run to completion denoting a *turn*. A turn is executed atomically, i.e. an actor cannot be suspended or blocked while processing a message.

Figure 2 illustrates actors as communicating event loops. The event loop (represented by dotted lines) processes incoming messages one by one and synchronously executes the corresponding methods on the actor's owned objects. Only an object's owning actor can directly execute one of its methods. Communication with an object in another actor happens asynchronously by means of *far references*: object references that span different actors. For example, when A sends a message to B, the message is enqueued in B's message queue, which eventually processes it. As such, a turn consists of the execution of a number of synchronous method invocation and

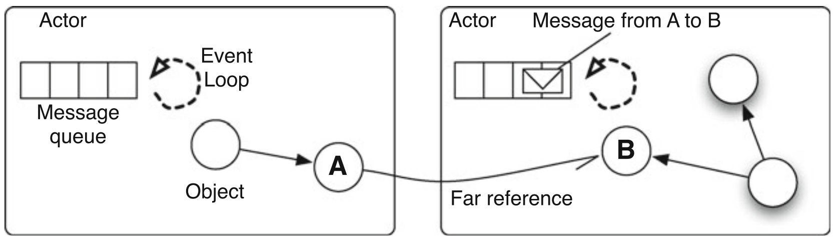

Fig. 2 Communicating event loop model

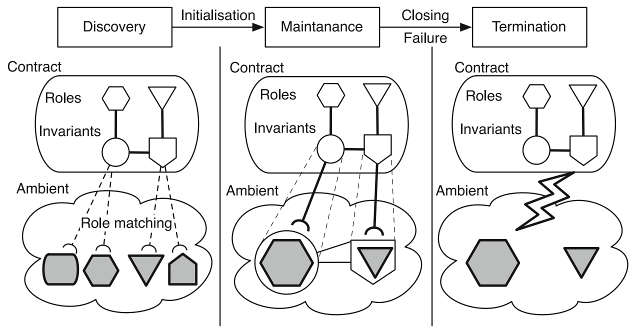

Fig. 3 Diagram of the different stages of an ambient contract throughout its lifetime

asynchronous message sends. This means that the method invocation stack is empty both at the start and at the end of a turn.

4.2 Operational description

A contract describes a cooperation between a number of services, where each service 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 enters 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 does not disconnect during the interaction).

In Fig. 3, the different phases of the lifetime of an ambient contract are shown: discovery, maintenance and termination. An ambient contract starts in the discovery phase, searching for remote services which can fulfill the roles specified in the contract (the hexagon and the triangle in the figure). When the contract discovers a remote service, it is conceptually put in a pool of *connected* services. If the service matches one of the roles in the contract, the ambient contract also verifies the invariants that apply to it (shown right below the roles). These invariants also include relationship constraints that are verified in this phase (R3). If all the constraints are met, the newly discovered service is requested (R4) to join the contract, fulfilling this specific role.

The discovery phase lasts until all required roles are filled in, at which point the contract is initialized and goes into the maintenance 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, the phase change from discovery to maintenance happens atomically (R1). During this phase, if specified in the contract, certain services can be replaced by other equivalent services (R2).

Finally, a contract can be terminated normally (all participants agree) or abnormally (one or more participants have violated the invariants).

4.3 Service definition

An important part of our model is how services are defined and how they are exported into the ambient. As services are discovered in an ad-hoc manner a programmer can not expect services to have direct support for a specific composition. In our model services only have to offer a well defined interface and access protocol such that they can be discovered and used for composition. Once a service is exported, other entities in the environment can discover it by requiring a role that matches a subset of the exported interface. In the next sections we explain in detail how this matching works.

4.4 Ambient access model

In current languages for programming mobile ad-hoc networks [9, 4], once an object is exported to the ambient environment, all devices in the environment can use it without limit or control. This model is too restrictive and does not provide a structured way of limiting the access. In this section we show *ambient views*, which form the access model in our ambient contracts.

Consider a system $\sum$ consisting of objects B, operations O and invocation triples I. We say that an object b contains a set of operations, instance variable accessors and mutators are represented by nullary and unary operations, respectively. In a statically typed language these operations include a type signature, but in a untyped or uni-typed language the name is sufficient to identify the operation. An invocation triple is a tuple of the form (s, c, o), where $s \in B$ is called the server object, $c \in B$ is called the client object,

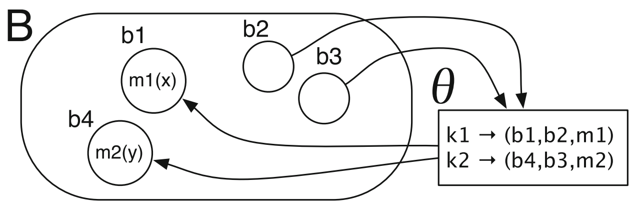

Fig. 4 Specifying the access rights with an intensional description

and $o \in O$ is an operation of the server object s. A client c can only invoke an operation o on an object s if a triple (s, c, o) exists. Objects can always access their own methods.

A view θ is a substitution map of operations to invocation triples. A client object c can only invoke an operation o on a server object s by invoking the operation k on a view θ, if the invocation triple (s, c, o) is an element of θ_k. In order to clarify this consider Fig. 4 which depicts a system with four objects and one view θ. This view allows $b2$ to invoke $m1$ on $b1$ by invoking the operation k1 on the view θ. Similarly it allows $b3$ to invoke $m2$ on $b4$ by operation $k2$.

In a closed system B can be described by using an extensional description, but this is not the case for ambient systems. As objects are ad-hoc discovered at runtime the set of client objects of a view θ can in general *only* be described by using an intensional description. In our model intensional descriptions are modeled by special invocation triples (s, P_c, o) where P_c is a predicate on client objects. These invocation triples allow our model to react to the dynamically changing set of objects. A client object $c1$ can only invoke an operation o on a server object s by invoking the operation k on a view θ if the key k maps θ to a tuple (s, P_c, o) where $P_c(c1)$ holds.

We define the function $\Phi(\theta, b)$, which returns the set of invocation tuples that the object b can invoke on the view θ as follows:

$$\Phi(\theta, b) = \{(s, b, k) | \forall k \in \mathrm{Dom}(\theta), \forall (s, P_c, o) \in \theta_k, P_c(b)\}$$

By offering views as the interface for remote objects the programmer can retain control over the functionality remote objects can invoke, as stated in (R4).

4.5 Role-service matching

Regular programming language abstractions for service discovery do not deal very well with the dynamic nature of an ambient environment. Discovery usually involves *exact* interfaces to be matched on a centralized naming server (on its nominal type), where all services have to register themselves before they can be discovered. A centralized server for service discovery can not be reconciled with the characteristics of an ambient environment where devices discover each other spontaneously, without any infrastructure. Furthermore, reusability and evolvability of services is impeded by the requirement for matching on nominal types, which forces all applications to have a common code base.

In our model, a role is an abstract description of the operations a service should support before it can enter the contract. Roles and services are matched based on their structural type instead of on nominal type in order not to violate the ad-hoc nature of the applications that we target. Services discovered in the ambient are only considered eligible for a role if the role is a subtype of the service being offered. The subtype relationship between a role r and a service θ is defined as follows:

$$r <: \theta \Leftrightarrow \forall o \in r : \exists (s, P_c, o) \in \Phi(\theta, r)$$

4.6 Modeling compositions

As mentioned before, a service composition is formed by matching services in the ambient to the roles defined in the composition. For every role defined in the composition, there will usually be multiple matching services in the ambient. Service composition writers often only want to address a subset of these matching services. In our model we allow service composition writers to specify the desired cardinality with three *mapping operators*: `one`, `exactly` and `many`, which restrict a role-service mapping to respectively one, an exact amount, or a minimum amount of services.

While these mapping operations specify the cardinality of role-service mappings, they do not assist programmers in case of failure. Indeed, due to user mobility services might get disconnected and other services might become available. There are several ways to handle the disconnection of a service: the most obvious way is to wait for that service to reconnect, but this is not always possible. In certain cases it might be appropriate to replace a disconnected service with another equivalent service. For example, mobile phones always rebind themselves to the nearest cellphone tower while moving about. In our model, services that join the contract must be annotated with a *disconnection strategy*. Our model supports three disconnection strategies: the first strategy is `Frail`, which breaks the composition if a service annotated with this keyword disconnects. A second strategy is `WeakRebind`, which denotes that the composition pauses until an equivalent service is discovered. Finally, a service can be annotated with the `SturdyRebind` strategy, which works like `WeakRebind` but only resumes if the original service reconnects.

4.7 Agreement verification

The verification of a contract consists of checking that certain agreements by the participants are met. These

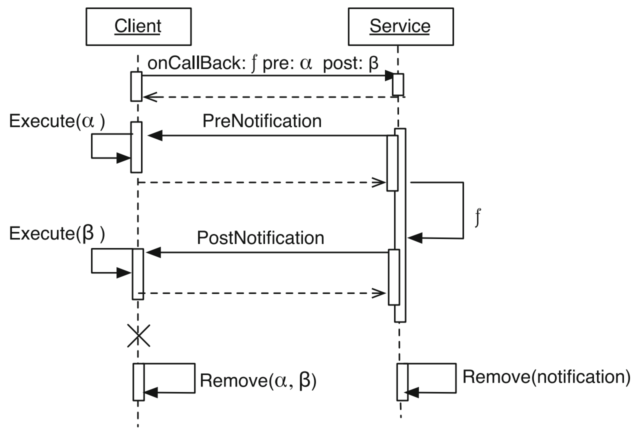

Fig. 5 Registration of pre- and post-conditions on a remote object for local verification

agreements are expressed by runtime deployable pre- and post-conditions. In traditional systems these pre- and post-conditions are statically defined and interweaved with base level code. By contrast, our verification is only deployed when all the participants have joined the contract. Additionally, these pre- and post-conditions are unweaved when the contract is terminated or violated. Specifying the pre- and post-conditions for all the participants of a contract in a single place makes reasoning about their interactions easier.

The interception points of our pre- and post-conditions are defined very similar to pointcuts in aspect oriented programming. This allows the programmer to modularize pre- and post-conditions. In our model an agreement is thus characterized by a pointcut and two functions, one that verifies the preconditions and one that verifies the postconditions.

These pre- and post-conditions can be executed either on the client which deploys the contract, or on the service on which they are defined. Both have their advantages and their disadvantages: in the first case the client does not have to execute untrusted code. Instead, the service will have to notify the client about the pointcut invocations, which results in more messages being transmitted. This pattern is shown in Fig. 5: first the client registers the pre- and post-conditions to the services. The service will monitor its execution flow and notify the client when a function *f* matches the pointcut defined in the registration. Before executing *f* the service will send a `preNotification` to the client which will verify the interaction. When the notification has been acknowledged, *f* is executed and when before returning a `postNotification` is sent to the client. In case the client or the service crashes or disconnects (or both), both will clean up accordingly.

4.8 Enforcing agreements

Traditional contracts are not used for making changes to the base level functionality. We have observed that in an ambient context many services are not written such that they can easily be used in unanticipated compositions. Therefore in our ambient contract model the programmer can enforce certain agreements; for example to broadcast a message to all participants when a phone call is received. The enforcement of agreements follows the same protocol for agreement verification.

It is the service itself which decides whether to allow these kind of agreement enforcements, by allowing contracts to install remote code or not.

4.9 Failures

During the discovery phase, disconnections are not treated as errors and will be dealt by the underlying model, in our model the state transfer from discovery to maintenance is defined as one atomic step. However, once the contract has entered the maintenance phase all participants are expected to obey the agreements stated in the contract. One of these agreements involves the connectivity of the devices: the programmer can state in the contract that a certain object is deemed essential for the composition. This is done by annotating the role of such objects with the disconnection strategy `Frail`. When a service marked as `Frail` disconnects during the maintenance phase of the contract, the contract is said to be *violated* (satisfying R2) and terminated. Similarly, the contract is also violated when one of the participants of the contract violates the pre- or post-conditions stated by the contract.

4.10 Termination

A contract can be terminated gracefully; in this case the contract is *fulfilled* and the participants are no longer obligated to follow the agreements stated in the contract. However a contract can also be terminated prematurely when one of the participants violates the agreements stated in the contract. In this case all the participants of the contract will be notified of the contract violation, which in turn will lead to the unweaving of the installed pre- and post-conditions.

In the ambient contract model, programmers no longer need to be concerned about device discovery, disconnections and reconnections: they can declaratively specify which services are required and which ones are optional and how they must work together to maintain the invariants laid down by the contract.

5 DEAL: an ambient contract framework

In this section we present DEAL:[1] a prototype implementation of the ambient contract model using AmbientTalk [4], a high level ambient-oriented programming language following the Ambient-Oriented Programming paradigm as discussed earlier in Sect. 4.1.

Before showing the implementation of our example scenario as well as a more complex verification we give an overview of the abstractions offered by our framework.

5.1 Framework overview

In Table 1 we have summarized the ambient contract abstractions. They are grouped according to the functionality and the requirements that we have stated in Sect. 3. Many of our abstractions use a keyworded syntax like Smalltalk, e.g.`vector.at: 3 put: 'foo'`.

Programmer can define contracts, roles and services. Roles can be grouped together in order to easily express the number of concrete services that should take part in a contract (R1). Adding a Role$_{group}$ to a contract for discovery always requires the specification of the disconnection strategy. For example, `Frail: one(Phone)` states that exactly one `Phone` service is necessary in a contract and that a disconnection of that service breaks the contract.

Agreements are specified over a Role$_{group}$ by registering a `InvariantRegistrations` block on them. Inside of such a block the programmer has the possibility to register: `onCallBack`, `onCall` and `stateInvariant` invariants. Each of these invariants will be installed to all the objects within a certain Role$_{group}$ when the contract enters the maintenance phase.

Services can be exported in the environment by using the `export` abstraction. The last group of abstractions are defined on a contract to start, stop, and signal a failure. Programmers can also register a callback to be executed if one of these methods are triggered by for example an agreement violation.

5.2 Scenario implementation

In this section we show the complete implementation of the example scenario. We assume that the phone and the audio devices are *not* implemented with the base functionality in mind (pausing the audio devices on an incoming call). Therefore in this section we will highlight the *enforcement* of interactions, in the next subsection we show how to *verify* interactions.

[1] Available as part of the AmbientTalk distribution: http://tiny.cc/tl6ho

Table 1 Overview of the Ambient Contract Abstractions

Constructor functions
`Contract: ContractDefinition` → Contract
`Role: RoleDefinition` → Role
`Service: ServiceDefinition` → Service
Grouping roles (R1)
`One: Role` → Role$_{group}$
`Many: Role`→ Role$_{group}$
`Exactly: Number of: Role` → Role$_{group}$
Role-contract binding for discovery (R2)
`C.Frail:` **Role**$_{group}$
`C.WeakRebind:` **Role**$_{group}$
`C.SturdyRebind:` **Role**$_{group}$
Agreements registration (R3)
`Invariant: InvariantRegistrations on:` **Role**$_{group}$
`onCallBack: Pointcut pre:` λ_{pre} post: λ_{post}
`onCall: Pointcut pre:` λ_{pre} `post:` λ_{post}
`stateInvariant: Id on:` **Role**$_{group}$ `equals: Value`
Exporting views (R4)
`Export: service to:` $\lambda_{criteria}$
`Export: service`
Contract lifetime functions and callbacks
`C.start()` | `C.stop()` | `C.fail(reason)`
`initialise: Block` | `close: Block` | `fail: Block`

```
def Phone := role: {
    def IncomingCall(callerId);
    def name();
    def location ();
}

def AudioDevice := role: {
    def pause();
    def location ();
}
```

Fig. 6 Definition of the phone and audiodevice roles

First we define two roles, `Phone` and `AudioDevice`, as shown in Fig. 6. Each role contains a list of functionality that must be matched with a concrete service. For the `Phone` role, we specify that it has to support the `IncomingCall` method with one argument `callerId`. The other two methods, `name` and `location`, indicate that the service can be asked for its name and its whereabouts. Similarly, the `AudioDevice` role specifies the requirement to support a `location` and `pause` method Fig. 7.

We use these roles to define the services which need to be discovered before the contract can be initialized (lines 12–13). The first line states that exactly one `Phone` is necessary and that the contract will be violated when the phone disconnects, as dictated by the `Frail` keyword. The next line (13) binds all `AudioDevices` discovered in the environment to the variable `devices`. This is done by means of the `many` keyword. A disconnection of an audio device is considered non-critical and when an audio device disconnects it can be replaced by another one. This is indicated by the keyword `WeakRebind`.

Next we specify that only those audio devices which are in the same room as the phone should be included in the composition, using a `stateInvariant` over the set of audio devices.

Finally, lines 20–23 describe a functional invariant: before the `IncomingCall()` method is executed on the phone, all audio devices must receive the `pause()` message. The `beforeCallBack` message installs a callback on the remote service, but the advice (sending the `pause` message to all audio devices) is executed on the deploying client.

Next to the contract, we also show how a service might be implemented. The `Television` service is shown in Fig. 8. We have shown two methods of this service: `pause`, which is accessible by the ambient, and `updateFirmware` which only local clients can use. This access control is done by making use of the two access functions `Public` and `Private` (which map to a predicate P_c in our model).

5.3 Coordinated atomic actions

While the example scenario showed the *enforcement* aspect of ambient contracts, in this section we will highlight the *verification* aspect. We show the implementation of a simplified version of Coordinated Atomic Actions (CAA). This distributed object-oriented coordination pattern is used to implement transactional semantics over a group of objects. A CAA is a contract of limited duration, during which all the objects that take part of the CAA can only send messages to each-other, *not* to outsiders. When one of the participants disconnects or signals a failure, all side-effects are rolled back. In case everything goes fine all the objects commit their changes at the end of the CAA.

An important aspect of a CAA is thus the verification that no messages are being sent to objects which are outside of the CAA. When a message is sent to an object outside the CAA, the receiving object does not know how to report eventual failures or disconnections to the CAA. In Fig. 9 we show the contract which verifies this agreement. The contract has one invariant registration, `onCallBack`, which is triggered for *all* incoming messages (`''.*''`) on participants of the CAA. The precondition verifies that the caller is one of the participants of the contract (no messages may be received from objects outside the CAA). The postcondition verifies that all messages sent in response to an incoming message are sent only to participants. On initialization of the contract, it sends a message to all the participants to start the atomic action. Likewise, when the contract fails either due to a violation or a disconnection, it sends a `RollBack` message to all participants. Finally it sends an `EndAtomicAction` message in case the contract was closed successfully.

```
11 def MuteWhenPhoneRings := contract: {
12     def phone   := Frail: one(Phone);
13     def devices := WeakRebind: many(AudioDevice);
14
15     stateInvariant: ‘location on: devices equals: phone.location();
16
17     invariant: {
18         onCallBack: ‘IncomingCall pre: { |methodName, args, cancel|
19             system.println("Pausing all audio devices");
20             devices<−pause()
21         } post: { |methodName, args, cancel, sentMessages, return|
22             system.println("Unpausing all audio devices");
23             devices<−unpause();
24         };
25     } on: phone;
26 }
```

Fig. 7 Expressing the example scenario by using a contract

```
def Television := service: {
    def location := "Living Room";
    def pause()@Public {
        //method implementation
    }
    def updateFirmware(key, data)@Private {
        //method implementation
     }
}

export: Television;
```

Fig. 8 Service definition with DEAL

6 Evaluation

6.1 Improved software engineering practice

We have evaluated our work by implementing the example scenario both in our ambient contract model and in Java. In Java we have implemented the communication layer by using M2MI [9], a state of the art middleware for the

```
contract: {
    def participants := Frail: exactly(amount, AAParticipant);
    invariant: {
        onCallBack: ".*" pre: { |methodName, args, cancel, caller|
            if: (! participants.contains( caller ) ) then: {
                cancel←cancel();
            };
        } post: { |methodName, args, cancel, sentMessages, return|
            if: ( !sentMessages.receivers().setOf(participants) ) then: {
                cancel←cancel();
            };
        };
    } on: participants;

     initialise : { participants←StartAtomicAction(); };
    fail : { participants←RollBack(); };
    close: { participants←EndAtomicAction(); };
};
```

Fig. 9 Coordinated atomic action verification of incoming and outgoing messages using DEAL

implementation of ambient applications. While M2MI has support for discovery and offers different kinds of group references, we found that it scales badly for the implementation of service compositions. The implementation in Java required 168 lines of code, while the DEAL implementation only used 46 lines of code. We have determined how the code is distributed between the different requirements and core functionality. The result of our analysis is shown in Table 2.

From this table it is clear that in the Java implementation a considerable amount of code is spent on requirement one and two. As the resulting Java code was so entangled we found it impossible to separate the Multi-Service Discovery concern (R1) from Managing Connectivity (R2). M2MI discovery is based on interfaces and thus does not support operations to shield the access to a particular service (R4). As an access protocol would have blown up the implementation too much we have not taken this requirement into account in our comparison. Because a considerable amount of code in the Java code deals with the declaration of interfaces and roles, we did not take that code into account either.

Even though there are significant differences between the amount of code dedicated to the various requirements, the LOC for implementing the core functionality is approximately the same. By making use of ambient contracts the programmer can focus more on the core composition, while in Java the programmer has to deal with all the complexities of an ambient environment manually.

6.2 Computational overhead

In this section, we report on benchmarks of the implementation of our DEAL framework. The aim of our benchmarks is to measure the overhead created by our abstractions compared to a hand-crafted solution which does not verify the interactions. We benchmarked our abstraction by applying a contract with one invariant which verifies a simple identity function of a remote service. As this function is very simple, it just returns its argument, the overhead created by the verification process is presumably high. However as can be seen in Fig. 10 we have determined the overhead of processing an increasing number of messages and observed that this overhead stays below 20%. This is relatively low when taking into account that the execution of the function itself requires almost no time. One reason for this low overhead is that most of the time is consumed in the communication layer.

6.3 Discussion and future work

DEAL addresses all of the requirements distilled from the motivation: first of all, ambient contracts track state changes automatically; the programmer does not have to write statefull discovery code by using event handlers which are difficult to compose and maintain (R1). Secondly, the use of rebinding strategies explicitly declare which objects are essential to the contract (R2). The programmer does not need to be concerned about the individual connectivity of objects: if any of the audio devices disconnects, it is removed from the *devices* set and the contract continues. The third issue (imposing constraints on communication partners) is addressed by functional and state invariants (R3): the relationship between the phone and the audio devices is expressed using the state invariant. On the other hand, the functional invariant takes care of the desired functionality, namely that an incoming phone call should silence the audio devices. Finally, services can declaratively control access to their methods by using access modifiers (R4).

While we provide hooks in the implementation to deal with the registration of multiple contracts (which is outside of the scope of this paper) encoding which object can be safely included in multiple contracts is still cumbersome. Future work will investigate of how contracts can be safely combined.

7 Related work

Our work consists of the model and implementation of a framework which allows the discovery *and* composition of

Table 2 Code distribution in terms of percentage

Requirement	R1&R2 (%)	R3 (%)	Core (%)
Java	73.7	19.6	6.5
Contracts	25	37.5	37.5

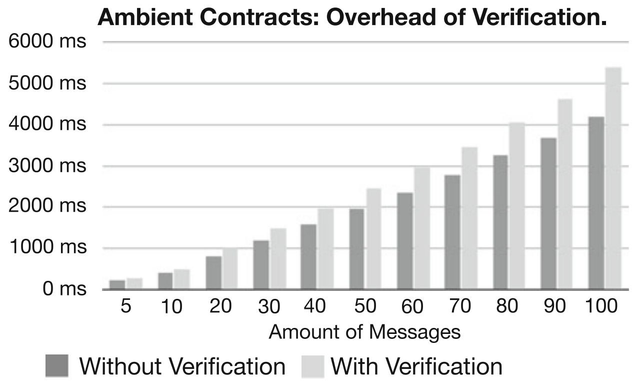

Fig. 10 Overhead of utilizing pre- and post-conditions on a remote object for local verification

groups of objects communicating over an unreliable network. Programmers can verify the interaction and the functionality of a particular service as well as alter its base functionality. Each aspect of our work is related to existing approaches, we have listed these below and show why none of them fulfill all the requirements that we have distilled from the example scenario.

Design by contract is a software correctness methodology which is based on the principle of pre- and post-conditions to assert the change in state caused by certain functionality of the program.

Design by contract is currently the most requested feature[2] to be added to the Java language. They were popularized in the Eiffel programming language and since then adopted in many languages including C, C++, Smalltalk, Haskell, Perl, Python, .NET4 and Scheme [5].

Contracts have been applied in multithreaded object-oriented systems to coordinate a group of objects [8]. However none of the existing contract frameworks are applicable in an ambient environment: many do not allow the replacement of a participant at runtime nor do they support a discovery mechanism.

Many *discovery mechanisms* are currently deployed on mobile phones such as M2MI [9] and Universal Plug and Play (UPnP). Ambient References [3] were introduced to deal with object discovery and interaction in a mobile environment. In order to deal with ad-hoc discovery they support various references, unihandles (one particular object), omnihandles (all collocated objects) and multi-references. They offer similar disconnection strategies as our work, but only support the discovery of homogenous groups of objects. Additionally, they do not provide any abstractions to intercept messages in order to support service composition.

Many service composition frameworks for ambient systems have been developed as summarized in [11]. Contrary to our approach almost all of these systems assume a centralized server to orchestrate the composition. Other frameworks for composition [1] also take this assumption or do not deal with volatile connection [7]. As pointed out in [11] the real problem lies in the use of *physical* proximate services which are combined ad-hoc. This is one aspect of the problem that Ambient Contracts offer a solution for.

Aspect-oriented programming has proven to be useful for dealing with context-awareness [6, 10]. This is *not* the focus of this work, where we have used aspects as an interception abstraction to compose multiple services. The aspects defined in this work are dynamically installed at runtime and cross the boundaries of a single device. We are the first to use remote aspects for coordinating ambient services which *unweave themselves upon failure* in order to deal with device mobility.

[2] http://bugs.sun.com/bugdatabase/top25_rfes.do

8 Conclusion

In this paper we have shown that current programming practice based on event handlers to discover and compose services over unreliable networks falls short for the discovery and composition of ambient services. Next we have proposed our solution: Ambient Contracts, a novel programming abstraction to deal with the discovery and composition of ambient services over unreliable networks. It does so by incorporating a declarative discovery mechanism to address groups of heterogenous services which then can be composed by means of an aspect-oriented interception mechanism. We have evaluated our approach by means of the implementation of an example scenario in Java and compared it to our approach. In the Java implementation the programmer clearly needs to spend most of his time managing the connectivity while in our approach the programmer can focus on the core functionality.

Acknowledgments Christophe Scholliers is funded by a doctoral scholarship of the IWT-Flanders. Dries Harnie is funded by the Prospective Research for Brussels (PRFB) program of IWOIB-IRSIB. É. Tanter is partially funded by FONDECYT Project 1110051.

References

1. Chen H, Finin T, Joshi A (2004) Semantic web in the context broker architecture. In: Proceedings of percom 2004, pp 277–286
2. Costanza P, Hirschfeld R (2005) Language constructs for context-oriented programming: an overview of contextl. In: DLS '05. ACM, New York, pp 1–10
3. Van Cutsem T, Dedecker J, Mostinckx S, Gonzalez E, D'Hondt T, De Meuter W (2006) Ambient references: addressing objects in mobile networks. In: OOPSLA '06. ACM Press, New York, pp 986–997
4. Dedecker J, Van Cutsem T, Mostinckx S, D'Hondt T, De Meuter W (2005) Ambient-oriented programming. In: OOPSLA '05. ACM Press, New York
5. Findler RB, Felleisen M (2002) Contracts for higher-order functions. SIGPLAN Not 37:48–59
6. Fuentes L, Gámez N (2009) Modeling the context-awareness service in an aspect-oriented middleware for ami. 3rd Symposium of Ubiquitous computing and ambient intelligence 2008, pp 159–167
7. Gu T, Pung HK, Zhang DQ (2004) A middleware for building context-aware mobile services. In: Proceedings of IEEE Vehicular Technology Conference (VTC)
8. Helm R, Holland IM, Gangopadhyay D (1990) Contracts: specifying behavioral compositions in object-oriented systems. ACM SIGPLAN Notices 25(10):169–180
9. Kaminsky A, Bischof HP (2002) Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In: OOPSLA 2002. Citeseer
10. Tanter E, Gybels K, Denker M, Bergel A (2006) Context-aware aspects. In: Proceedings of the 5th international symposium on

software composition (SC 2006) LNCS 4089. Springer, New York, pp 227–249

11. Urbieta A, Barrutieta G, Parra J, Uribarren A (2008) A survey of dynamic service composition approaches for ambient systems. In: SOMITAS ’08. ICST, Brussels, Belgium, Belgium, pp 1–8
12. Van Cutsem T, Mostinckx S, Boix EG, Dedecker J, De Meuter W (2007) Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In: SCCC. IEEE Computer Society, pp 3–12