Vrije Universiteit Brussel

Faculteit Wetenschappen
Departement Informatica
en Toegepaste Informatica

Engineering Mobile Applications using Declarative Field Annotations

Proefschrift ingediend met het oog op het behalen
van de graad van Licentiaat in de Informatica

Toon Verwaest

Promotor: Prof. Dr. Theo D'Hondt
Begeleiders: Jorge Vallejos Vargas
Dr. Wolfgang De Meuter

MEI 2006

Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
and Applied Computer Science

Engineering Mobile Applications using Declarative Field Annotations

Graduation thesis submitted in partial fulfillment of the
requirements for the degree of Licentiate in Computer Science

Toon Verwaest

Promotor: Prof. Dr. Theo D'Hondt
Advisors: Jorge Vallejos Vargas
Dr. Wolfgang De Meuter

MAY 2006

Samenvatting

De opkomst van mobiele netwerken heeft de nood voor applicaties met de capaciteit te verhuizen van apparaat naar apparaat aangewakkerd. Hoewel er al oplossingen voor dit probleem zijn, komen deze meestal neer op de mobiliteit van aparte entiteiten.

In deze verhandeling onderzoeken we de verschillende types van relaties tussen bewegende objecten die men tegenkomt in mobiele omgevingen. Om deze relaties eenvoudig te kunnen opleggen, stellen we voor de huidige oplossingen uit te breiden met *declaratieve annotaties van velden*. We valideren deze techniek door, aan de hand van een programmeertaal uitgebreid met deze annotaties, een reizende *TrafficWare routeplanner* te implementeren.

Abstract

The uprise of mobile networks has generated the need for parts of mobile applications to be capable of moving from one device to another. While there are already solutions for moving applications, they are mostly constrained to the mobility of single entities.

In this dissertation, we investigate the different types of relations between moving objects that can be found in mobile environments. To easily impose these relations we propose extending the current solutions with *declarative field annotations*. We validate this technique by using a language extended with it to implement a moving *TrafficWare route planner*.

Acknowledgements

I would like to thank all the people who supported me during the research and the writing phase of this dissertation.

Prof. Dr. Theo D'Hondt for promoting this thesis.

Dr. Wolfgang De Meuter for coming up with an interesting thesis subject even while my wishes were not always clear.

Jorge Vallejos Vargas for the hours of discussion on the subject, for reading and rereading my dissertation several times (even until the very last moment) and correcting me where necessary.

All former and current members of *InfoGroep* for the friendship and the knowledge I have received over the years. Special thanks goes to Jan Vandenbussche who proofread parts of my dissertation and Kim Gybels for all I have learned from him, the long nights of programming together, and finally also proofreading my dissertation.

Laura Sanchez Serrano for helping me relax during the writing of the thesis, and wasting precious time we did not have.

Finally my parents for letting me study anything I wanted. My mother for pushing me to work hard enough and my father for calming down my mother from time to time.

Contents

List of Figures

List of Tables

Listings

1 Introduction

With the uprise of mobile devices equipped with wireless connectivity, not only the ease for the users, but also the application complexity have increased. Most of these mobile devices are equipped with wireless units with limited communication range. At the same time their physical mobility capability allows them to move out of communication range. Applications written for these devices are becoming more and more environment aware. Thus the applications might depend on resources (software and/or hardware services) running on remote devices. Network partitions, caused by users walking out of communication range, would result in those services becoming unavailable, possibly resulting in stalling the applications. A possible solution for this scenario might be moving the running applications in such a way that the used resources stay available. This calls for the need of *code mobility*, the ability to move running programs from one device to another.

1.1 Problem Statement

Code mobility is not a new concept. In the past, several mechanisms and facilities have been implemented to move code along the nodes of a network. These implementations however mostly boil down to the addition of one *move operator* to the programming language. This enables programmers to move single objects from one device to another. The problem with these solutions lies in the fact that these languages are only focused on the mobility of single objects in the system, while

applications are usually composed of groups of objects working together on tasks.

The evolution in mobile software engineering might be compared to the upcoming of object-oriented programming. In the beginning, research was focused on the development of "single object systems" with the coming of SIMULA I and SIMULA 67 [Dha01]. Subsequent research was conducted on how to create the objects using classes and inheritance. However, as these systems were actually being used more and more, it became clear that not only the design of the objects themselves is important, but also the design of the relationship between objects. This resulted in so-called design patterns. Focusing on such mechanisms during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored [EGV94].

1.1.1 Single Entity Mobility

While single moveable objects serve as starting point for applications, exploiting the mobility of a programming language, like objects are the starting point for object-oriented applications, they are not sufficient for good mobile program design. Objects in applications are usually not isolated computational entities. They rely on other objects, representing other parts of the application or resources for the application. On moving a computational entity from one device to another, it is very probable that something also needs to happen to the cooperating objects and/or resources of the moved object. Languages only based on the single entity mobility feature will result in program code cluttered with mobility specific statements, written to handle the mobility of interlinked objects.

1.1.2 Multiple Mobility Semantics

For programs written in single entity mobility languages, in which the mobility situation for certain objects has to change according to the state of the application, the situation would even be worse. Such programs are characterized by branched mobility code fragments. In some cases it might even be necessary for programmers to extend functions which have nothing to do with mobility, with mobility specific code, because of the state changes these functions imply.

Imagine a drawing program for example. After only drawing a few squares and circles it will probably be faster to redraw the canvas when moving the application instead of moving the state of the canvas over the network. This could be achieved by re-applying the performed operations to the recreated canvas. In other cases however, the optimal solution might be different. If one would draw some fractal which takes a long while to generate, it might be faster just to move the current state of the canvas.

1.2 Proposed Solution

In order to avoid this code cluttering due to single entity mobility, there is the need for well-defined relations between moving objects and their resources, an easy way to impose and an easy way to change these relations.

In [AFV98], Alfonso Fuggetta already pointed out the need for different mobility semantics depending on the type of resource. He presented a list of possible relations between moving objects and their resources. Unfortunately current mobile systems do not provide support for easily applying these mobility semantics. They all rely on one or two of the possible mechanisms as default behaviour, leaving the non-standard work to the program designer and his eventual mobile program code.

1.2.1 Multiple Entities Mobility

By examining code mobility as mobility applied to groups of objects working together on a problem, we identify the different possible relations between moving objects in different scenarios. To easily impose the different relations, we propose extending mobile programming languages with a *declarative field annotation system*. This way the fields of moveable objects can be annotated with mobility semantics based on the relations we identified. In order to easily change the mobility semantics, we advice to use a dynamic annotation system rather than a static one.

1.2.2 Advantages of Declarative Field Annotations

The annotation mechanism will enable programmers to write mobile applications with less code entanglement, making the code better structured and clearer. Secondly the programmer does also not have to think about the different mobility scenarios any more. He is presented with a list of possible annotations which he can use to define the mobility relations between different objects in his program.

1.3 Outline

In this dissertation we will show how to extend a programming language designed for mobile networks, with the declarative field annotations for mobility. We will also explain which annotations should be installed by default. Finally we will validate our solution by using these annotations to implement a real-world application. The document is organized as follows:

In chapter 2 we will describe mobile networks and a programming paradigm specially designed for these dynamic environments. Secondly we will present a concrete language of the paradigm called *AmbientTalk*. In chapter 3 we will look at code mobility more thoroughly. We will point out why we want code mobility in the first place and which type of mobility is the most feasible. We will also implement this type of mobility in AmbientTalk. In chapter 4 we will identify the different mobility scenarios and propose the use of *typeboxes* as dynamic declarative field annotation system. Next we will extend the version of AmbientTalk we created in chapter 3 with an implementation of these typeboxes and we add default typeboxes based on the identified mobility scenarios. In chapter 5 we will validate our solution and show how it can be used in real-world examples. We finish the dissertation in chapter 6, where we draw conclusions from the accomplished goals during our research.

2 Ambient-Oriented Programming in AmbientTalk

This chapter introduces ambient-oriented programming. We will show how it simplifies the task of programming distributed applications for mobile networks. Since this is the type of network in which our research is based, this is the ideal paradigm as starting point. Then we will see a concrete ambient-oriented programming language called AmbientTalk. The language is designed as a reflective kernel. This allows language engineers to easily adapt the language in order to experiment with language constructs. The language will be used as basis for our research experiments.

2.1 Mobile Networks

Software development for mobile devices is given a new impetus with the advent of *mobile networks*. Mobile networks surround a mobile device equipped with wireless technology and are delimited dynamically as users move about. Mobile networks turn isolated applications into cooperative ones that interact with their environment [JDM06]. Such mobile networks have four main characteristics which distinguish them from other other types of networks:

- **Connection Volatility.** Two devices which perform a meaningful task together cannot assume a stable connection. The limited communication

range of wireless technology combined with the fact that users can move out of range, can result in broken connections at any point in time. However on re-establishing a broken connection, users typically expect the task to resume.

If possible, it might even be better to avoid this situation by moving the two parties to one device, so network partitions between the parties are not possible anymore.

- **Ambient Resources.** Because the availability of a resource may depend on the location of a device, users who walk around with their mobile device might cause these resources to become dynamically (un)available in the environment. This is in contrast with stationary networks in which references to remote resources are obtained based upon the explicit knowledge of the availability of the resource.

- **Autonomy.** Most distributed applications today are developed using a server-client approach. The server often plays the "higher authority" which coordinates interactions between clients. In mobile networks such a "higher authority" is unavailable because every device should be able to act as an autonomous client en an ever changing environment.

- **Natural Concurrency.** In a client-server setup, a client might explicitly wait for the result of a request to a serving device. Since waiting undermines the autonomy of a device, we conclude that concurrency is a natural phenomenon in software running on mobile devices.

Developing software for mobile networks in conventional programming languages is extremely hard. The main reason is that contemporary programming languages lack abstractions to deal with the mobile hardware characteristics. For instance, in traditional programming languages, failing remote communication is usually dealt with using a classic exception handling mechanism. This results in application code polluted with exception handling code because failures are the rule rather than the exception in mobile networks. Observations like these justified the need for a new programming paradigm, the *Ambient-Oriented Paradigm* (*AmOP* for short) which consists of programming languages which explicitly incorporate potential network failures in the very heart of their basic computational steps.

2.2 The Ambient-Oriented Programming Language

This section presents a collection of language design characteristics that discriminate the AmOP paradigm from classic concurrent distributed object-oriented pro-

gramming. These characteristics are directly derived from the hardware phenomena we summarised in section 2.1. Until now, it seems that the object-oriented paradigm is the most successful one with respect to dealing with distribution and its induced concurrency, because it successfully aligns encapsulated objects with concurrently running, distributed software entities [JPBL98]. Therefore, our most basic research assumption is that ambient-oriented programming languages necessarily are concurrent distributed object-oriented languages. However, ambient-oriented programming languages differ from conventional distributed concurrent object-oriented programming languages in at least one of the four ways presented below.

2.2.1 Classless Object Models

As a consequence of argument passing in the context of remote messages, objects are copied back and forth between remote hosts. Since objects in class-based programming languages cannot exist without their class, the copying of objects implies classes have to be copied as well. However, a class is, by definition, an entity which is conceptually shared by all its instances. From a conceptual point of view there is only one version of any class on the network, containing the shared class variables and method implementations. Since objects residing on different machines can autonomously change class variables, or update the implementation of "their" version of the class, a classic distributed state consistency problem among replicated classes arises. Allowing programmers to deal with this phenomenon requires a *full* reification of classes and the instance-of relation. However, languages like Smalltalk and CLOS already illustrate that, even in the absence of wireless distribution, this results in extremely complex meta-machinery.

A much simpler solution consists of getting rid of classes and the sharing relation they impose on objects all together. This is the paradigm defined by prototype-base languages like Self [US87].

2.2.2 Non-Blocking Communication Primitives

Autonomous devices communicating over volatile connections necessitate non-blocking communication primitives since blocking communication would harm the autonomy of mobile devices. First, blocking communication is a known source for (distributed) deadlocks which are extremely hard to resolve in mobile networks since not all parties are necessarily available for communication. Second, a program or a device could block for a unwanted long period of time upon encountering volatile connections or temporary unavailability of another device.

2.2.3 Reified Communication Traces

Non-blocking communication implies that devices no longer communicate synchronously. However, two devices working together on a task might need to communicate synchronised, to avoid ending up in an inconsistent state. When such an inconsistency is detected, both parties should be able to restore their state to a consistent one. Therefore the AmOP language should provide the programmers with an *explicit representation* (i.e. a reification) of the communication details which led to inconsistent state.

2.2.4 Ambient Acquaintance Management

The fact that autonomous devices detect ambient resources dynamically while they are roaming about, means that they do not necessarily depend on a third-party to interact with each other. As opposed to the client-server model, where the server directs communication between clients, devices do not need explicit reference to one another beforehand. This distributed naming provides a way to communicate without knowing the address or location of the ambient resource.

2.3 An Ambient-Oriented Language: AmbientTalk

Now we are familiar with the basic characteristics for AmI[1] languages, we present a concrete descendant called AmbientTalk [JDM06]. It is designed as a reflectively extensible kernel, as a language which enables language designers to explore the realm of language features which facilitate AmOP. Since this makes the language perfectly fit for our research, we are going to use this language as basis for the rest of the dissertation. In this section we begin by explaining the essential characteristics of its object model.

2.3.1 A Double-layered Object Model

AmbientTalk has a concurrent object model that is based on the model of ABCL/1. This model features active objects which consist of an interminable running thread, an updateable state and a message queue. These concurrent active objects communicate by the use of asynchronous message passing. Upon reception, these messages are stored in the receiving active object's message queue, and are processed sequentially. By excluding simultaneous message processing, race conditions on the updateable state are avoided. The advantage of this model is that it

[1] AmI: Ambient Intelligent

		Receiver	
		Passive Objects	Active Objects
Parameter	Passive Objects	Reference	Copy
	Active Objects	Reference	Reference

Table 2.1: Parameter Passing in AmbientTalk

combines imperative object-oriented programming and concurrent programming without suffering from possible race conditions.

In order to avoid programs to be equipped with explicit concurrency provisions, and to avoid that every message send in a program must be considered to be a concurrent one, a more fine-grained model which distinguishes between active and passive objects is used. This allows programmers to only deal with concurrency when strictly necessary. Since passive objects are always handled by one and the same thread, synchronous message passing is used. However, if we are combining active and passive objects, we must make sure a passive object is never shared by two active ones. Otherwise this would easily lead again to race conditions. AmbientTalk's object-model ensures this by obeying to the following two principles:

- **Containment.** A passive object is always contained in at most one active object. Thus there is no sharing of passive objects between active ones. The only thread which can enter a passive object, is the thread of the active object in which the passive object is contained.

- **Parameter Passing.** When an asynchronous message from one active object to another is sent, a passive object might be passed as an argument. In order not to violate the containment-principle, this passive object will be passed by copy. Since message sending to active objects obeys the parameter passing principle by definition, if we pass these type of objects as argument, it will simply be by reference. Of course, arguments passed while sending synchronous message to passive objects will always be passed by reference. These type of messages to not cross the border of active objects so there is no possible containment violation. (See table 2.1)

The synchronous message-sending used by passive objects is not reconcilable with the non-blocking communication characteristic for AmI programming languages, derived in section 2.2.2. Therefore active objects are the unit of distribution in AmbientTalk. This means that active objects are the basis for concurrent as well as distributed programming. This implies that Applications written in

AmbientTalk are conceived as suites of active objects deployed on multiple autonomous devices.

Passive Objects

Following the design of standard object-oriented languages, passive objects in AmbientTalk are implemented as a set of slots mapping names to objects and methods. These passive objects are created by using the `object(...)` form. The state of the object is the resulting environment of the evaluation of the expression passed as argument. Since environments in AmbientTalk are lexically scoped, so are passive objects. This has as effect that variables in the surrounding environment can be used inside the passive object. Fields can be mutable and private (declared with `:`) or immutable and public (declared with `::`). Both field selection and synchronous method invocation use the dot notation. When selecting a method, without invoking it, this will return a first-class closure encapsulating the receiver-object, which can be called as any other function. The code below shows an example of the use of passive objects. It is a code fragment which creates the abstract data type "queue".

Listing 2.1: Passive Objects in AmbientTalk

```
1  makeQueue(): object({
2
3    queue:[[]];
4    tail:queue;
5
6    enqueue(element)::{
7      tail[size(tail)]:=[element,tail[size(tail)]];
8      tail:=tail[size(tail)]
9    };
10
11   dequeue()::{
12     if(queue=[[]],
13         error("Queue is empty"),
14         {result:queue[1,1];
15          queue[1]:=queue[1,2];
16          if(queue=[[]],tail:=queue);
17          result})
18   }
19 })
```

As illustrated by `makeQueue()`, objects can be created in the body of functions. Such a function will be referred to as a constructor function. It is Ambient-

Talk's equivalent to the object instantiation role of classes. Objects can also be created by extending existing objects. This by the use of the form `extend(parent,...)`. It creates a new object, whose parent is `parent`, and whose additional fields are defined by the evaluation of the second argument. Messages not understood by the child object, will automatically be delegated to the parent object. Manual delegation is also available by the use of the `super` keyword. By use of the `this` keyword, most specific methods for an object can be found from within parent-objects. Finally passive objects can also be cloned. For this purpose AmbientTalk features a special type of methods called *cloning methods*. An object which would implement a method with the name `cloning.methodname(parameters)`, could be sent the message `object.methodname(arguments)`. AmbientTalk will respond to this by copying the original object and executing the body of the *cloning method* as initialization in the newly created instance.

Next to passive objects, AmbientTalk also features the standard built-in elements: numbers, strings, a null value `void` and native functions.

Active Objects

Active objects in AmbientTalk, as explained in section 2.3.1, have their own messages queues and an interminable thread which processes incoming messages one by one by invoking corresponding methods. These actors, as active objects in AmbientTalk are also called, only have one thread, such that state changes by the assignment operator `:=` doesn't cause race-conditions. Actors are created by use of the form `actor(object)`, in which `object` is the object specifying the behaviour of the newly created actor. To obey the *containment principle* stated in section 2.3.1, the object passed as argument is deep copied into the new actor. Otherwise the object would be shared by the newly created actor and the actor which created the new actor, hence allowing race conditions once more. After creating an actor, the first message it will automatically receive is the `init()` message. This message will be used to initialize the actor. Equivalent to the `this` keyword for passive objects, actors can refer to themselves by the use of the `thisActor` keyword. The code below exemplifies the use of actors. The code implements an actor which holds a frame with a button. Messages `show()` and `hide()` can be sent to the actor to show or hide the frame in the actor.

Listing 2.2: Active Objects in AmbientTalk

```
1  createFrame(): actor(object({
2    frame: void;
3    button: void;
4
5    init()::{ display("Initializing frame-actor",eoln);
6              frame:=Frame("New Frame");
7              button:=Button("New Button");
8              frame.add(button);
9              thisActor()#show()
10           };
11
12   show::{ frame.show };
13   hide::{ frame.hide }
14 }))
```

An actor can be sent asynchronous messages by using the # primitive, in the same fashion as the dot-operator is used for passive objects. Using the dot-operator on actors is of course considered an illegal operation, since all message passing to active objects needs to happen asynchronously. When selecting a message of an actor, without invoking it, this will yield a first-class message encapsulating the sender, the receiver and the message name. On passing objects as arguments of a message to an actor, those objects are crossing the boundaries of the actor containing the object. As a logic consequence, to preserve the containment principle the object must be passed by copy, as explained in the *parameter passing principle* in section 2.3.1.

Because the internal design of actors in AmbientTalk is important for further reading of this dissertation, we will go further into detail upon the implementation. Internally actors are represented by two different types of Java objects. The first one, `AGLocalActor` represents the actual actor with all its internals as visible for programmers who write programs in AmbientTalk. It implements an actor which is available on the local device. The second type is the `AGRemoteActor`. This is the local representation for an actor which is remotely available. When passing an active object as a parameter over the network, the *WriteReplace* mechanism of Java will automatically take care of sending out an `AGRemoteActor` pointing to the device where the `AGLocalActor` actually resides. This `AGRemoteActor` is polymorph to its local version, but instead of processing the messages itself, it will forward them to the corresponding `AGLocalActor`.

2.3.2 First-Class Mailboxes

The AmbientTalk model presented until now satisfies two of the four language constraints stated in section 2.2. To support the other two, AmbientTalk extends the standard actor model with eight first-class mailboxes.

Reified Communication Traces

The first four mailboxes are used to support reified communication traces of section 2.2.3. The single message queue is replaced by mailboxes representing the four different states in which a message can be: an outgoing message not received yet, an outgoing message known to be received, a received but unprocessed message and an incoming message which has already been processed. Every actor has access to its mailboxes through their identifiers `outbox`, `sentbox`, `inbox` and `rcvbox`

Ambient Acquaintance Management

In order to comply with the required Ambient Acquaintance Management presented in section 2.2.4, the next four mailboxes are: `providedbox`, `requiredbox`, `joinedbox` and `disjoinedbox`. An actor that wants to make itself available for collaboration can add a description for the service it provides in the `providedbox`. Actors requiring services can add descriptions for requested services in the `requiredbox`. When two actors, one providing a service, and the other requiring the same service, come into communication range, a *resolution object* containing the service-description and a reference to the providing actor will be added to the `joinedbox` of the requesting actor. Conversely, when two previously joined actors move out of communication range, the resolution is moved from the `joinedbox` to the `disjoinedbox`. A programmer can add observers to these boxes, so actors are capable of reacting to changes in their environment.

2.3.3 AmbientTalk with Language Symbiosis

For the validation of this thesis, we do not use the standard version of AmbientTalk. Instead we use a version that was extended with language symbiosis with Java, for this dissertation[2]. By using the extended version we are able to create graphical user interfaces etc. from within AmbientTalk. In the following part we will briefly explain how this symbiosis-system works.

[2]For a more complete technical overview about the implementation of symbiosis in AmbientTalk see [Ver06]

Fall-back to Java

Upon referring to a non-existing variable in an AmbientTalk environment, the variable lookup will fall-back to the symbiosis-system. It will check if there is a Java-package with the same name as the referred variable name. If such a package exists, it will be returned to AmbientTalk as a reference. By calling the dot-operator on packages, we can access sub-packages or classes in a package. In the same way we can also refer to methods of classes. New instances of Java-classes can be created by invoking class constructors as if they were mere AmbientTalk functions.

Like any ordinary Java method, class constructors can take arguments. This implies the need for a conversion mechanism between Java and AmbientTalk values. In this way we can pass AmbientTalk values to Java methods, as they were Java values.

Because Java supports overloading, the system features a method resolution algorithm which will automatically find most specific method for the given arguments, if any. Because this algorithm might be slow for highly overloaded methods, you are also able to guide this system.

Variables are first looked up in the AmbientTalk environment before looking into the symbiosis-system, so AmbientTalk can mask certain Java packages by defining AmbientTalk variables with the same name as the masked Java package.

Interfaces

Some Java-methods do not just take Java-values as argument, but also anonymous classes implementing a requested interface. This is mostly used to insert functionality into user interface components, known as *listeners*. To accomplish this from within AmbientTalk, the symbiosis-system was extended with a mechanism in which AmbientTalk actors can be passed as anonymous class. Internally all calls to the anonymous class will be converted into messages which are sent asynchronously to the given actor.

Semantic Grid

The main semantical extensions implied by adding the symbiosis-system to AmbientTalk can be summarized in the following grid:

	Package	**Class**	**Object**	**Method**
a	root package reference	class reference	object reference	method reference
a()		class instantiation		invocation
a[]				signature selection
a.b	package traversal class reference	static member reference	member reference	
a.b()	class instantiation	static method invocation	method invocation	
a.b[]		static signature selection	signature selection	

All combinations not listed in this grid have no semantics and are thus erroneous.

Next we will give an example of how the symbiosis-system can be used in AmbientTalk. The code creates a *Java AWT* frame with a blue background colour and a height and width of one hundred.

Listing 2.3: Example of Symbiosis Code in AmbientTalk

```
1  {
2    listener:actor(object({
3      actionPerformed(e)::{
4        display("Button Clicked",eoln)
5      }
6    }));
7
8    f:java.awt.Frame("Test");
9    f.setBackground(java.awt.Color.blue);
10   f.setSize(100,100);
11
12   b:java.awt.Button("Button");
13   b.addActionListener(listener);
14
15   f.add(b);
16   f.show()
17 }
```

2.4 Summary

This chapter introduced ambient-oriented programming. We identified the problem in object-oriented programming languages. Then we showed how ambient-oriented programming languages can help in the automation of communication-

specific code and what should be the minimum requirements for a language to be called AmI. Then we introduced an AmI language called AmbientTalk. We pointed out how AmbientTalk fulfills the requirements for AmI languages. We chose this paradigm and language because of its design characteristics which match the reasons for using code mobility. The design of the paradigm based on the type of network makes the paradigm ideally fit for moving around objects from device to device. The language AmbientTalk has even the extra advantage of being explicitely designed to experiment with new language constructs. That is why we will use this language in the next chapters as growing ground for a strong mobility implementation.

3 Mobile Applications in AmbientTalk

This chapter introduces *mobile computations* in AmbientTalk. By mobile computations we mean the ability to start a computation on a site, suspend the execution of the computation at some point, migrate the computation to a remote site and resume its execution there [Car99].

In this chapter we are first going to introduce the computational concepts which are considered when talking about mobility, as well as the different flavours of mobility. Next we give an overview of object-oriented languages in which *strong mobility* is incorporated. As we will see, programming language design for mobility is pretty much unexplored. Existing language proposals mostly boil down to the addition of a *move* operator to the language. This implies that those languages only concentrate on moving single active objects from one host to another. Finally we are going to extend AmbientTalk with strong mobility using move-methods.

3.1 Reasons for Code Mobility

Before we start explaining how code mobility should be handled, we will show why we need code mobility in the first place. In this section we will identify three main sources for mobility.

3.1.1 Intentional Mobility

By *intentional mobility* we mean that code mobility is explicitly requested by the program. This means that code mobility is a part of the goals of the designer of the code. The designer can have several reasons to do so:

- **Resource optimization.** This is the classic load balancing case for mobility. At some point in time, two active objects in the system might be generating so much network traffic that it would be more interesting to move one of the objects to the other object's location.
- **Software engineering.** Mobility might also be caused by design considerations. Designers of mobile applications might consider some objects to "logically belong together" even though there are no quantitatively measurable reasons for allocating them to a certain machine. This might mean that these active objects have to "move along" with another object.
- **Identity preservation.** If we want an object in a system to be a singleton, we will want local references to the object to be redirected to the moved object on the new host, after sending it over the network. By doing this, at all times we only have one object that is the "real one". An example of such objects which are necessarily entities is electronic money. Obviously this money should not be copyable.

3.1.2 Distribution-Caused Mobility

When passing around passive objects in a distributed systems, they will be copied from host to host. So when we use a passive object received as a parameter, we know that that object is local. On the other hand if an active object is passed as parameter, standard behaviour states that it will be passed as reference to the actual object residing on some device. When an active object is now received as a parameter, the object could be local as well as remote. It could be possible that the programmer explicitly wants the active object to be locally available after it is being passed as an argument. This while the passed object could even still have a full message queue. To make this possible we need mobility to be able to move the active object from one device to another.

3.1.3 Mobility for Mobile Computing

A last source of mobility is particularly relevant with respect to *ambient-oriented programming* as explained in chapter 2. Here we combine the differences between *mobile computation* (software entities that roam networks) and *mobile computing*

(hardware that roam physical locations) [AFV98]. Consider for example a PDA on which someone is writing an email using a web browser showing a free online email-service. When arriving at a location with a personal computer, with a better keyboard for example, one could want to be able to transfer the running application to the personal computer to continue writing more efficiently. This is an example to illustrate that mobile hardware has a high probability of generating the need for mobile software in the near future.

3.2 Choosing Code Mobility

This section is burdened with choosing the type of mobility. It will explain the differences between the different available types, and point out why we prefer one over another.

3.2.1 The Computational Context

In order to distinguish between the different kinds of mobility it is useful to consider the *computational context* [Meu04] of an actor. The computational context is the knowledge a language processor has about the actor it is executing. This computational context is divided into *data context*, a *control context* and a *resource context*. This distinction helps us to classify the different kinds of mobility.

The *data context* or *state* of a running actor consists of the variable bindings that are accessible by and allocated for the actor at that moment in time.

The *control context* of a running actor consists of the status of the computation. This information usually takes the form of the combination of a reference to a point in the code (like a program counter or a "current expression") and a description of those past states of computation that are still relevant to determine the future states of the computation (which typically takes the form of a runtime stack or a continuation).

The *resource context* of an actor consists of the bindings of the actor which are not used by the actor alone. These bindings are bindings in the internal memory of the operating system (e.g. libraries) as well as bindings to external resources (e.g. a database).

3.2.2 Kinds of Mobility

Based on the different types of contexts specified in the previous section, four different types of mobility can be identified. These four different types are summarized in table 3.1. However, since only *strong mobility* and *weak mobility* are

	data context	control context	resources context
weak mobility	✖	✖	✖
semi strong mobility	✔	✖	✖
strong mobility	✔	✔	✖
full mobility	✔	✔	✔

Table 3.1: Types of Mobility

interesting and opportune [Meu04], we are only going to go into detail on those two.

- **Weak Mobility** means that "dead code" is able to travel over the network. The code is considered to be dead because no contextual information whatsoever is provided. This means that when the code arrives on its new destination, it will start running as it never ran before. There are two ways to initiate weak mobility: the sending device can send code to a receiving device (called "remote evaluation") or a receiving device might undertake the initiative and ask code to be downloaded from a sending device (called "code on demand").

  Most people know weak mobility because of its popularity due to Java applets. Web browsers can download stateless Java code in the form of an applet, and run it on the client that initiated the download.

- **Strong Mobility** occurs when actors have the ability to move from one device to another, taking into account both the data context and the control context. It allows a running process to migrate without manually having to halt or restore the computation it is performing.

To write full mobile programs, strong mobility is preferable over weak mobility [DHH01]. Weak mobility is easier to incorporate in a programming language, but programs using it will be written in an "unnatural" non-modular programming style. It will also be more difficult to reason about such programs, thus making them harder to debug. This results from the fact that you can only send dead code over the network. To move specific parts of code, you would manually need to prepare the dead code from the runtime state. This means that algorithms would have to be explicitly divided along the mobility lines.

3.2.3 Move Methods

When integrating strong mobility into programming languages, the next decision to make is how to perform the move from one device to another. Different ways

to implement mobility have been presented in the past. We consider three technologies [AFV98].

- **Pull** technology is also called "code on demand" or "fetch technology". The idea is that code is download from one machine to another, to be executed on the downloading device. The initiator for the code movement is the downloading machine. The classical example for pull mobility are Java applets.
- **Push** technology is the second kind, which is also called "ship technology". In this type of systems, the initiator is the sending machine rather than the receiving one. The receiving device will execute the code, and possibly send the result of execution back to the initiator.
- **Agent** technology is the last type. In systems based on agent mobility, the running processes themselves decide to move from one device to another.

Since we are moving running processes instead of "dead code" as is the case in weak mobility, it is mostly *agent technology* which appears to be closely linked to strong mobility. Now to implement a system based on this agent technology, it is important to consider the three parties involved in moving active objects:

- The **Sending Device** is the device or program which actually sends the object to another location. Since it is the device containing the object that will be sent, it can choose how much of the object graph to send to the other device.
- The **Moved Object** itself does not only involve a designated object, but also, in cooperation with the sending device, the full object graph which has to be sent. Together with the receiver it will need to make sure that an amount of resources of the object are rebound, recreated, etc. on arrival.
- The **Receiving Device** is the device which will accept the moved object. It will need to provide the object with access to local resources, such that the moved object can rebind to them.

In order to exclude security violations stated in the dissertation by De Meuter [Meu04], which seem to be inherent in the standard mobile agent approach, both the **receiving device** as the **moved object** will need to take part in the decision to move the object. So neither objects should be able to push themselves to other devices, nor remote devices should be able to pull objects. Mobility should be based on a *two-party* contract between objects or programs which are going to move, and objects or programs on the receiving device, requesting the object to move.

	Free **Transferable**	**Fixed** **Transferable**	**Fixed** **Not Transferable**
identifier	Move (Network reference)	Network reference	Network reference
value	Copy (Move, Network reference)	Copy (Network reference)	Network reference
type	Rebind (Copy, Move, Network reference)	Rebind (Copy, Network reference)	Rebind (Network reference)

Table 3.2: Bindings, Resources and Data Space Management Mechanisms

In the message-oriented mobility model, actors can declare that they are allowed to be moved by other actors by implementing a new type of methods, called *move methods*. An actor on a receiving device can then request the moveable actor to move to its location by sending a *move message*. By sending the *move message*, the receiving actor actually pulls the moveable object to its location. The moveable object will then be moved while processing the message.

3.2.4 Data Space Management

A last aspect of mobility, but a very important one, too often neglected in papers discussing strong mobility, is the *data space management* of the moved object [AFV98] (i.e., rearranging the set of bindings to resources accessible by the moved object). This is why, as stated in the introduction of this chapter, we consider most mobility implementations to be immature. When arriving on a new location, there are several different possibilities for what should happen with the data members of the moving object. This may be voiding bindings, re-establishing new bindings or even migrating some resources to its new location. The choice should obviously depend on the nature of the resources, the type of the binding to the resource and the requirements posed by the application.

Fuggetta models resources as a triple $Resource =< I, V, T >$ where I is a unique identifier, V is the value of the resource and T is the type, which determines the structure of the information contained in the resource as well as its interface. The type of the resource determines also whether, in principle, it can be migrated over the network or not. Of course this is not only defined by "hard constraints" on the resources. While a database for example, could be migrated over the network, it would not be feasible to do so.

Next Fuggetta states that there are three forms of bindings which should constrain the data space management mechanisms that can be exploited upon migra-

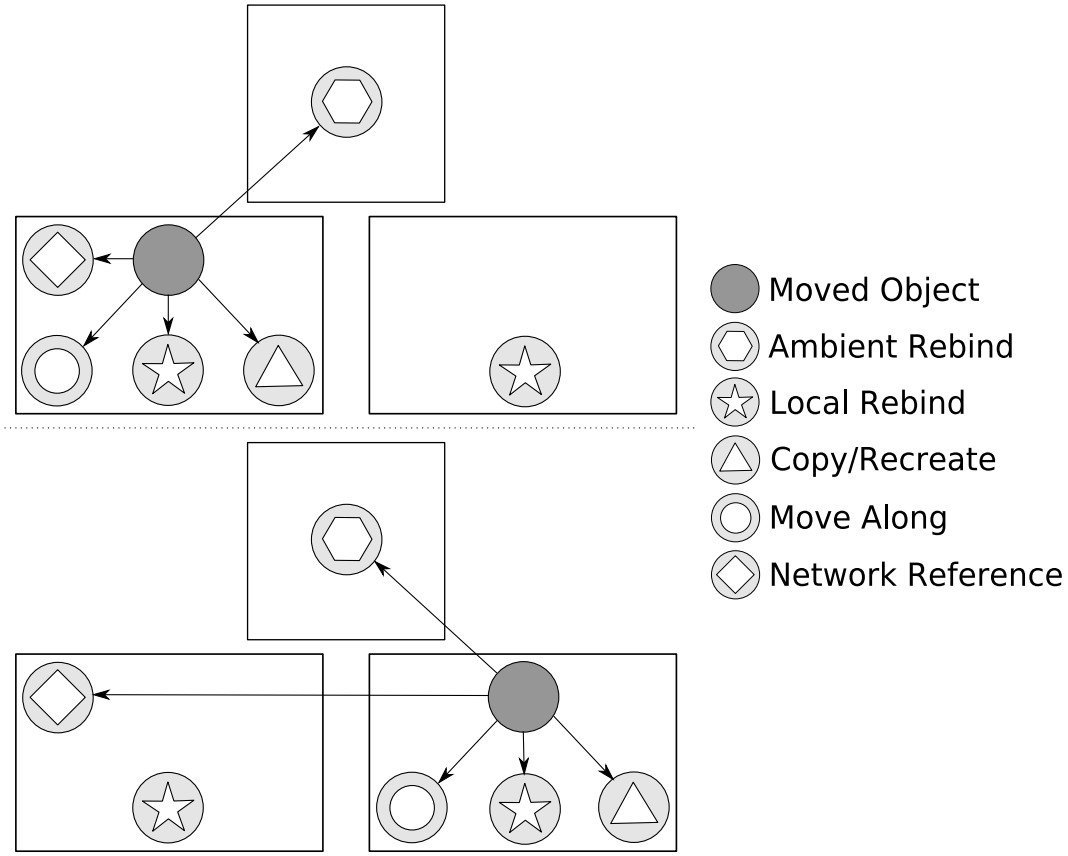

Figure 3.1: Resource Rebinding

tion.

- By **identifier** is the strongest type of binding. It means that the binding is required to be linked to a specific resource, which can not be substituted by some other equivalent resource.

- By **value** declares that, at any moment, the resource must be compliant with a given type, and its value can not change as a consequence of migration. This kind of binding is used when an actor is interested in the *contents* of a resource rather than the actual resource itself. An actor will probably want to access the resource locally after moving of a resource rather than the identity. An actor will probably want to access the resource locally after moving.

- By **type** is the weakest form of resource binding. In this case an actor is not interested in the identity or value of a resource, but rather in the type. This

could be used for example to bind to a local set of speakers, or a display. Secondly, it could also be used for resources remotely available on the new location, like a network printer.

From this list of bindings, combined with the type of resource linked by the binding, we can distil table 3.2 specifying what should happen with bindings after moving an object. If we would for example have a resource bound by type, which is *free* and thus *transferable*, most common behaviour will be to *rebind* to a local resource of the same type. Of course we could also keep a *network reference* to the resource we were using before the move. We could also *copy* or *move* such a resource to the new device. An alternative to copying a resource could be *recreating* the resource. Which of these possibilities mechanisms is going to be used, should be specified by the programmer.

As default behaviour it might be best to turn resource references into *network references*, since this mechanism is a possibility for all resources and can thus be used without knowing the type of resource or the type of binding.

What could happen to any resource-binding based on table 3.2 without looking to type of binding or type of resource, can be seen in figure 3.1. There are five different scenarios possible. The first is the default behaviour, a binding to a resource which is turned into a network reference. This could be useful for a database which we do not want to move. The second specifies rebinding locally to a resource, which could be used for a set of speakers. The third is related closely to the second and is about remote rebinding. This comes down to a resource bound by type, which is not available locally. Hence it will be rebound remote. This could be useful for a printer, which might not be installed on the working device, but on a device available on the network. As fourth we can have a resource which we want to move along with the moving object, for example for a library used only by the moving application. The last possible behaviour is copying or recreating a resource, which would be needed to rebind to a local logfile for example.

3.3 Languages with Strong Mobility

In this section we are going to take a look at how current programming languages incorporate strong mobility. Of course we especially look closely to the use of a *data space management system* in these languages. Since they all use strong mobility, they all have a *unit of mobility*, which will also be identified for all languages.

- **Ara** has got moveable agents. These agents are unable to share resources, except for system resources. These system resources are removed on move. [PS97]

- **Agent Tcl** allows programmers to move a thread from one device to another. Since threads on one device do not share data-space, the only resources they can share are system resources. On moving a thread, these resources are removed. [Gra95]

- **Obliq** as stated by De Meuter [Meu04] is arguably a mobile programming language. It has no explicit mobility operators. But with some effort it is possible to get objects to move themselves. As for data space management, all objects which are passed over the network are seen as fixed transferable resources, so they can be copied if necessary. As default behaviour they are turned into network references.

- **Emerald** can move active objects from one device to another. As default behaviour all resources used by this object are turned into network references. Because it might be useful to move multiple objects together, Emerald allows programmers to attach objects to each other. This attachment-system is transitive. [EJB88]

- **Telescript** has got moveable agents. Telescript has an ownership-system to make it possible to define owners of agents. When arriving on a new location, resources owned by the moved agent are moved along. All other resource-bindings are automatically removed. [Whi96]

As should be noticed, none of these systems provide a solution for all possible data space management mechanisms.

3.4 Mobile AmbientTalk

After making different mobility design choices in the previous sections, we are going to use them to extend AmbientTalk with a strong mobility implementation. This implementation will be used in the next chapter as starting point for our mobility extensions.

3.4.1 Actors as Unit for Mobility

In AmbientTalk, as stated in section 2.3.1, actors are the objects with a thread running through them. Because of their asynchronous message passing they are also the unit of distribution. Passive objects can be passed around from one active object to another, but this always happens by copy. Active objects on the other hand are passed by reference. When an active object is passed as parameter from one active object on a device, to another active object on another device, this is

also done by reference. This clearly indicates that if we apply strong mobility to AmbientTalk, this should be done on active objects (or actors as they are called in AmbientTalk).

3.4.2 Move Methods

Concluding from the choice made in section 3.2.3, we are going to add a new type of methods to AmbientTalk actors, the *move-methods*. To be conform with other special methods in AmbientTalk, like the cloning-methods discussed in section 2.3.1, we are going to use a similar syntax for move-methods. For a method of an active object to be a move-method, it has to be prefixed with `move.`. For example, we could now define the method `move.come(resources)::{...}` on actor `A1`. If an actor `A2` on a remote device would send the message `A1#come(r)`, the actor `A1` would move to the device on which `A2` is residing.

Before an actor moves from one device to another, it should be able to do some *preprocessing* on itself. This because the actor could have open file-descriptors, windows, etc. on the sending device, which it needs to finalize before moving away. For this reason we definitely include a pre-move part in the move-method. It would not be a good reason to allow an actor to send messages to other actors in this part of the move-method. If the actor would send a `move.*(...)` to another actor, this could lead to unexpected results. At the time of processing the `move.*(...)`, the sending actor, which is moving too, could still be residing on its old location, but could just as well be moved to its new device. This results in non-determinism on the new location of the receiving actor. To prevent this situation from happening, we freeze the outboxes during the pre-move-method. This means that messages can still be added, but they will only be sent after arriving on the new device.

Right after moving an actor, it should also be able to do some *post processing*. The reason is that the actor should be able to reconfigure itself, integrating in its new environment. For example, if the moving actor has closed a window, a file-descriptor, etc. on his old location, the odds are good that the actor wants to re-open it on the new location.

Since the move-method should be divided into two parts, a *pre move* and a *post move* part, we are going to extend the environment of the move-method with a `moveNow()` method. This method can only be called from within a move-method, and can only be called once in the move-method. All code before the `moveNow()` call will be executed on the sending device. All code after the call will be executed on the receiving device. If no `moveNow()` call is added to the body of a move-method, all code of the move-method will be executed on the sending device, and the `moveNow()` will automatically be called at the end of the body. The next piece of code is an example of a “frame-actor” implementing

a move-method:

Listing 3.1: A Mobile AmbientTalk Actor

```
1  A1: actor(object({
2     f: nil;
3
4     init():{
5        f:=Frame("A Frame");
6        f.show()
7     };
8
9     move.come():{
10       f.hide();
11       moveNow();
12       f.show()
13    }
14 }))
```

3.4.3 Implementation of Move Methods

Unifying Remote and Local Actor Address

Of all actors, there should only be one instance. As explained in section 2.3.1, for every actor there is a `AGLocalActor` object on the device where the actor actually resides. On the other devices there is a `AGRemoteActor` object which links to the actual position of the actor. To ensure that all internal pointers to these objects will always point to the current state of the actor on a device (local or remote), we add an extra abstraction layer to actors, called `AGActorAddress`. An `AGActorAddress` is a container for an `AGLocalActor` or an `AGRemoteActor`. In this way we can easily move actors from one device to another. We just have to replace the `AGRemoteActor` on the receiving device by the `AGLocalActor` on the sending device. On the sending device we replace the `AGLocalActor` by a new `AGRemoteActor` pointing to the new device.

Move Process

As shown in figure 3.2, the moving of an active object happens in steps, to ensure that the actor always stays available for communication. In the figure we have three devices with three local representations for an actor, originally residing in `A`. `B` as well as `C` are `AGRemoteActor` references to `A`. When an actor on the device where `B` lives, sends a move-message to the actor, which calls `moveNow()`, a

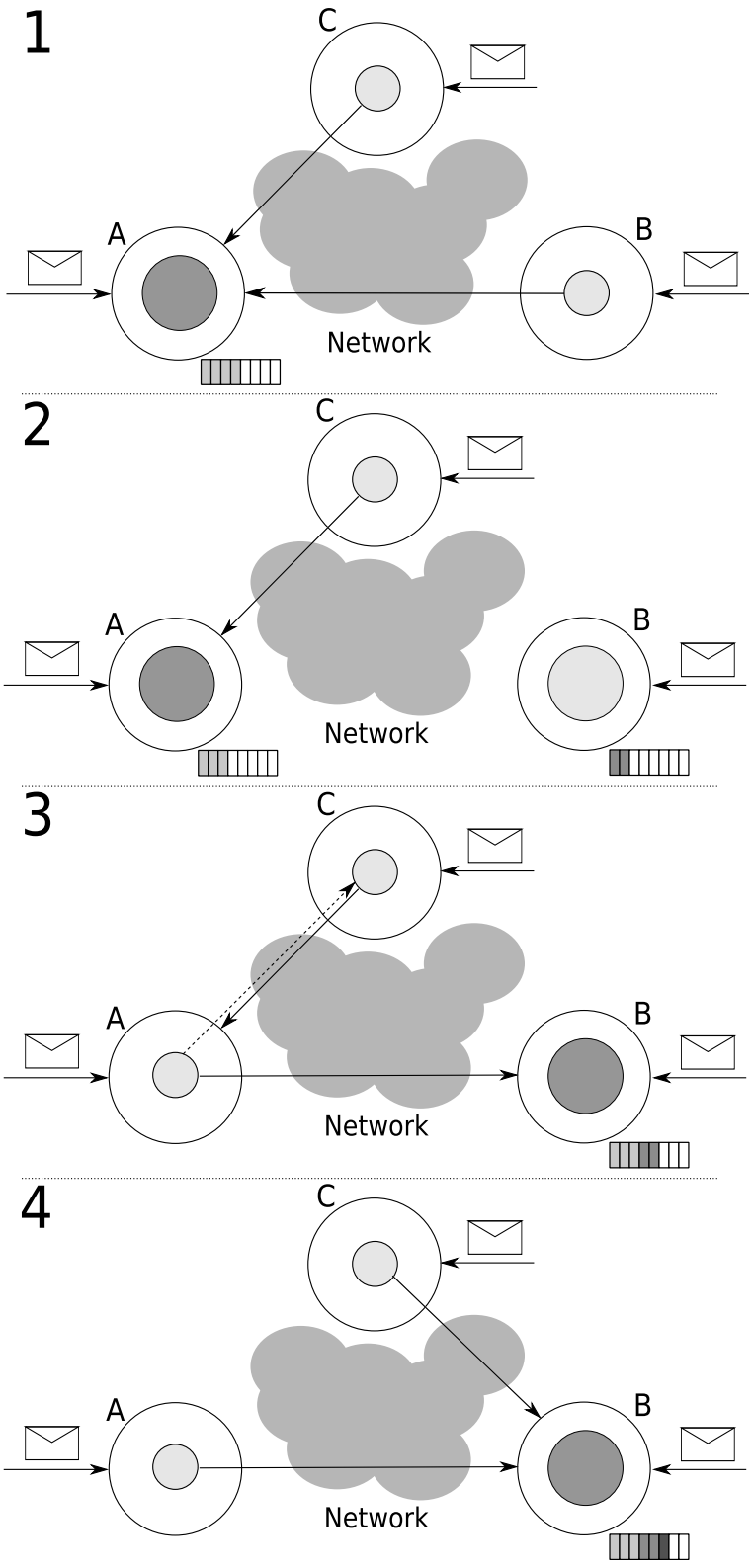

Figure 3.2: Strong Mobility in AmbientTalk

copy of the AGLocalActor in its current state (behaviour, continuation-frames) will be put in the AGActorAddress B. B will get an own message queue, so it can start receiving messages locally. The new AGLocalActor in B is not activated yet. In the next step (image 2 to 3), the old AGLocalActor will be replaced by the new AGRemoteActor reference pointing to the new location, this after sending the contents of the message-queue on the old location to the new AGLocalActor. These messages will of course be added to the new message-queue in front of the messages received on the new device. Then the new AGLocaActor is re-activated.

The final step in moving an actor is updating all references to the actor on other devices than the sending or receiving device. When sending a message to an AGRemoteActor like C, this message should directly be forwarded to the correct device. The receiving device thus needs to have a AGLocalActor in the AGActorAddress. If this is not the case, this means that the actor has moved to the AGRemoteActor in the receiving device (from A to B). In this case the receiving devices informs the sending device about the new location of the actor. The sending device C will then update its local AGActorAddress with the new AGRemoteActor and will try to send the message again following the new AGRemoteActor (to B).

Race Conditions on Mobile Actors

An import security aspect of the implementation of mobility are race conditions on mobility. The problem is presented in figure 3.3. Imagine we would have three actors on three different devices as in part one. If both actor A and actor C would request for B to move to their location, we could end up in the three different scenarios presented in part two, three and four. In the third and the fourth part, we have the scenarios were A or C requested the move a little bit earlier. This results is the second request to be forwarded to the new location of B (the device of the actor who requested the move first). As a result the actor will be moved to the location of the actor who requested the move a little bit later, where the actor will stay until further notice. These scenarios are not problematic situations from language engineering point of view, since there is no change in the program environment because of the subsequent moves.

The scenario in part two however is critical for the program logic. In this scenario two actors have requested a move right at the same time. Because actors run different threads, as race condition it could occur that multiple actors ask one actor to move at the same time to different locations. If the moving actor would respond directly to the call, we could end up with multiple versions of the moved actor on different locations. This scenario is automatically avoided in the AmbientTalk implementation of mobility by using *move methods* (and corre-

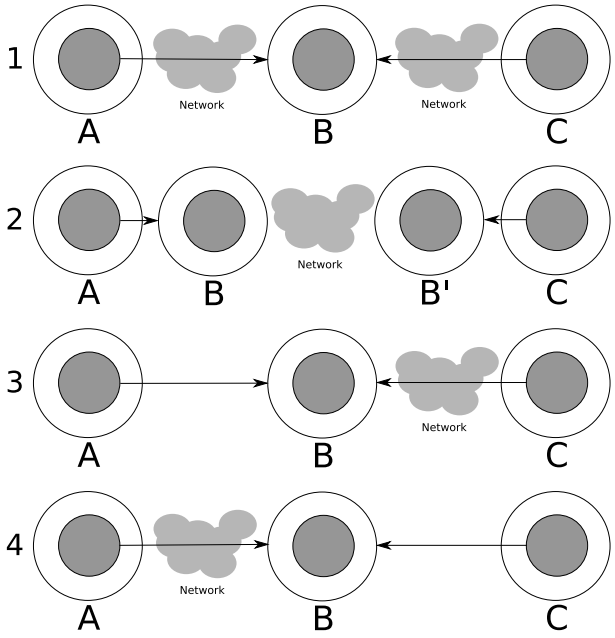

Figure 3.3: Race Conditions in Mobility

sponding *move messages*). If an actor requests another one to move, it must send a *move message* to the other actor, which will be put in the inbox of this other actor. The actor which will be requested to move handles all its messages one by one. Since they are handled one by one, an actor can only respond to one move request at a time. So even if we would have multiple move requests at the same time, we would always end up in scenario three or four.

3.4.4 Data Space Management

Until now we are not going to define special data space management semantics on the strong mobility system of AmbientTalk. As stated in 3.2.4, it is feasible to turn all resources into network references after a move, since this behaviour is a correct solution for all resource-types and all types of bindings to resources. Since AmbientTalk already automatically turns active objects, after passing them over the network, into network references, as pointed out in section 2.3.1, this goal is automatically achieved by just passing the behaviour of the `AGLocalActor` over the network. Because passive objects in AmbientTalk are passed by copy, these will also be copied to the new `AGLocalActor` when copying the behaviour.

3.5 Summary

In this chapter we introduced code mobility. Then we specified how *strong mobility* should be implemented, by defining all necessary characteristics. We looked closer to how the resources of a moving application should be handled, and how existing mobility implementations handle this. Finally we defined a strong mobility system for AmbientTalk based on the characteristics. In the current chapter we only focussed on incoming references[1]. This implementation will be used as basis for the next chapter, where we will extend the current model to allow programmers to perform proper data space management. This means we will also focus on outgoing references in that chapter.

[1]Incoming references: external objects referring to the moved object

4

Declarative Field Annotations in Mobile AmbientTalk

In this chapter we are going to extend the AmbientTalk model with strong mobility, prepared in section 3.4, with a system to define mobility semantics for full applications. We start from the observations about the *data space management* in section 3.2.4. We are going to specify a solution in the first part of the chapter and then we will go further into detail upon the implementation in the second.

4.1 Declarative Field Annotations

When we discussed the implementation for strong mobility in AmbientTalk, we fulfilled the task of updating all incoming pointers to the moving object. Since incoming pointers always should point to the actual object, the implementation is pretty straightforward and the update can be done automatically by the system without any further knowledge of the semantics of the program and without affecting these semantics. Updating outgoing pointers on the other hand, pointers to other active objects which are the resources for the moving object, can not be done in an automated way, since the location of resources partly defines the semantics of the application.

In section 3.4.4, we specified that resource references as default behaviour should automatically be turned into network references. Passive objects on the other hand should be passed by copy. Now programmers should be able to easily

change this default behaviour. Without special language constructs, programmers are forced to write large move methods specifying explicitly in which way objects should be changed during the move. Even worse, if the mechanism for a resource would have to change during the lifetime of the active object, the resulting code would probably be cluttered with if-else-branches. To avoid this, we suggest extending the language definition with *declarative field annotations* declaring on-move semantics on fields.

4.1.1 Dynamic Annotations

A standard way to add field annotations, or a typing system, to a language, is by introducing keywords in the programming language, stating the nature of the annotated fields. Known examples for this way of annotating are the private/public/protected of Java or C++. However, since we are working in a duck typed language [Wik], this kind of statical keywords would break the dynamism of the language. This is already a first reason for a more dynamic solution. The second and more important reason with respect to our research, is that programmers might want to change the way a resource is dealt with while moving, dynamically. Because of these two reasons we opt for the use of a more dynamic solution, adding a number of first-class so-called *typeboxes* to the definition of the language, in the same way as first-class mailboxes were added (section 2.3.2).

These typeboxes are disjunct sets of variable-names, for which the typebox specifies what should happen to the value of the variable, when the active object it would move. If one would prefer to introduce keywords in the language too, this could easily be achieved with typeboxes. The keywords would be nothing but syntactic sugar for the addition of a field to a specific typebox declared by the keyword.

4.1.2 Dynamic Mobility Types

To not only make the declarative annotations for fields dynamic, but also the set of available mobility-types, we suggest the addition of a native to easily create new typeboxes for a specific active object. This native should take two closures as parameter, which specify what should happen to an object annotated with the typebox before and after the actual move. These two closures are required in the line of the design of move methods in section 3.2.3. There we pointed out that some objects might need pre- and post-processing. Since typeboxes will be used to automate mobility, and thus shorten move methods, pre- and post-processing-functions are required for typeboxes too. For example if one uses a lot of Frame-objects in a moving actor, which have to be moved to the new device as part of a

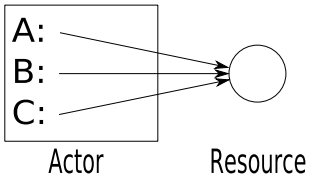

Figure 4.1: Multiple fields using the same resource

moving graphical user interface, one could want to add a typebox in the following way:

Listing 4.1: Typebox for a moving GUI

```
1 framebox:createTypeBox(
2   function(object) → { 'preMove'
3     object.hide()
4   },
5   function(object) → { 'postMove'
6     object.show()
7   }
8 )
```

4.1.3 Disjunct Typeboxes

Typeboxes should be disjunct, since every binding can only be handled by one mobility mechanism. So typeboxes should make sure that every field is only included in one typebox and any time. We suggest this to be the last typebox to which a field was added. So when adding a field to a typebox it has to be removed from all other typeboxes. Secondly we also have the possibility that multiple fields point to the same object or resource while moving. In figure 4.1 we see three fields pointing to a resource. If we would be able to define two different scenarios for fields `A` and `B`, what should happen for binding `C` is undetermined. To prevent this scenario from happening, we define that only one mechanism can be applied to one object or resource for any actor. This will also be by default, the last type applied to any field pointing to the object at move-time.

4.1.4 Default Typeboxes

After adding a way to add new typeboxes to an active object, we are now going to define which typeboxes should be added by default. Thus specifying which mobility annotations should always be available. We claim that there are five default typeboxes necessary. These typeboxes are derived directly from the five

scenarios listed in section 3.2.4 (figure 3.1) and need to represent the following behaviour:

- **defaultTypeBox.** Fields added to the default box will be turned in the *network references* if the value is an active object. The field will be *moved along* if the value is a passive object. On the one hand, this typebox is not necessary, since this behaviour is implied by default mobility policies. On the other hand, this addition might serve as an easy way to switch back from any other kind of behaviour to the default.

- **localRebindTypeBox.** These fields will be overwritten with a binding to a (similar) local resource on the device to which the moving actor is migrating.

- **ambientRebindTypeBox.** This typebox is comparable to the localRebindTypeBox, but in this case the value will be overwritten with a binding to a remote resource.

- **moveAlongTypeBox.** This typebox will request the objects bound by its fields to move along. Since only active objects are subject to message sending and strong mobility, only fields with bindings to these kinds of objects should be added to this box.

- **recreateTypeBox.** Fields added to this box will recreate their value upon arrival on the destination of the moving actor.

4.1.5 Further Typeboxes

One could remark that perhaps also other typeboxes are necessary, typeboxes defining semantics for combinations of actors moving together. However, we state that the current typeboxes are sufficient for standard objects and resources[1], whatever the form of the resource graph is at moving-time. We do this by looking at a case where we have indirect mobility semantics on resources. We claim that all the semantics which can be appointed to the graph, are already automatically imposed by using combinations of the current typeboxes.

The most typical case for a difficult binding graph with indirect pointers is the diamond-shaped graph shown in figure 4.2. Here we have an actor `A` relying on two other actors `B` and `C`. These two actors then rely further on one and the

[1] We do not state anything about objects (mostly passive, but possibly also active objects) needing special treatment before/after the move. For such objects, like the moving frames-actor in example code 4.1, typeboxes are used for very specific needs. This is only a convenient surplus of the createTypeBox function.

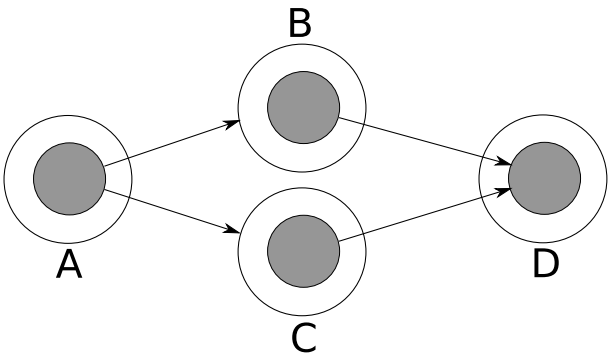

Figure 4.2: Diamond-Shaped Resource Bindings

same actor `D`. The scenario could be related to the problem in section 4.1.3, where we showed that we only want one mobility mechanism to be applied to resource-bindings. However updating resource-references in one actor that moves is not the same as updating resources in multiple actors. The reason lies in the fact that actors are objects which work individually (section 2.3.1). Their contact only happens via asynchronous message sending. This implies that a move of an actor is completely decoupled from the mobility of its fellow actors. Actors can of course request other actors to follow them, but when the other actors will respond to this request depends on the state of their message queue at the time of receiving. This first remark implies that an automatic mechanism to solve these relations is impossible.

The second remark will show that such a mechanism is unnecessary and strong mobility with current typeboxes is as powerful as any programming language would be with the addition of typeboxes for such combined indirect relations.

Any moving application with a shape as shown in figure 4.2 will only have semantical difficulties if `A` moves from one device to another and `A` expects `B` and `C` to move along, but for the bindings to `D` to be changed into something else. This can mean, based on the different mobility scenarios proposed, that `D` has to be recreated or rebound (locally or remote). If `D` is asked to be turned into a *network reference*, or to be moved along, the pointers in `B` as well as `C` will stay linked to one and the same object (the original one) and no special care has to be taken[2]. In that scenario, in the line of what was explained in section 4.1.3, about one actor having multiple bindings to one resource, one could also expect the bindings to `D` from `B` as well as `C` to be redirected to the same new resource replacing `D`. To be

[2]Remark that if D has to be moved along, only one of the actors A,B or C has to send a *move message*. However if in the implementation of *move methods* a *move message* received from an actor on the local device would mean to ignore the *move message*, it would be no problem for multiple actors to request a move along at the same time.

able to answer the question about how to solve this riddle, it is useful to look at how this structure was created in the first place. There are a few different designs for this structure to arise from:

- **Binding by Type:** `B` **and** `C`.
  `B` as well as `C` require a resource of a specific type. "By accident" they are bound to the same resource.

- **Binding by Type:** `A`.
  `A` has required a resource of a specific type and asks `B` as well as `C` to use this resource in their work.

- **Binding by Type:** `B` **or** `C`.
  `B` (or `C`) has acquired a link to a resource of a requested type and has passed this resource to `C` (or `B`) to work with the same resource.

- **Binding by Value:** `B` **and** `C`.
  `B` and `C` have the resource `D` in their local environment and both bind to it. This can be "by accident", or because the programmer intended this to happen. In the last case, there must be some link between `B` and `C` which is available at the time the resource is shared, otherwise the sharing of the same resource has no meaning. This can be a direct link, or a mutual parent-link from an actor as `A` in the figure.

- **Binding by Value:** `A`.
  `A` has a binding to a resource and passes this binding to `B` and `C` to use in further computations.

- **Binding by Value:** `B` **or** `C`.
  `B` (or `C`) has a binding to a resource and passes this binding to `C` (or `B`) to use.

In this list we make a distinction between the intentional and non-intentional diamond-designs. For the design of typeboxes we are only interested in maintaining structures that were intentionally created by the programmer. For this reason we do not consider the first possible origin for the diamond structure. The cases which are considered intentional can be categorized into two different categories:

- **By Communication.** In this category the diamond construction is obtained by passing the resource binding from one actor to another. This can happen in two ways. `A` can be the origin of communication. In this case `A` will pass the binding to `D` to `B` and `C`. The other possibility is that, instead of having an actual diamond structure with no connection from `B` to `C` (or the other

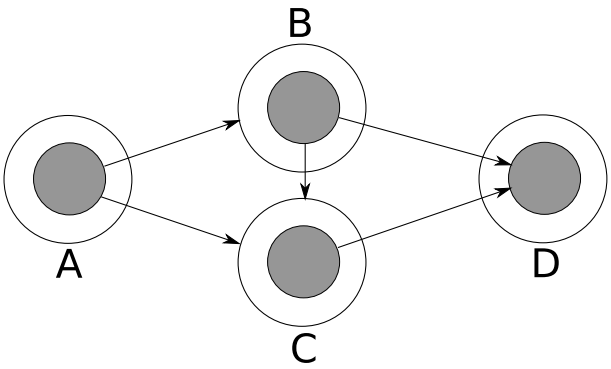

Figure 4.3: Diamond-Shaped Resource Bindings Unfold

way around), we have a structure as presented in figure 4.3. In this situation the binding to `D` is passed from `B` to `C` (or the other way around). We call the actor who originally found or created the resource the *dominating actor*. The actors depending on the *dominating actor* to provide the resource are the *dominated actors*.

In these cases there was always communication between actors from which the structure originated. To move and recreate this structure, the same thing must happen. The dominating actor has to move before the dominated actors. After arrival it has to rebind or recreate the resource. The actor will probably do this by use of the typeboxes. Then the *dominating actor* is responsible for also moving along the *dominated actors*[3] and passing its new resource binding. Since the *dominating actor* sends the move message, by using the typeboxes it can pass along the new binding automatically by adding it to the resources.

While these scenarios are solvable with the normal typeboxes, they do impose an order on the processing of those typeboxes. A programming language using typeboxes must always make sure that typeboxes changing bindings are processed before messages using these bindings are sent to the outside world (before the move along typebox is being used).

- **By Construction.** The only origin of the diamond model which is by construction, is the *Binding by Value: B and C*. This means that a programmer intended for `B` and `C` to point to the resource `D`. To do so he wrote the code in such a way that both `B` and `C` see `D` in their scope. Because both `B` and `C` are also bound in `A`, this must mean that `A` has both `B` and `C` in its scope.

[3]While we would initially think that in a setting as presented, A is going to request both B and C to move, actually A will only ask the dominating actor to move along. This dominating actor (B in the figure) will then ask C to come along.

By combining these two scope constraints, we know that also `B` and `C` must be in the same scope. Now since there is a contact possible between `B` and `C`, this setting can be handled in the same way as the *intentional designs by communication*.

4.2 Extending AmbientTalk with Typeboxes

In this section we are going to extend the version of AmbientTalk in which we already incorporated *strong mobility* in section 3.4 with an experimental implementation of the *typeboxes* presented in the previous section.

4.2.1 Adding Typeboxes

Based on the decision we made in section 4.1.2, we add a function to AmbientTalk which enables us to easily create new typeboxes. This function called `createTypeBox(preMove,postMove)` relies on an object which represents the available typeboxes in an actor, `typeBoxes`. To allow actors to choose themselves if they want to use typeboxes, we add also a native next to, and based on the `moveNow()` which we defined in section 3.4.2, namely `moveNowWithResources(resources)`. The native requires `resources` as parameter, which represents a set of resources which will be available on the device to which the actor is moving. This parameter will be used by the typeboxes for rebinding to local resources. The resources will be passed to the `preMove` and `postMove` closures, which have to accept three parameters: `function(object, resources, config)`. The first parameter is the object which has to be pre- or post-processed. The third parameter is a table containing the precedence-id of the field followed by the extra arguments specified while adding the field to the typebox. Typeboxes can thus require to pass along extra arguments to fulfill there pre- and/or post-move processing.

Listing 4.2: Implementation of moveNowWithResources

```
1 ' When the moveNowWithResources is used, the '
2 ' typeboxes will be processed at the moment  '
3 ' between the pre- and post-move part of the '
4 ' move method                                '
5 moveNowWithResources(resources)::{
6   typeBoxes.preMove(resources);
7   moveNow();
8   typeBoxes.postMove(resources)
9 }
```

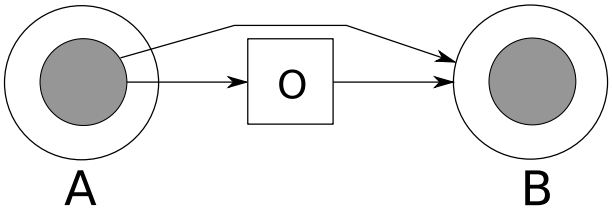

Figure 4.4: An Unchangeable Passive Object Pointing to a Resource

4.2.2 A Design Choice

In AmbientTalk there are two possible ways for implementing typeboxes: in Java, the language in which the AmbientTalk-interpreter is written, or in AmbientTalk-code. This is similar to how Smalltalk is mostly implemented in Smalltalk. To keep the core language as small as possible we preferred to implement the typeboxes in AmbientTalk.

Because of this design choice, in combination with the fact that we need some reflection in AmbientTalk, and access to the the communication layer of Java, we will need to add three native functions to the AmbientTalk interpreter.

The first native we will add is an extension of the reflection of AmbientTalk. `getField(String)` has to be added to get the actual value for `String` defining a field. We add strings as field-description to typeboxes because, as with other declarative annotations, we want to annotate a field, not an (initial) value of a field.

The second and the third native come from the design choice made in section 4.1.3 where we stated that all pointers from an actor to one and the same resource should undergo the same mobility mechanism. This might imply that a binding to a resource is changed by a binding to another resource, which is local on the arriving device. Since AmbientTalk is written in Java, and all of its pointers are Java-pointers to internal objects representing AmbientTalk-objects, we have to be able to redirect a set of Java-pointers from an AmbientTalk-object, to another AmbientTalk object. An example goes as follows and is also represented in figure 4.4:

Listing 4.3: A Passive Object Encapsulating a Resource-Binding

```
1  actor(object({
2    B:void;
3    O:void;
4
5    createLocalResource():{
6      actor(object({
7        print()::display("I am a local resource",eoln)
8      }))
9    };
10
11   init()::{
12     B:=createLocalResource();
13
14     'We encapsulate the resource-link inside the    '
15     'unchangeable object which will be moved along  '
16     O:=object({
17       localBinding:B;
18       do()::localBinding#print()
19     });
20
21     'On moving the resource has to be recreated     '
22     recreateTypeBox.add("B",createLocalResource)
23   };
24
25   move.come(resources)::{
26     moveNowWithResources(resources)
27   }
28
29   do()::{
30     B#print(); O.do()
31   }
32 }))
```

The example shows a forged implementation of an actor `A` with a binding to a resource `B` it created itself. On moving to another device, the actor `A` wants to recreate the resource, for any reason. The actor also has a pointer to an object `O` which holds a binding to the resource `B` himself. In this example, because `A` is going to be recreated, this could lead to two different possible solutions.

- **Without typeboxes.** In the first solution we are going to redirect the resource-pointer `B`, but since we have no way of changing `O` (the object does not pro-

vide any mutators on its local version of the resource-pointer), this object will keep on pointing to the old resource residing on the old device after moving `A`. If we would now send the message `A#do()`, this would result in a display of the `I Am A Local Resource` message on both devices. One initiated by `A` itself, on the new device. The other one on the old device, sent by the passive object `O`. The only way to end up with feasible semantics, is by also recreating `O` on arrival at the new location. Even while in some cases this solution might be "powerful" enough, it will probably fail for some cases too. In any case this style of programming will result in "waterfall code". This means that if we would have an object `Z` pointing to `O`, it is possible that also this object needs to be recreated, and so on.

- **With typeboxes.** The most logical solution would, unlike the solution *without typeboxes*, be to also automatically update the pointer inside the object `O`. Unfortunately this is not possible in plain AmbientTalk code, as we pointed out by this example. Because we want to allow programmers to be able to use this kind of scenarios, without having to worry about the internal workings of updating the pointer, we enrich typeboxes with the capabilities to automatically deal with this situation. In the next part we will show how we do this. Notice that this will make mobility with typeboxes more powerful than "normal" single entity mobility.

In order allow typeboxes to do such kind of "magic", typeboxes in AmbientTalk must be able to access the communication-layer of AmbientTalk. To be more precisely, it needs access to Java's ReadResolve and WriteReplace. These are the only moments when moving objects in Java can redirect all incoming pointers to another object. Because we chose to implement typeboxes in AmbientTalk code, and standard AmbientTalk has got no access to that layer, we need to extend AmbientTalk so typeboxes do have access to it:

- `writeReplace(oldobject, newobject)` will be used for example to change the pointer to a resource of a field, to a similar resource on the new device. The function will annotate the `oldobject` for the moving actor in such a way that when Java is `WriteReplacing` the object, it knows it has to change it by `newobject`.

- `readResolve(closure)` will create a new AmbientTalk-object used to send a description of an object over the network. The description which takes the form of a closure, will be `ReadResolved` into the actual object (by executing the closure) after arriving at the new location.

Since these two natives would allow programmers to circumvent the communication layer of AmbientTalk while message-sending, they are only made accessible from the *pre-move* part of move methods, the part where the preprocessing of typeboxes also takes place. Recall that in this part of the *move method* message sending is not allowed and that this is enforced by freezing the outbox of an actor in this part as discussed in section 3.4.2. Also the fact that `readResolve` generates objects which could be used as parameters in messages, sent in the pre-move part of a move-method, which are stored in the outbox without actually sending them, is no problem. When the outbox of the actor is moved to the new device, before it is reactivated, the arguments will already be ReadResolved into the actual objects.

Notice that if we would implement the typeboxes in plain Java-code, as extension of the core language, these two last natives would be unnecessary. The "magic" could be done automatically by encoding it into the Java implementation.

4.2.3 Default Typeboxes in AmbientTalk

Now we have a way to easily add new typeboxes to AmbientTalk, we add the typeboxes which we specified to be required (section 4.1.4) to AmbientTalk:

- `moveAlongTypeBox.add(String)` will cause a *move message* `object#come(resources)` to be sent to the `object` bound to the field with name `String`. This implementation implies that a standard move method should be implemented by the actor which will be requested to move along.

- `recreateTypeBox.add(String, closure)` will call `closure` on the device to where the actor is moving, and will replace the value of field with name `String` by the result of the evaluation.

- `localRebindTypeBox.add(String, localResourcePattern)` can be used to replace a resource bound at field `String` by a local resource defined by pattern `localResourcePattern`.

- `ambientRebindTypeBox.add(String, ambientResourcePattern)` works in the same way as the `localRebindTypeBox` but will be used for resources which are not necessarily local.

- `defaultTypeBox.add(String)` is a typebox which does no special pre- or post-processing. Adding to this box means removing it from all the other boxes, which implies that the default mobility behaviour is going to be applied to the given resource.

The actual implementation of these typeboxes can be found in appendix A. They can be used as a reference to create new typeboxes.

In section 4.1.5 we mentioned that an order should be imposed on the processing of the typeboxes. This because all rebinding should be finished before we start sending out messages using these reestablished bindings. However, since in section 3.4.2 we chose to freeze the outboxes before moving and because resources are rebound before the unfreezing of these outboxes[4], we do not have to manually encode this order. It is automatically imposed by the prohibition of message sending until after rebinding in general.

4.3 Summary

This chapter focussed on the mobility of full applications instead of the mobility of one actor. Therefore we first identified the five different scenarios for moving interconnected objects. Next we extended the mobility model of chapter 3 with a dynamic declarative field annotation system. This system uses first-class typeboxes, which can be added on-the-fly. For standard mobility situations, the typeboxes are used to annotate fields with one of the five mobility semantics we identified. In the next chapter we are going to use this implementation in AmbientTalk to validate the use of mobility annotations.

[4]This automatically results from the fact that ReadResolve and WriteReplace is used to rebind, as seen in section 4.2.2.

5 Case Study: Mobile Route Planner

Now we have a concrete example of a programming language implementing the proposed declarative field annotations, it is time to validate the solution. This chapter will do so by using this extended version of AmbientTalk to create a moving *TrafficWare* application, based on a scenario requested by the inter-university project CoDaMoS [cod04].

5.1 Scenario

Doctor Healthy is in a hurry to get to a patient. Unfortunately he does not know the route to where she lives. When she called he typed in her address in his *TrafficWare* program on his office computer. *TrafficWare* now uses the global positioning system, GPS, to find out its own position and calculates the shortest route. In the meanwhile Dr. Healthy asks *TrafficWare* to move to the board computer of his car to keep him informed of the route while he is driving. As he starts his car, the board computer automatically gets switched on too. The wireless connection established between the car and his office computer makes it now possible for the *TrafficWare* program to migrate to the board computer of the car, as was requested. On installing itself on the board computer, *TrafficWare* uses the board computer's gestalt manager to reconfigure itself to fit on the screen. Then *TrafficWare* reconnects to GPS to stay updated about the position of the car, so it can

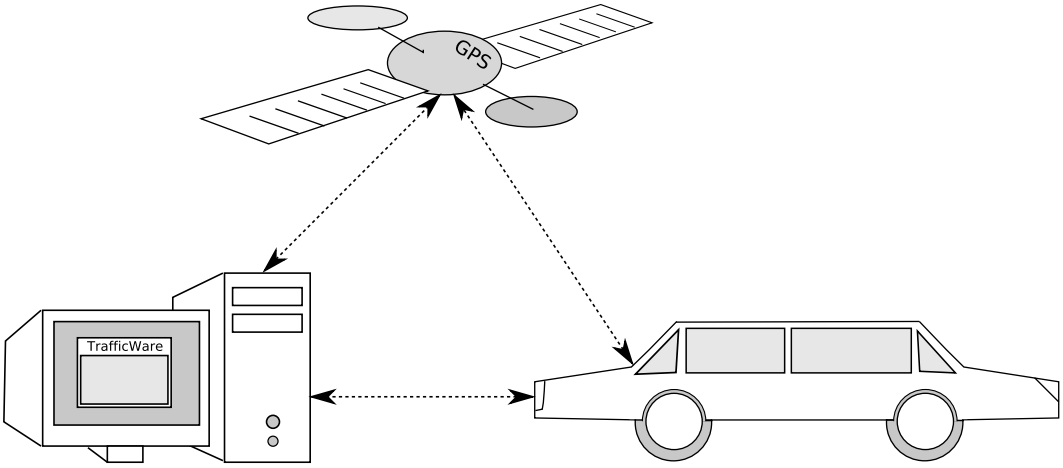

Figure 5.1: A Mobile Route Planner Program

recalculate the itinerary if it appears that doctor Healthy is deviating from the originally calculated one.

On arriving back at the office after the visit, *TrafficWare* informs the "monthly expenses program" running on the office computer about the travelled distance. Before shutting down the car, doctor Healthy asks *TrafficWare* to move back to his office computer. This scenario is visualized in figure 5.1.

5.2 Analysis and Implementation

In the scenario we presented a route planner called *TrafficWare* on the move. For every part of the program we will analyse the type of binding we need and show how to implement this. Since the *TrafficWare interface* is the most interesting part from a mobile application engineering point of view, we will mostly focus on the implementation of that part of the program. The eventual design is visualized in figure 5.2[1].

In the beginning, the program is running on the office computer. Before starting to calculate the route, it uses the GPS to figure out its own position. So the it will need a ambient reference to a GPS. Of course after moving the program we want to reconnect, so the field must be annotated to rebind to this ambient resource after moving. In the next piece of code we extend the standard *TrafficWare* behaviour with GPS support.

[1]Notice the comparison with figure 3.1

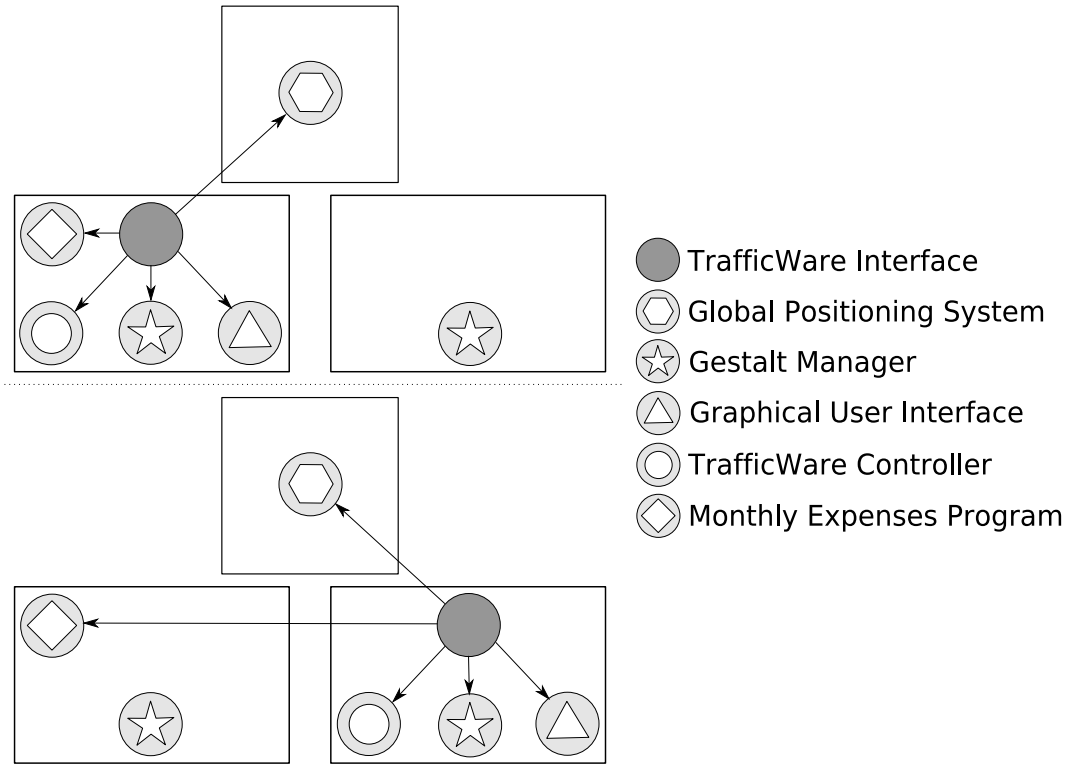

Figure 5.2: Mobile Route Planner Analysed

Listing 5.1: GPS: Ambient Rebind

```
1  mixinGPS(behaviour)::extend(behaviour,{
2    gps:void;
3    init()::{
4      super().init();
5      gps:=ambientRef("GPS");
6
7      ‘ On moving the application, we want to recreate ‘
8      ‘ a local proxy for the GPS ambient service      ‘
9      ‘ identified by the pattern "GPS"                ‘
10     ambientRebindTypeBox.add("gps","GPS")
11   }})
```

TrafficWare has a graphical user interface which will be recreated after the program has moved. To establish a good integration on new devices, it uses the gestalt manager to figure out the resolution of the screen. Of course the gestalt manager is a local resource and will be needed to be rebound locally after the move. The following code will show how the graphical user interface is being recreated, as well as how the gestalt manager is being rebound.

Listing 5.2: GUI: Recreate, Gestalt Manager: Local Rebind

```
1  mixinGUIGestalt(behaviour)::extend(behaviour,{
2    gui:void;
3    gestaltManager:void
4
5    createGui():{ ‘ This function will create a GUI ‘ };
6
7    init()::{
8      super().init();
9      gui:=createGui();
10     recreateTypeBox.add("gui",createGui)
11     ‘ By definition, on first creation of the actor,  ‘
12     ‘ a local gestalt manager actor will be available ‘
13     ‘ in the scope of the actor.                      ‘
14     gestaltManager:localGestaltManager;
15     ‘ We expect an actor who retrieves us to pass     ‘
16     ‘ along local resources containing the            ‘
17     ‘ "GestatManager" resource                        ‘
18     localRebindTypeBox.add("gestaltManager","GestaltManager")
19   }})
```

At the moment when the car arrives back home, the travelled distance has to

be sent to the "monthly expenses program". Since *TrafficWare* was already linked to the program when it was still residing on the office computer, this resource has to be handled in a default way. This means that while the application is on the board computer of the car, the program will be thought of as a remote resource. In the code below you can see that nothing special needs to be done for the "default" fields. If you want to ensure this behaviour is used at some point in the program, the command `defaultTypeBox.add("field")` can be used.

Listing 5.3: Monthly Expenses: Default Behaviour, Become Network Reference

```
1 mixinExpenses(behaviour)::extend(behaviour,{
2   ‘ By definition, on first creation of the actor, ‘
3   ‘ a local monthly expenses actor will be available ‘
4   ‘ in the scope of the actor. ‘
5   monthlyExpenses:localMonthlyExpenses; })
```

The *TrafficWare* itself will be designed as a two-layered program. First we have the interface to the outside world. This actor has links to all the external resources and will communicate with the outside world. The second actor is the controller of the program. It does the actual route calculating. Since these parts work closely together, if one moves it automatically means that the other has to move too. In this case the controller will always be asked to follow the interface because the latter does the communication. The following code features two needed parts of the program. The first part shows how to extend the controller to become moveable. The second part shows how to automatically let a moveable field be fetched when moving yourself.

Listing 5.4: *TrafficWare Controller*: Move Along

```
1  ‘ The extended behaviour for the now moveable ‘
2  ‘ TrafficWare controller ‘
3  mixinMobility(behaviour)::extend(behaviour,{
4    move.come(resources)::{
5      moveNowWithResources(resources)
6    }})
7
8  ‘ The behaviour for the TrafficWare interface ‘
9  mixinController(behaviour)::extend(behaviour,{
10   controller:void;
11   init()::{
12     controller:=actor(twControllerBehaviour);
13     moveAlongTypeBox.add("controller")
14   }})
```

Since mobility is done by using *move methods* as seen in section 3.2.3, devices accepting programs need some sort of a device manager. A program who wants to move to a device needs to send a migration request to the manager. This manager can then choose to allow the migration by responding with a move message, or choose to deny the request ignoring the message. Because *TraffícWare* wants to move between the office computer and the board computer of the car, both of them will need a device manager. Both of the devices already have a gestalt manager, so we will extend this actor to also fulfill the duty of "gatekeeper". The last piece of code is divided into two parts. The first part extends the *TraffícWare interface* again. This time with graphical user interface code to allow users to request the move to another device. The second part are extensions for the gestalt manager. With the extensions it is able to forward a move request of an application to the linked device on the one hand. It is also able to accept a request by replying with a move message on the other hand.

Listing 5.5: Making a Program Moveable

```
1  ' The behaviour for the TrafficWare interface '
2  mixinMoveable(behaviour)::extend(behaviour,{
3    f:void;
4
5    ' Create move-button in the gui '
6    createGui()::{
7      f:java.awt.Frame("Mobility Interface");
8      b:java.awt.Button("Request Move");
9
10     ' Use the trafficware as actionListener '
11     b.addActionListener(thisActor());
12     f.show();
13     f
14   };
15
16   ' Because this actor will also play actionListener '
17   actionPerformed(e)::{
18     ' The gestalt manager knows how to find '
19     ' the other device, so it is going to send '
20     ' out the request. The other device should '
21     ' respond by sending 'come' to thisActor() '
22     gestaltManager#sendRequestMove(
23       thisActor()#come
24     )
25   };
```

```
26
27    ‘ Make the TrafficWare interface actually moveable ‘
28    move.come(resources)::{
29      moveNowWithResources(resources)
30    };
31
32    init()::{
33      f:=createGui();
34      recreateTypeBox.add("f",createGui)
35    }})
36
37  ‘ The mobility extensions for the gestalt manager ‘
38  mixinMobileApplications(behaviour)::extend(behaviour,{
39    ‘ The gestaltmanager keeps an ambientRef to the ‘
40    ‘ other device. This is the example for the     ‘
41    ‘ desktop computer                              ‘
42    other:ambientRef("MyCar");
43
44    sendRequestMove(resultMessage)::{
45      ‘ Prepare the message as if it came from the   ‘
46      ‘ application requesting to send a request out ‘
47      message:other#requestMove;
48      message.setSource(thisMessage().getSource());
49      message.setArgs([resultMessage]);
50      ‘ Send out the message ‘
51      message.send(this())
52    }
53
54    ‘ Method to be able to answer to a move request ‘
55    requestMove(resultMessage)::{
56      resultMessage.setSource(thisActor());
57      ‘ The gestalt manager handles local resources ‘
58      resultMessage.setArgs([resources]);
59      ‘ Send out the message ‘
60      resultMessage.send(this())
61    }
62  })
```

5.3 Summary

In this chapter we showed by example how strong pull mobility based on move messages extended with declarative field annotations can easily be used to engineer mobile applications. With only a few lines of code, a full application depending on a lot of different types of resources can be moved back and forth from device to device. The fields containing resources just have to be annotated with mobility semantics defining what should happen to linked resource while moving an application and the actual moving, rebinding, etc. is handled by the programming language.

6 Conclusions

As explained in chapter 1, programs designed for *mobile networks* are easily going to be required to be mobile themselves too. This is caused by the characteristics of these type of networks which are different from stationary networks.

6.1 Problem Statement Revisited

There are already a number of programming languages providing support for this feature. Unfortunately they only consider the mobility of single entities in a program. If we would want to write actual mobile applications in these languages, it would most likely result in code cluttered with mobility-specific code.

This brings us to the goal of our research, extending programming languages featuring computational mobility with *declarative field annotations* in order to easily impose and change mobility relations between interconnected objects and resources.

6.2 Annotations in a Nutshell

6.2.1 Ambient-Oriented Mobility in AmbientTalk

We used the ambient-oriented programming paradigm as basis for our research because of the design reasons for the paradigm. It is designed to automatically cope with the characteristics of mobile networks. The advantage of this situation

is that the programming language is set up perfectly to start with. To be more specific we chose the ambient-oriented programming language called AmbientTalk. The reason was that AmbientTalk is designed as a reflective kernel, to allow programmers to easily adapt the language in order to experiment with new language constructs.

We extended this programming language with *strong mobility*. This choice came from the fact that we want to move running applications from one device to another, thus excluding *weak mobility*. Weak mobility focusses on sending or retrieving "dead code" to or from remote devices, for evaluation. Although moving running processes is doable by using only weak mobility, it will most certainly result in a bad coding style. When using weak mobility, in order to move running code, the runtime state needs to be saved and restored manually. This makes code non-modular, hard to reason about and thus harder to debug.

Move Methods

We are using AmbientTalk where actors are the unit for distribution. Therefore actors are also used as unit of mobility. Passive objects in AmbientTalk are already moveable from one device to another. As a security measure, we decided to use *move messages* to initiate the move of an actor. This implies that on the one hand, the moveable actor has to implement a *move method* (the actor must agree to be moveable), and on the other hand that the message must be sent from the accepting device (the device must agree to accept the moving actor). An extra advantage of using this model in AmbientTalk is that it automatically excludes race conditions on mobile actors, since an actor in AmbientTalk processes its asynchronously received messages one by one. This ensures that actors only react to one *move message* at a time.

Because several objects might need some pre- and/or post-processing (think of file-descriptors or windows which need to be closed on the device from which the actor is leaving, or opened on the device on which the actor is arriving) the move method is divided into two parts. A pre-move part, of which all code will be executed on the sending device. Secondly a post-move part, of which all code will be executed on the receiving device.

Updating Pointers

When moving objects in a programming language, the language must make sure that all object pointing to the moved object are automatically updated. There are two aspects in our solution. First we decided to add an extra abstraction for actors, internally containing a local or a remote representation of the actor. In this way pointers on the sending and receiving device do not have to be updated since they

keep pointing to the abstraction. To move an actor we replace the old local by the new remote representation and the old remote by the new local. As second step *third-party pointers* are updated when they are used to send a message to the actor. The new remote will inform the third-party pointer with the new location of the actor.

6.2.2 Declarative Field Annotations for Mobile Applications

Since this strong mobility implementation only defines what should happen to pointers pointing to the moved object, the next step is to look what should happen to outgoing pointers. For incoming pointers there is only one scenario possible. All pointers should always point to the actual object, wherever it resides. For outgoing pointers on the other hand, we identified five different scenarios. As an easy way to define which of these scenarios applies to which pointer, we suggested adding a *declarative field annotation system* to the programming language. In this way programmers can easily annotate fields with mobility semantics and further fully neglect the inner workings of moving an application.

Dynamic Annotations

We added two levels of dynamism to this system. First we made sure that the annotation of a field can be changed. This by using first-class disjunct sets of field-names, called *typeboxes*, instead of static typing. This is one of the requirements of adding annotations concerning mobility, since the mobility semantics of a field might change in the lifetime of an active object.

Then we added a way to easily add new types of mobility semantics for fields, by introducing a native which can create new typeboxes. These typeboxes all have to specify a pre- and a post-processing mechanism, in the line of the design of the two-sided move methods. Even while this is not explicitly a requirement it might prove very useful for objects with similar mobility behaviour for multiple resources, deviating from the standard behaviours specified in this dissertation. Adding typeboxes could be thus be used as extra abstractions by the programmer.

Standard Types of Resources

We pointed out that in every language using typeboxes, there should be five default typeboxes available, which are deduced from the five scenarios we identified for the outgoing pointers of moving active objects. We distinct between remote and local resources which have to be rebound, resources which have to stay at their original location, resources which have to be recreated and resources which have to be moved along with the moving object. When the resource stays behind or

moves along, the pointer from the moving object to the resource stays the same. In the other three cases the pointer has to be changed by a pointer to the new resource replacing the old one.

6.3 Future Work

In this section we present some topics which are related to this dissertation and which may be interesting for future research.

- **Race Conditions on Strong Mobility.** Even while we presented a solution for this problem, which is acceptable from a language engineering point of view, the result might not be feasible for the programmer. In our result, we do not get any logic errors, but the position of an actor depending on the order of incoming messages also presents fuzzy semantics.

- **Multiple Annotations for One Object.** All objects in one moving actor, in the current annotation system, will only be moved in one and the same fashion. The design reason is explained in section 4.1.3. As a solution we decided to just use the last annotation applied to an object, in the same line as adding a field-name to a typebox cancels a previous addition to another typebox. As this might be a good and logical solution, there may also be a more logic solution. For example one might impose precedence rules on typeboxes.

- **Mobile Symbiosis.** In our current solution Java objects are represented as passive objects. These Java objects internally may spawn multiple threads. From within AmbientTalk programmers are allowed to pass AmbientTalk code as interface-implementing objects, mostly used for listeners. To keep the AmbientTalk part thread-safe, we only allow programmers to pass actors, to which synchronous calls will be translated into asynchronous messages.

  This solution has a few problems which we did not address, but which could be interesting for future use. One of the problems of this solution is that these messages are asynchronous. This implies that calls to the interfaces always return a null-pointer, while some of the Java object actually might want a real return value.

A Typeboxes Implementation

Here you find the implementation of the default typeboxes in AmbientTalk, as specified in section 4.2.3.

Listing A.1: Typeboxes in AmbientTalk

```
moveAlongTypeBox::createTypeBox(
  lambda(object,resources,config) → {
    ` Pre-Move, no pre-processing is     `
    ` necessary                          `
    void
  },
  lambda(object,resources,config) → {
    ` Post-Move, a come-message is sent `
    ` to the actual object               `
    object#come(resources)
  }
);

recreateTypeBox::createTypeBox(
  lambda(object,resources,config) → {
    ` Pre-Move, the object is going to  `
    ` be writeReplaced with an object   `
    ` defined by the closure passed     `
    ` while adding the field to the box `
```

```
    ` This is the first argument, thus  `
    ` the second element of config      `
    writeReplace(object,readResolve(config[2]))
  },
  lambda(object,resources,config) → {
    ` Post-Move, no post-processing is  `
    ` necessary                         `
    void
  }
);

localRebindTypeBox::createTypeBox(
  lambda(object,resources,config) → {
    ` The object is going to be         `
    ` writereplaced with a local        `
    ` resource defined by the passed    `
    ` argument                          `
    writeReplace(object,resources.get(config[2]))
  },
  lambda(object,resources,config) → {
    ` Post-Move, no post-processing is  `
    ` necessary                         `
    void
  }
);

ambientRebindTypeBox::createTypeBox(
  lambda(object,resources,config) → {
    ` The object is going to be         `
    ` writereplaced with an ambient     `
    ` resource defined by the passed    `
    ` argument                          `
    writeReplace(object,
                 readResolve(lambda() → {
                               ambientRef(config[2])
                             })
                )
  },
  lambda(object,resources,config) → {
    ` Post-Move, no post-processing is  `
    ` necessary                         `
```

```
    void
  }
);

defaultTypeBox::createTypeBox(
  lambda(object,resources,config) → {
    `Pre-Move, no pre-processing is    `
    `necessary                         `
    void
  },
  lambda(object,resources,config) → {
    `Post-Move, no post-processing is  `
    `necessary                         `
    void
  }
)
```

Bibliography

[AFV98] GP. Picco A. Fuggetta and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 1998.

[Car99] L. Cardelli. Abstractions for mobile computation. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, 1999.

[cod04] Codamos: Context-driven applications of mobile services. d1.1.2 scenarios. 2004.

[Dha01] Ole-Johan Dhal. The birth of object-orientation: the simula. 2001.

[DHH01] P. Brand D. Havelka, C. Schulte and S. Haridi. Thread-based mobility in oz. 2001.

[EGV94] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[EJB88] N. Hutchinson E. Jul, H. Levy and A. Black. Fine-grained mobility in the emerald system. 1988.

[Gra95] R. S. Gray. Agent tcl: A transportable agent system. 1995.

[JDM06] T. D'Hondt S. Mostinckx J. Dedecker, T. Van Cutsem and W. De Meuter. Ambient-oriented programming in ambienttalk. 2006.

[JPBL98] R. Guerraoui J.-P. Briot and K.-P. Lhr. Concurrency and distribution in object-oriented programming. 1998.

[Meu04] W. De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, 2004.

[PS97] H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. 1997.

[US87] D. Ungar and R. Smith. Self: The power of simplicity. 1987.

[Ver06] T. Verwaest. Language symbiosis between ambienttalk with mobility and java. 2006.

[Whi96] J. White. Mobile agents white paper. 1996.

[Wik] Wikipedia. Duck typing article. url `http://en.wikipedia.org/wiki/Duck_typing`.