Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen
Software Languages Lab

Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Andoni Lombide Carreton

Promotor: Prof. Dr. Wolfgang De Meuter
Copromotor: Prof. Dr. Theo D'Hondt

Oktober 2011

Print: Silhouet, Maldegem

2011 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5487 972 5
NUR 980 / 986 / 989
Legal deposit D/2011/11.161/143

"Simple molecules combine to make powerful chemicals. Simple cells combine to make powerful life-forms. Simple electronics combine to make powerful computers. Logically, all things are created by a combination of simpler, less capable components. Therefore, a supreme being must be our future, not our origin!"

– Scott Adams (1957 -), Dilbert comic, February 11, 1996

Abstract

In ubiquitous computing research it is generally accepted that software has to be loosely-coupled and event-driven. Applications have to be decoupled in space, time and arity in order to cope with dynamically reconfiguring wireless networks and intermittent network connections. They have to be event-driven in order to react to fluctuating wireless connections, users, other computers, sensors, etc. This has led to a number of programming models tailored towards such distributed event-driven applications. The one on which this work is based is the *ambient-oriented programming* paradigm [DVCM+05], which is a programming language paradigm specifically tailored towards applications running on mobile devices, such as smartphones.

RFID is an emerging wireless technology that allows to enrich everyday objects with a digital representation. Current RFID-enabled applications merely use RFID tags as barcodes, associating physical objects with a digital representation in a centralized backend database. However, RFID technology is increasingly integrated in mobile devices, giving rise to *mobile RFID-enabled applications*. We begin this dissertation by extending ambient-oriented programming to be able to conceive mobile RFID-enabled applications as ambient-oriented programs running on mobile devices where remote software objects can designate tagged real-world things.

Like any event-driven architecture, ambient-oriented programming requires the programmer to coordinate an application by means of callbacks, which causes the well-documented problem of inversion of control [HO06]. This problem becomes much more prominent in mobile RFID-enabled applications because of the massive number of events that such applications need to react to. This dissertation proposes dataflow programming in order to tackle this problem. Dataflow programming allows reacting to events without inversion of control [MRO10]. Unfortunately, current incarnations of dataflow programming do not offer the aforementioned decoupling in space, time and arity. To achieve this decoupling, we integrate dataflow programming with ambient-oriented programming into *ambient-oriented dataflow programming*.

We analyze ambient-oriented dataflow programming on two levels, namely the node level and the network level. With *node-centric* ambient-oriented dataflow programming, nodes specify their distributed dataflow dependencies declaratively by subscribing to and publishing events, and let the language runtime take care of coordinating dependent event-driven code instead of relying on callbacks. However, since dependent code is evaluated implicitly in response to remotely signaled events, the global control flow of a large distributed applications becomes implicit and hard to predict. This leads us to design a *network-centric* visual ambient-oriented dataflow language in which an explicit network-level data flow is used for coordinating an application. In order to cope with the dynamic nature of mobile RFID-enabled applications, it allows to visually specify the arity and binding semantics of distributed dataflow dependencies.

Samenvatting

In onderzoek naar ubiquitous computing is het algemeen aanvaard dat software losgekoppeld en event-gedreven moet zijn. Toepassingen moeten ontkoppeld zijn in tijd, ruimte en ariteit zodat ze kunnen omgaan met dynamisch herconfigurerende draadloze netwerken en frequent wegvallende netwerkverbindingen. Ze moeten event-gedreven zijn om te reageren op fluctuerende draadloze verbindingen, gebruikers, andere computers, sensors, enz. Dit heeft geleid tot een aantal programmeermodellen toegespitst op dergelijke gedistribueerde event-gedreven toepassingen. Dit werk is gebaseerd op het *ambient-georiënteerd* programmeerparadigma [Ded06], wat een programmeertaalparadigma is dat specifiek gericht is naar toepassingen die draaien op mobiele toestellen, zoals smartphones.

RFID is een opkomende draadloze technologie die toelaat alledaagse objecten te verrijken met een digitale representatie. Huidige RFID-verrijkte toepassingen gebruiken RFID slechts als een barcode, waarbij fysieke objecten geassocieerd worden met hun digitale representatie in een gecentraliseerde backend database. Echter, RFID-technologie wordt meer en meer geïntegreerd in mobiele toestellen zoals smartphones, wat aanleiding geeft tot *mobile RFID-enabled applications*. We beginnen dit proefschrift met het ambient-georiënteerd programmeren uit te breiden zodat mobile RFID-enabled applications kunnen geschreven worden als ambient-georiënteerde programma's die draaien op mobiele toestellen en waarin gedistribueerde software-objecten getagde objecten uit de echte wereld (things) kunnen voorstellen.

Zoals bij alle event-gedreven architecturen, vereisen ambient-georiënteerde toepassingen dat de programmeur de toepassing coördineert door middel van callbacks, wat het uitvoerig gedocumenteerde probleem van inversion of control met zich meebrengt [HO06]. Dit probleem doet zich bij mobile RFID-enabled applications op een veel grotere schaal voor door het massale aantal aan events waarop deze toepassingen moeten reageren. Dit proefschrift stelt dataflow programming voor om dit probleem aan te pakken. Dataflow programming laat toe op events te reageren zonder inversion of control [MRO10]. Helaas bieden huidige incarnaties van dataflow-technologie niet de bovennoemde losse koppeling in tijd, ruimte en ariteit aan. Om een dergelijke ontkoppeling te bereiken, integreren we dataflow programming met ambient-georiënteerd programmeren in *ambient-georiënteerd dataflow programming*.

In dit proefschrift analyseren we ambient-georiënteerd dataflow programming op twee niveaus, namelijk het knoopniveau (node level) en het netwerkniveau (network level). Met *node-centric* dataflow specifiëren knopen declaratief hun gedistribueerde dataflow-afhankelijkheden door zich te abonneren op of door het publiceren van events, en laten het coördineren van afhankelijke event-gedreven code over aan de runtime van de taal in plaats van gebruik te maken van callbacks. Echter wordt hierdoor afhankelijke code impliciet geëvalueerd als reactie op gedistribueerd gesignaleerde events, wat de globale control flow van grotere gedistribueerde toepassingen impliciet en moei-

lijk te voorspellen maakt. Dit leidt ons tot het ontwerp van een *network-centric* visuele ambient-georiënteerde dataflow-taal, waarin om een toepassing te coördineren de expliciete data flow op netwerkniveau wordt gebruikt. Om te kunnen omgaan met het dynamische karakter van mobile RFID-enabled applications, laat deze taal toe de ariteit en bindingssemantiek van gedistribueerde dataflow-afhankelijkeden visueel te specifiëren.

Acknowledgements

First and foremost, I would like to thank my promotor Prof. Wolfgang De Meuter and copromotor Prof. Theo D'Hondt for giving me this opportunity and promoting this thesis. Not only over the past four years, but during my time as an undergraduate as well, they have continuously challenged me, picked my interest, and given invaluable advice. Especially Wolfgang's relentless but infinitely useful proof-reading and discussions about my work and in the end my dissertation have transformed me from a student to a researcher.

I sincerely thank the members of my jury for the time and effort they put in evaluating this work and in giving insightful comments: Prof. Patrick Eugster (Purdue University), Prof. Tom Holvoet (Katholieke Universiteit Leuven), Prof. Tom Lenaerts (Université Libre de Bruxelles), Prof. Beat Signer (Vrije Universiteit Brussel) and Prof. Viviane Jonckers (Vrije Universiteit Brussel). Having to reflect on and defend my work on such a deep level was hard but fair and a satisfying experience. I can only repeat how much I truly appreciate the effort that must have gone into this.

I would like to again thank Prof. Wolfgang De Meuter, Prof. Theo D'Hondt and Prof. Viviane Jonckers for sparking my interest in programming languages, software engineering, and Computer Science research in general. From where I am now, I appreciate their outstanding teaching even more than when I was an undergraduate student. It is also a joy to take part in such teaching ethics when assisting their courses.

The role of the other members of the Software Languages Lab should neither be neglected. Together, they have made my work environment an intellectual playground where a lot of good memories have been forged. Space constraints prevent me from listing all of them here. but to some of them I would like to especially express my gratitude. Dr. Tom Van Cutsem for his guidance and proof-reading my text, especially in my first years. Kevin Pinte for contributing to a large part of this research, both technically and intellectually, and for being an excellent office mate. Stijn Mostinckx for the many fruitful discussions and for putting me on track. Engineer Bainomugisha for proof-reading my dissertation.

Special thanks also goes to the Context & Ambient-oriented Programming Group within the lab for the collaboration. Especially the legendary Friday-afternoon hacking sessions with Christophe Scholliers, Elisa Gonzalez Boix and Dries Harnie were we discussed, fought and laughed, but in the end realized quite a lot of the crazy ideas we came up with. Special thanks also goes to Dr. Jorge Vallejos for helping me out with setting everything up for my private and public PhD defense.

Other than colleagues, I have made a lot of friends at the lab. Without these friendships, my incentive to perform would have been much lower. So thanks to my drinking buddies (coffee at noon or beer on Friday evenings, take your pick): Prof. Wolfgang De Meuter, Christophe Scholliers, Elisa Gonzalez Boix, Dr. Coen De Roover, Dr. Ellie d'Hondt, Matthias Stevens, Stefan Marr, Dr. Andy Kellens, Dirk Van Deun, Dr. Car-

los Noguera, Nicolas Cardozo and Stijn Timbermont. I would like to thank Dr. Sofie Goderis for showing me around as my first office mate.

Next to my colleagues, I would like to thank a number of people that in essence had nothing to do with my work, but contributed in other kinds of ways to my success. While my colleagues made me glad to go to work, these people made me look out for relaxation after a hard day of work, hence contributing to my mental well-being. Glenn Van Den Wijngaert, Thomas Tooten, Wouter Tooten and Jelle De Waegeneer for the many not-so-sober nightly philosophical discussions in the Mechelen area. Cedric Ghequiere and Hans Vanweyenberg for all our musical endeavors.

Finally, I would like to thank the people that have been the closest to me these years. My parents for supporting me in too many ways to list here. Without their support, nothing of this would have been possible. And last but not least, my girlfriend Tine Lesire for her endless support and for being there for me, whatever happened. She not only endured me in some of my most stressful moments, but dragged me through them.

This work is funded by a PhD scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ubiquitous computing is a futuristic vision on computing first proposed by Mark Weiser [Wei91, Wei93] that is slowly becoming reality. In ubiquitous computing, computers are not the user's central point of attention anymore. Instead, the use of computers is integrated in the everyday tools that users interact with. At the time of writing this text, we are already observing a paradigm shift into the direction of ubiquitous computing, similar to the one that has been observed when mainframe computing transitioned into personal computing. Everyday and everywhere, people are carrying a multitude of mobile devices that interact with the user, other mobile devices, computing systems connected through the internet, or even the physical world. These mobile devices can be devices that require the user's attention, such as smartphones, portable media players, laptops, game consoles, etc. In many other cases computers are integrated in our physical environment and aid us in our everyday tasks without us even knowing. Examples are driving assistance systems in cars, home theater systems, product scanners in supermarkets, digital picture frames that change the displayed picture when they detect the presence of a certain person, etc. Just like the mainframe in the past, the Personal Computer is no longer the only gateway to digital information and digital interaction. Everyday objects are enriched with microprocessors and are taking over that role.

Together with the integration of computing devices in everyday objects and tasks, an increase in the proliferation of sensor devices is observed. Indeed, computing devices can only assist in their users' everyday tasks unobtrusively if they are able to autonomously infer information about the physical world in which their users reside. A modern mobile phone is equipped with motion sensors and/or gyroscopes, light sensors, a microphone, a touch screen, one or more cameras, a GPS module, an RFID reader, etc. and other kinds of sensors will undoubtedly follow. The recent explosion of sensor integration in these devices can be aligned with computing applications that interact with the physical world: GPS modules determine the location of objects, objects are identified while being filmed, the movement of objects is tracked etc., slowly causing the digital world and physical world to converge. In the context of web applications, Serrano terms this convergence *diffuse computing* [Ser09].

However, for this to happen, sensors alone are not enough. In all but the simplest applications, sensor information must be reacted upon, processed and delivered to the interested parties, which can be software components, devices or people. Hence, wireless communication is essential to allow sensor-equipped mobile devices to interact with each other and thus contributing to a meaningful application [TP05]. Such an application in a ubiquitous computing environment is not a classic monolithic pro-

gram that accepts user-supplied input and processes it into some user-targeted output. Instead, ubiquitous computing applications execute fluidly and spontaneously when heterogenous mobile devices interact. To remain causally connected to each other and their environment, such applications continuously have to respond to stimuli from their environment, whether they be sensor input, user input, or communication with other software components. Hence, these applications are *highly event-driven*.

Additionally, in many cases ubiquitous computing applications need to be aware of which physical objects are in their proximity. The most widely used sensor technology for this purpose is Radio Frequency Identification or RFID [WFGH99]. RFID is considered a key enabler for ubiquitous computing [WJ09, Ble06]. RFID-enabled applications consist of RFID tags that can be used to "tag" physical objects and RFID readers that wirelessly communicate with these tags using radio frequencies. RFID is already used in many application domains such as public transport identification cards, stock management, product tracking etc. At the same time, RFID tags are becoming so cheap and tiny that it will soon be possible to tag one's entire environment, thereby wirelessly dispersing information to interested parties and providing digital representatives of physical objects in the digital world. This vision is coined the *Internet of Things* by the International Telecommunication Union [Uni05].

Just like other types of sensors, RFID technology is increasingly integrated in smartphones, rendering RFID-enabled applications one of the prime examples of mobile highly event-driven applications that wirelessly communicate sensor data. In this dissertation, we will name such applications running on mobile devices and using RFID technology as a bridge between the physical and the digital world *mobile RFID-enabled applications*.

The screenshot shown in figure 1.1 below is from an example mobile RFID-enabled application that was developed in the context of our research. This example application

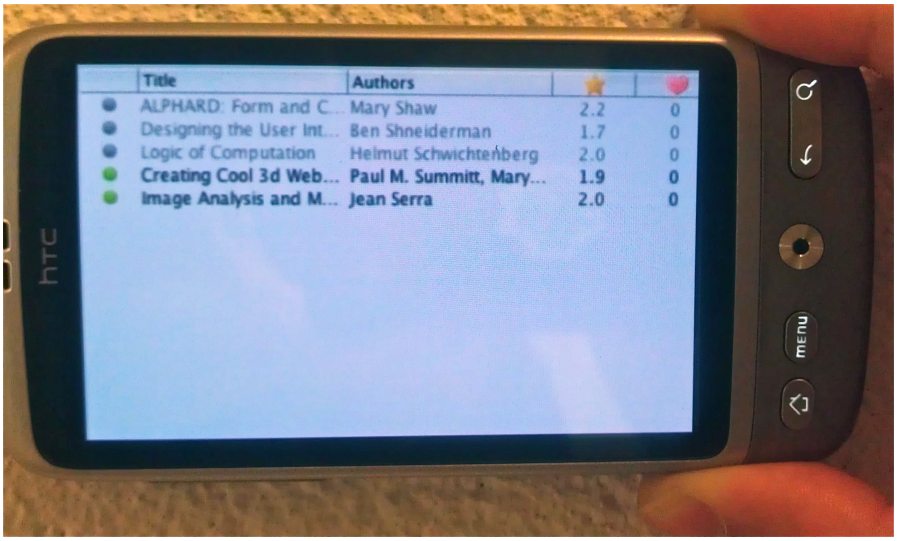

Figure 1.1: Screenshot of a mobile RFID-enabled library application.

lists RFID-tagged (physical) books that are in RFID communication range of a user's smartphone. The greyed out books are books of which the RFID connection is currently broken. The important part is that the list adapts itself as soon as new books are

discovered or the connection with other books is broken while the user moves about. This means that this simple application is steered by external events which are continuously generated by the RFID hardware, such as new RFID tags that are detected or RFID tags with which the connection is broken.

Although computing and sensor hardware is rapidly developing and scenarios with mobile event-driven applications are abundant, programming such applications is still done using very rudimentary tools. These programming tools are traditional tools that focus on applications running on a personal computer and not on ubiquitous computers. Although many advancements have been made in programming abstractions targeted towards applications that consist of spontaneously interacting wireless devices, the *reactive* aspect of mobile event-driven applications has not received equal attention. The aim of this dissertation is to advance the state of the art of software engineering in the field of mobile RFID-enabled applications. Concretely, we will apply results from preceding research in mobile and wireless computing applications to present a programming model for mobile RFID-enabled applications. Unfortunately, applying this preceding research as is for implementing such highly event-driven applications that must react to continuous streams of events causes the implementations to exhibit an unpredictable control flow. Therefore, our second goal is to investigate how this unpredictable control flow driven by events can be made predictable again.

1.1 Research Context

As discussed above, our research lies in the application domain of mobile event-driven and mobile RFID-enabled applications. This application domain can be seen as the crossroads of a number of research domains:

Mobile Ad Hoc Networks are key to the deployment of ubiquitous computing applications. Mobile ad hoc networks spontaneously emerge by the colocation of devices with wireless networking capabilities, without needing extensive configuration or additional infrastructure. This allows ubiquitous computing software to interact unobtrusively and spontaneously. The different ranges of wireless network technologies lead to computer networks composed of mobile devices of which the topology dynamically changes as devices move about, causing network connections to form, break and restore frequently and unpredictably. Mobile RFID-enabled applications operate in a mobile ad hoc network. Furthermore, RFID-technology can be regarded as an extreme case of mobile ad hoc networking technology: the mere presence of an RFID tag can initiate wireless communication with devices equipped with an RFID antenna. This wireless communication is prone to failures because of the mobility of the devices and the unreliable nature of RFID communication. Therefore, our research in the software engineering aspect of mobile RFID-enabled applications should also cope with programming issues caused by the highly dynamic nature of these networks.

RFID Technology is an enabling technology for connecting software to physical objects and is therefore one of the key technologies in ubiquitous computing. As mentioned in the preceding discussion about mobile ad hoc networks, the spontaneous and unreliable nature of RFID tags broadcasting data to all devices in range that are able to receive that data, induce highly dynamic applications. Combining the vision of unobtrusive mobile computing with RFID technology leads to a type of applications that we coined mobile RFID-enabled applications earlier in this chapter. Our research

tries to answer the software engineering problems caused by representing RFID-tagged physical objects using software proxies a in highly dynamic environment where communication happens by unreliable wireless network connections.

Distributed Event-Driven Programming naturally arises when networked software systems are deployed in an ad hoc fashion in a dynamic environment. Software has to react to incoming requests from other software applications or users and to the appearance and disappearance of other parties in the network. This is already true for internet-scale and client/server applications, as can be seen from the widespread usage of event-driven web technologies such as Ajax [MGB+09]. For mobile ad hoc networking applications, several event-driven programming models have been conceived as well. One of them is the *ambient-oriented programming* paradigm, which we use as the starting point for our research. In a ubiquitous computing setting, the portion of software that is driven by external events is much higher because mobile ad hoc networks are highly dynamic and the software is embedded in the physical world. Mobile RFID-enabled applications are an extreme case of distributed event-driven software, because of a greater number of and greater variety in events combined with the fact that these applications operate in unreliable mobile ad hoc networks.

In the crossroads of these three domains, one can find a number of problems that render the development of mobile RFID-enabled applications with current programming technology a tedious task. These problems are summarized in the next section.

1.2 Problem Statement

The problems we address in this dissertation are twofold:

Lack of programming abstractions to develop mobile RFID-enabled applications Mobile RFID-enabled applications use RFID technology in a radically different way than current RFID systems. Current systems basically use RFID tags as mere digital barcodes and almost never exploit the writable memory on these tags [PS11, DMS07, MTCS07]. For this, today's systems assume infrastructure in the form of a centralized backend database that associates the digital barcode with its semantics, which restricts RFID to traditional applications such as stock management or item tracking. Mobile RFID-enabled applications require more functionality from passive computing devices such as RFID tags. It should be possible to enrich everyday RFID-tagged objects with application-specific contextual information without relying on a backend database or other infrastructure by storing the data on the tags themselves.

To make use of this functionality, developers of mobile RFID-enabled applications are forced to rely on lower-level abstractions to interact with RFID hardware, in most cases even directly on the hardware driver level. Furthermore, when implementing mobile RFID-enabled applications with traditional programming abstractions, the programmer must deal with the fact that communication with RFID tags is prone to many failures. Tags close to each other can cause interference and can move out of the range of the reader while communicating with it. These failures may be permanent, but it may be that at a later moment in time the same operation succeeds because of minimal changes in the physical environment. For example, a tag moves back in range or suddenly suffers less from interference. As a consequence, dealing with these intermittent failures and interacting with the low-level abstraction layers offered by RFID

vendors from within a general purpose programming language results in complex and brittle code. In short, today we lack the programming abstractions to develop mobile RFID-enabled applications.

Inversion of Control In event-driven architectures, of which mobile RFID-enabled applications are an extreme case, it is no longer the programmer who steers the application's control flow. Rather, control is handed over to the application logic by means of callbacks whenever an event is detected. In other words, by adopting an event-driven architecture, the application logic becomes scattered over different event handlers or callbacks which may be triggered independently [CM06]. This is the phenomenon known as *inversion of control* [HO06, PE02]. Control flow among event handlers has to be expressed implicitly through manipulation of shared state. Unlike subsequent function calls, code triggered by different event handlers cannot use the runtime stack to make local variables visible to other event handlers (called stack ripping by Adya et al. [AHT+02]). Because handing over state from one event handler to another no longer relies on parameter passing, these variables have to be made instance variables, global variables, etc. This is why in complex systems such an event-driven architecture can become hard to develop, understand and maintain [LC02, KR05, MRO10].

Even though, as we will discuss in chapter 3 section 3.3, there exist event-driven programming technologies that allow writing event-driven code without suffering from inversion of control, these technologies are not suitable for decentralized applications running in mobile ad hoc networks. They assume an entirely local application or a traditional client-server setup.

1.3 Research Goals

The initial goal that sparked our research is implementing mobile RFID-enabled applications using modern distributed programming techniques. The different research goals outlined below follow naturally from this initial vision.

- Programming abstractions for mobile RFID-enabled applications should allow programmers to implement these applications without having to deal with irrelevant technological and/or hardware issues and expose important events regarding RFID hardware interacting with these applications. Our approach is to come up with a number of criteria that capture the essence of programming mobile RFID-enabled applications using object-oriented techniques. These criteria together must allow us to program mobile RFID-enabled applications as distributed object-oriented applications where the notion of a software object in an object-oriented programming language is unified with a real RFID-tagged physical object or "thing".

- Combining, as envisioned above, mobile RFID-enabled applications with distributed object-oriented programming requires the integration of the imperative nature of object-oriented programming in a distributed event-driven architecture. As mentioned in the previous section, this integration leads to a complex callback-based programming style causing inversion of control. Our goal is to provide programming support for the distributed event-driven nature of mobile RFID-enabled applications that integrates better with the imperative, object-oriented programming style. Concretely, we try to eliminate the need for call-

backs as much as possible, while still keeping the distributed application components loosely-coupled from each other and resilient to the frequent failures inherent to wireless communication.

- Even though, as we will show further in this dissertation, we will make it possible to implement mobile RFID-enabled without resorting to callbacks, the control flow of such an event-driven program no longer corresponds to the program text as imperative computations are triggered by external events. Our final goal is to provide a tool to give the application programmer – in addition to the classical node-centric view – a network-centric view on the control flow of a mobile RFID-enabled application.

1.3.1 Contributions

We achieve these goals by constructing a distributed object-oriented programming framework for mobile RFID-enabled applications. Additionally, we embed this framework into a dataflow programming language that provides a number of language constructs to construct mobile RFID-enabled applications without resorting to callbacks. Finally, we build a network-centric, visual coordination language on top of this architecture to expose the control flow of the event-driven application visually. Together, they form the main contributions of this dissertation:

Ambient-Oriented Programming for Mobile RFID-Enabled Applications In chapter 4 we describe the event-driven, object-oriented programming language AmbientTalk/2, that – as a scion of the ambient-oriented programming paradigm – specifically targets mobile applications. In chapter 5 we discuss the criteria that we have identified for supporting ambient-oriented programming with RFID technology. In the same chapter we build on the ideas of AmbientTalk/2 targeting applications running in mobile ad hoc networks, but extend the language to target mobile RFID-enabled applications that satisfy the criteria identified earlier. The features of the language already targeting distributed object-oriented mobile applications combined with our extensions to support RFID technology provide an event-driven, object-oriented framework for developing mobile RFID-enabled applications.

Ambient-Oriented Node-Centric Dataflow Programming In chapter 6 we discuss a version of AmbientTalk/2 targeted towards mobile event-driven applications called AmbientTalk/R. AmbientTalk/R employs a variant of the dataflow paradigm called *reactive programming* to construct event-driven applications without resorting to callbacks. Subsequently in this chapter, we discuss a number of language constructs that reconcile the dataflow programming style with techniques from distributed event-driven architectures to allow loosely-coupled distributed dataflow programming in mobile ad hoc networks.

Ambient-Oriented Network-Centric Dataflow Programming with AmbientTalk/R^V In chapter 7 we build on the abstractions discussed in chapter 6 to implement AmbientTalk/R^V: a visual coordination language based on the dataflow paradigm that allows the programmer of mobile event-driven applications to visually inspect and edit the coarse-grained application control flow. By unifying the control flow of the application with the data flow that drives its execution, the control flow can be represented

visually and becomes explicit again as the driving force of the visual program, allowing it to be separated from the fine-grained application logic and easily adapted. Furthermore, the visual representation of the dataflow program offers a network-centric view on the application while still offering the expressiveness of a full-blown object oriented language.

1.3.2 Supporting Publications

The following (co-)authored publications support the key ideas in this dissertation:

- **Distributed Object-Oriented Programming with RFID Technology** [LPD10]
  Andoni Lombide Carreton, Kevin Pinte and Wolfgang De Meuter
  10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2010)
  This paper proposes extending the ambient-oriented programming paradigm (detailed in chapter 4) to program mobile RFID-enabled applications, by considering RFID tags as intermittently connected *mutable software objects* hosted on mobile distributed computing devices. This work corresponds to chapter 5 of this dissertation.

- **Software Abstractions for Mobile RFID-Enabled Applications** [LPD11]
  Andoni Lombide Carreton, Kevin Pinte and Wolfgang De Meuter
  Software: Practice and Experience, 2011
  This paper extends the aforementioned paper with a mechanism to obtain data consistency in mobile RFID-enabled applications, details the implementation and benchmarks it.

- **Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks** [LMVD10]
  Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem and Wolfgang De Meuter
  48th International Conference on Objects, Models, Components, Patterns (TOOLS 2010)
  This paper proposes a number of language constructs that reconcile the elegant processing of events of a dataflow programming system with the loose coupling of a publish/subscribe system that is required to cope with the dynamic nature of mobile ad hoc networks. It corresponds to chapter 6 of this dissertation.

- **A Hybrid Visual Dataflow Language for Coordination in Mobile Ad Hoc Networks** [LD10]
  Andoni Lombide Carreton and Theo D'Hondt
  12th International Conference on Coordination Models and Languages (COORDINATION 2010)
  This paper presents an early version of AmbientTalk/R^V: a visual dataflow language tailored towards mobile event-driven applications to express the interaction between mobile components that operate on dependent time-varying values driven by remote events. By using a visual dataflow language as a separate coordination language, the coarse grained control flow of a mobile application can be specified visually and separately from the fine grained control flow. In its turn, this allows a very explicit and network-centric view on the control flow of the entire mobile application. It corresponds to chapter 7 of this dissertation.

1.4 Approach to the Problem

The work presented in this dissertation fits in the context of *ambient-oriented programming* (discussed in chapter 2), a programming paradigm focussing on mobile ad hoc networking applications, which are essential for ubiquitous computing to become realizable. As mentioned above, we consider RFID a restricted form of mobile ad hoc technology. Hence, we start by applying ambient-oriented programming to mobile RFID-enabled applications. In chapter 5, we extend the existing ambient-oriented programming programming language AmbientTalk/2 to be able to implement a mobile RFID-enabled application. The application implemented in this language from then on serves as an artifact that we will gradually adapt to make use of dataflow programming abstractions introduced throughout this dissertation.

Ambient-oriented programming deliberately structures mobile applications as event-driven architectures, but where reactivity is merely considered as a side effect of interacting throughout open and dynamically changing mobile ad hoc networks. This leads us to investigate a range of technologies that deal with reactivity as a main concern in chapter 3. We conclude that both *publish/subscribe technology* and *dataflow programming technology* offer solutions to different facets of the problem we are tackling. Hence, one of our objectives is to integrate publish/subscribe with distributed dataflow programming to obtain *ambient-oriented dataflow programming*.

Ambient-oriented dataflow is introduced in chapter 6. Subsequently, the effects of applying ambient-oriented dataflow programming techniques on our mobile RFID-enabled application are showcased. We observe that ambient-oriented and mobile RFID-enabled applications can be implemented without callbacks, but that the combination of a very loosely-coupled interaction style with a dataflow evaluation model introduces new issues, such as a very implicit global control flow.

For our final step, we recall from the surveyed technologies in chapter 3 that network-centric programming technology alleviates some of these issues. Hence, the final step is to bring the loosely-coupled dataflow programming model on a network-centric level by means of a visual coordination language that makes the global control flow very explicit by visualizing it. This is discussed in chapter 7, where it is again applied to the mobile RFID-enabled application used throughout this dissertation.

Finally, in the concluding chapter 8, we identify which issues remain to be solved, of which the most notable is the lack of distributed glitch prevention for ambient-oriented dataflow programs.

1.5 Dissertation Roadmap

The main contributions of this work are a decentralized mobile RFID programming model unifying software objects with physical objects, a node-centric dataflow programming model specifically geared towards mobile event-driven applications, and a network-centric visual dataflow coordination language for coordinating the control flow of such applications on a network-centric level. To point out why certain design choices were made and to explain the relation of our approach with various research fields, we survey a number of related technologies out of which we extract the properties that we consider useful for implementing mobile RFID-enabled applications. Afterwards, to ground the technical discussion, we also need to introduce AmbientTalk, the programming language in which our artifacts have been incorporated and implemented. Below, we summarize each subsequent chapter in the dissertation.

Chapter 2: Mobile RFID-Enabled Applications gives an informal description of what we call in this dissertation mobile RFID-enabled applications by discussing how the convergence of a number of emerging technologies leads to such applications. We discuss how certain hardware properties of these technologies have strong repercussions on software development, and extract a number of programming model requirements for concisely implementing such applications. Part of these requirements are taken into account by the ambient-oriented programming model, which we choose to extend to better support mobile RFID-enabled applications.

Chapter 3: Related Work Given the requirements identified in chapter 2, in chapter 3 we survey a number of technologies that offer interesting solutions for dealing with either event-driven applications, distributed applications or RFID-enabled applications, or a combination, and discuss how they fail to meet all our requirements. We extract a number of concrete solutions that will become the objectives of this work throughout the rest of this dissertation.

Chapter 4: Ambient-Oriented Programming with AmbientTalk/2 discusses the concrete programming language that we extended with constructs tailored towards mobile RFID-enabled applications. Our work is based on the ambient-oriented programming paradigm and AmbientTalk/2 is a concrete ambient-oriented programming language. It discusses which features of the language already offer us part of the solution. On the flip side, we discuss how AmbientTalk suffers from the problems arising in similar event-driven systems when the same event-driven techniques are applied to mobile event-driven applications.

Chapter 5: Ambient-Oriented Programming for Mobile RFID-Enabled Applications opens up the ambient-oriented programming paradigm to mobile RFID-enabled applications. This chapter describes one of the three contributions of our work: the unification of RFID-tagged physical objects with software objects acting as digital representations that are embedded in AmbientTalk's distributed, fault-tolerant, decentralized and event-driven programming model. Our approach is illustrated by a running example that will be used throughout this dissertation and a brief overview of the implementation in AmbientTalk is given. We observe that the problems with event-driven code are amplified when the same event-driven programming model is applied to such a mobile RFID-enabled application.

Chapter 6: Node-Centric Distributed Dataflow Programming discusses the second contribution of this work: a programming library offering a number of distributed dataflow language constructs on top of AmbientTalk/R: a variant of AmbientTalk with support for dataflow programming. We apply these constructs to our running example and observe that although many issues with event-driven code are alleviated, the control flow of the application becomes very implicit due to the dataflow evaluation model. In this chapter, we also go over the implementation of these constructs in AmbientTalk/R.

Chapter 7: Network-Centric Visual Dataflow Programming introduces AmbientTalk/R^V: a visual network-centric language that uses the dataflow paradigm to coordinate mobile event-driven applications from a network-centric point of view to alleviate the coordination issues mentioned in the foregoing chapter. This language forms the third and last contribution of our work. The visual

language is again applied to our mobile RFID-enabled application and its implementation is discussed.

Chapter 8: Conclusion summarizes the contributions made in this dissertation. At that point we are able to evaluate the contributions of the dissertation with hindsight, naturally leading to a discussion on the limitations of this work and on possible directions for future research.

1.6 Summary

This dissertation is about applications that are distributed in mobile ad hoc networks and that are highly event-driven. Mobile RFID-enabled applications in particular rely on RFID technology to not only interact with nearby computers, but with nearby RFID-tagged everyday objects as well. Such applications are the pinnacle of mobile event-driven applications as RFID hardware acts both as a sensor generating events and as a network interface towards networked objects generating network events.

We will describe how a problem with event-driven applications called inversion of control that is solved in local event-driven applications is not solved for mobile RFID-enabled applications, which are event-driven applications distributed in a mobile ad hoc network.

For tackling the distributed nature of these applications, we rely on ambient-oriented programming: a programming paradigm specifically designed for mobile ad hoc networking applications. We extend a concrete ambient-oriented programming language to consider mobile RFID-enabled applications as distributed applications where physical objects are represented as and unified with remote software objects. This allows the programmer to treat passive computing devices such as RFID tags as any computing device hosting a software object and to treat RFID failures just as any network failure. Due to the massive scale in which RFID tags can be present and due to high rate of failures when interacting with them, to remain causally connected to the physical environment, these mobile RFID-enabled applications are highly event-driven or reactive, aggravating the inversion of control problem.

For tackling the reactive nature of these applications, we resort to a paradigm specifically targeting highly event-driven applications called dataflow programming. We describe an integration of both paradigms which we shall name *ambient-oriented dataflow programming* and use the resulting language constructs for implementing a mobile RFID-enabled application. We propose two levels of ambient-oriented dataflow programming: *node-level* dataflow for finer-grained reactive code and *network-level* dataflow for specifying the global control flow and higher-level distributed dependencies of mobile event-driven applications.

In the next chapter, we go into the properties of mobile RFID-enabled applications in more detail and relate them to previous research on mobile ad hoc networks on which this work is based, namely ambient-oriented programming. This new application of ambient-oriented programming leads us to refine its programming model requirements to support mobile RFID-enabled applications as well. We continue in chapter 3 by surveying technologies that deal with reactivity of applications distributed in mobile ad hoc networks and investigating to which extent they satisfy our programming model requirements for mobile RFID-enabled applications. We take inspiration from these surveyed systems to lay out an approach to satisfy the requirements for mobile RFID-enabled applications that were not satisfied by the ambient-oriented pro-

gramming paradigm. Subsequently, in chapter 4, we detail AmbientTalk, a concrete ambient-oriented programming language that we choose to extend for mobile RFID-enabled applications. In chapter 5, we extend AmbientTalk to support mobile RFID-enabled applications, increasing the scale in which AmbientTalk applications have to be made event-driven. In the subsequent two chapters, we introduce ambient-oriented dataflow programming: in chapter 6, we introduce our node-level ambient-oriented dataflow abstractions and in chapter 7 AmbientTalk/R^V: our network-centric ambient-oriented dataflow language.

Chapter 2

Mobile RFID-Enabled Applications

This dissertation targets a specific class of software applications that we dub *mobile RFID-enabled applications*. Mobile RFID-enabled applications emerge due to an increasing embedding of RFID hardware in both everyday objects and mobile computing devices such as smartphones.

The combination of mobility and event-driven execution introduces a mismatch with existing programming technology that renders these applications hard to implement. The *mobile* aspect of these applications lies in the fact that spontaneously interacting devices need some form of wireless communication that does not restrict their movement. The *event-driven* aspect lies in the fact that these applications have to be responsive to the *events* (coming from example from RFID equipment) occurring in their surroundings. The combination of both aspects emerges as different applications notify each other of their internal application-level events or of events that they have detected using their (RFID) sensors by means of *mobile ad hoc networking* technology.

In this chapter, we start in section 2.1 by giving an overview of technologies available today that give rise to mobile RFID-enabled applications. First, mobile ad hoc networks are discussed in section 2.1.1. Subsequently, the combination of RFID technology with mobile devices into mobile RFID-enabled applications is discussed in section 2.1.2. The approach that we took in our work is to extend an existing programming paradigm targeting mobile ad hoc networks to support mobile RFID-enabled applications as well. This paradigm is called the ambient-oriented paradigm and is discussed in section 2.2. Finally, section 2.3 concludes this chapter by discussing the programming requirements that we put forward to be able to concisely implement mobile RFID-enabled applications. These requirements naturally follow from extending the ambient-oriented programming paradigm to better support highly event-driven software such as mobile RFID-enabled applications.

2.1 Enabling Technologies

Event-driven software is not a new concept. Any computer program that has to react to external stimuli is event-driven. This includes desktop applications with a graphical user interface, web servers, driving assistance systems in cars, etc. Why then investigate a subset of this type of software called mobile RFID-enabled applications? Mobile

devices are equipped with different wireless networking interfaces (WiFi, Bluetooth, 3G...) and a wide range of sensors (GPS, accelerometers, gyroscopes, RFID readers...). Because of their small size, their ubiquity and their strongly varying sensing technology, these devices make possible applications that are so strongly embedded in the physical world that they bring event-driven software to a whole new level. In these applications, the portion of an application's logic that deals with responding to external events and with communicating events to other devices or applications is larger and more important and because of the open nature of these applications the heterogeneity of the events is much higher.

Investigating all sensor and wireless communication technologies is clearly unfeasible and prone to becoming obsolete very soon. Instead, we focus on the software engineering issues that are induced by mobile RFID-enabled applications. In this section, we investigate what their implications are on software.

2.1.1 Mobile Ad Hoc Networks

Mobile ad hoc networks are computer networks that are formed when *mobile devices* dynamically establish a network connection when they are in each other's range by using their wireless communication facilities. They were also termed *device networks* by Bonnet et al. [BGS00]:

> "The widespread deployment of sensors, actuators and mobile devices is transforming the physical world into a computing platform. We will soon find computing power, memory and communication capabilities on temperature sensors and motion detectors, on door locks, light bulbs and alarms, on every cellular phone, in every vehicle, and soon in every person's wallet or key ring. Emerging networking techniques ensure that devices are interconnected and accessible from local- or wide-area networks."

The ad hoc-ness stems from the fact that these devices do not necessarily know each other beforehand, but simply establish a network connection by physical colocation. Similarly, disconnections between mobile devices in the mobile ad hoc network can occur at any point in time when two devices move out of each other's communication range. Hence, a mobile ad hoc network is a continuously changing graph of mobile nodes. Mobile ad hoc networks do not assume any infrastructure to assist the mobile nodes in maintaining a more stable network, such as naming servers, routers, access points etc. As a result, interaction among mobile nodes in the mobile ad hoc network must happen over unstable peer-to-peer connections.

One may argue that in times of internet connectivity over cellular networks and other (almost) pervasive uplinks to the internet, mobile ad hoc networks are overly restrictive. However, it should be noted that in the context of ubiquitous computing, mobile ad hoc networks are formed by a much broader range of devices than laptops, smartphones, etc. With current day hardware, we already see plenty of mobile ad hoc network applications. Some can be rather heavy-weight, such as media centers that automatically play content from dynamically discovered storage devices (i.e. MP3 players) or mobile computing devices that trigger interaction with a car's onboard computing infrastructure. However, our definition also applies to much more light-weight setups, such as tiny sensor nodes or RFID tags. The latter are the topic of section 2.1.2 below. Hence, we do not restrict our discourse to classic IP-networks.

Hardware Characteristics

As mentioned earlier, mobile ad hoc networks consist of mobile devices that are connected via wireless communication links which are established when these devices move in each other's communication range. Mobile ad hoc networks composed of mobile devices and wireless communication links exhibit a number of phenomena which are rare in their traditional, fixed counterparts. In his original work, Dedecker identified four discriminating "hardware phenomena" that are inherent to mobile networks [DVM+06]. These four phenomena were later reduced and generalized into only the two following fundamental ones by Van Cutsem in his thesis [Van08]:

Volatile Connections. Mobile devices equipped with wireless communication media only have a limited communication range. This implies that communicating devices may move out of earshot at any time because of the happenstance of node mobility. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Such network connections are also said to be intermittent [MCE02]. Quite often, transient network partitions should not terminate a distributed interaction. Rather, communicating parties in an ad hoc network are more interested in continuing their collaboration when the connection is restored. They expect communication to work in the presence of volatile connections. Dealing with partial failures is not a new ingredient of distributed systems, but mobile ad hoc networks expose applications to a much higher rate of partial failure than traditional distributed systems. In mobile networks, disconnections become so omnipresent that they should be considered the rule, rather than an exceptional case.

Zero Infrastructure. Mobile ad hoc networks have no or very little fixed infrastructure [MCE02]. Networks are formed by the mere colocation of mobile nodes. In such networks, the services available to an application thus depend on the host devices location. As devices move, services may spontaneously appear and disappear as their hosts join with and disjoin from the ad hoc network. Moreover, a mobile ad hoc network is often not globally administered by a central authority. In stationary networks, applications usually expect services to be available and accessible via an a-priori known URL or similar designator. In a mobile ad hoc network, applications have to find their required services dynamically in the environment. Services have to be discovered on proximate devices, possibly without the help of shared infrastructure.

The two other characteristics identified by Dedecker are [DVM+06]:

Autonomy. Most distributed applications today are developed using the client-server approach. The server often plays the role of a "higher authority" which coordinates interactions between the clients. In mobile networks a connection to such a "higher authority" is not always available. Every device should be able to act as an autonomous computing unit.

Natural concurrency. In theory, distribution and concurrency are two different phenomena. For instance in a client-server setup, a client device might explicitly wait for the results of a request to a serving device.

Van Cutsem argues that autonomy is a corollary of the lack of shared infrastructure [Van08]. Without a shared infrastructure, devices (or more accurately their applications) necessarily need to be autonomous. The fact that they should be autonomous follows from the fact that they cannot depend on infrastructure, because there is (mostly) none available. Ad hoc applications have to be structured such that they can cope with necessary resources or services being unavailable for an extended period of time.

Natural concurrency already follows from the fact that applications are deployed on a distributed computer network. Entirely sequential distributed applications require every distributed part of the application to wait until every other distributed part completed a single task, which blocks parts of the application that could perfectly make progress and hinders scalability of the application. Hence, any ad hoc networking application is essentially distributed and concurrent.

Implications on Software Engineering

The aforementioned hardware characteristics of mobile ad hoc networks have effects that pervade any distributed application that has to be deployed on these networks. They have effects on the structure of applications, on how distributed application components interact and on how application data can be represented. Because these effects are so profound, they cannot be hidden [WWWK96] from mundane application programmers. As a result, software technologies for mobile ad hoc networks will have to support them intrinsically.

Resilience to Volatile Connections Because of the volatile connections that interconnect application components, interactions between these components must be resilient to the frequent disconnection of communication partners. Moreover, because network connections are established by the physical colocation of mobile devices, the fact whether there is a connection or not can be semantically relevant to the application logic (e.g. the fact whether an RFID tag is “present” at a certain location or not). Hence, although applications have to be resilient to disconnections, in many cases they also have to be notified of changes in the status of the connections with their peers. Unfortunately, these requirements all have to be implemented “by hand” when using existent general-purpose programming technology.

Loose Coupling of Concurrent Application Components The lack of infrastructure and the presence of volatile connections in mobile ad hoc networks imply that applications cannot designate specific tasks to or assume certain services from a fixed number of a-priori known peers. Instead, mobile ad hoc network applications should be structured in a loosely-coupled way, both from an architectural point of view as from an execution-level point of view. On the architectural level, applications should be composed in such a way that allows discovering services at runtime without relying on infrastructure. Additionally, it should be possible to discard unavailable services or replace them with equivalent ones present in the network. On the execution level, to guarantee responsiveness, applications that lose the connection with some service that they require should not remain blocked if they have other useful work to do, even if no replacement service is immediately found. This means that all application components distributed in the mobile ad hoc network should run concurrently and only communicate using non-blocking communication primitives. Blocking on the reappearance of an unavailable service would halt the entire application.

Service Announcement Broadcasting and Notification In the above paragraph we pointed out that applications distributed in a mobile ad hoc network should be structured as concurrent components that can discover each other at runtime. Since there is no infrastructure assumed, these components are responsible themselves to announce which service they offer to a continuously varying set of clients that is not known beforehand. Similarly, clients that require one or more services are responsible themselves to detect these announcements and in response trigger the appropriate actions. In such an infrastructureless network setup, the availability of a service has to be announced to interesting parties by broadcasting advertisements.

Conclusion

In this section, we have given an overview of the hardware characteristics of mobile ad hoc networks and their implications on software engineering, without committing to any specific implementation technique or programming model. The key insights are that a suitable programming model should be:

- Resilient to volatile connections,
- Loosely-coupled and concurrent by employing non-blocking communication,
- Dynamically reconfigurable by broadcasting and reacting to service announcements.

This means that important parts of the application logic are driven by external events, e.g. network (dis)connections, services being discovered or discarded, or incoming communications that have to be handled, etc.

2.1.2 RFID Technology

An RFID system typically consists of one or more RFID readers and a set of RFID tags. The RFID reader is a device that broadcasts a radio wave on a specific frequency. RFID tags that are in communication range respond by powering up their circuits using the energy of the radio wave, and replying by sending back a radio signal themselves. These types of RFID tags are called *passive* tags [CH07]. In most cases, passive RFID tags have some form of on-chip memory, to which arbitrary data can be written. Usually, the response of an RFID tag simply consists of a read-out of its memory together with its unique serial number. An RFID reader willing to write on a tag has to broadcast the data it is willing to write together with the serial number of the tag it is willing to store the data on.

The most prominent characteristic that sets apart different types of RFID hardware is the communication range. The failure rate of reading and writing data from and to tags depends for the most part on the communication range, which in itself depends on different factors. First of all, there is the frequency of the radio signal. Higher frequencies allow longer communication ranges, but also require bigger, higher-powered RFID readers. Additionally, they require RFID tags that respond to these high frequencies over longer ranges, which implies that they have to respond to a strongly degraded signal. Because of this reason, tags that work over the highest frequencies (and longest communication ranges) offer a very limited amount of memory that requires less power to be read. A second point of variation is the power level on which radio signals are broadcasted. Without changing the signal frequency, it shows roughly the same effect:

higher power levels allow longer communication ranges but increase the size, power usage, and price of the RFID reader. It should come as no surprise now that more on-tag memory requires more power to be successfully read and/or written and requires closer communication ranges.

An alternative consists of powering up RFID tags with their own power source, such as a battery, but other technologies are close to being taken in production [PS05, KK07, BA08, Arn07, BTT+07, KKPG98]. These types of tags are called *active* tags. Their power source allows them to power up more memory, operate over longer ranges and to offer more reliable communication [Wei05]. They are of course more expensive. The most expensive ones can even have limited processing power, initiate communication autonomously, communicate with other active tags or have integrated sensors, blurring the distinction between RFID networks and sensor networks [SYP+08].

Although these advanced forms of RFID technology have interesting applications, we will focus in this text on cheap, passive, writable tags. First, because these tags are cheap, tagging large amounts of everyday objects is realistic. Second, it is easier to gradually extend a light-weight programming model targeting constrained hardware to hardware with more possibilities than to adapt a heavy-weight model to more restricted hardware.

Regardless of the issues that arise in classic RFID applications, RFID technology is generally considered as a key technology in developing ubiquitous, context-aware applications [WJ09, Ble06]. The reason is that RFID technology is a form of computing technology that is one of the cheapest and easiest to integrate in everyday objects [RGDMC09]. On the one hand, RFID tags are becoming so cheap that it will soon be possible to tag one's entire environment, thereby wirelessly dispersing information to nearby context-aware applications. On the other hand, RFID readers are also becoming increasingly power-efficient while costs and form factors drop, to the extent that RFID readers are being integrated in the latest generation of smartphones [PPS+05, TP05]. Furthermore, tag memory is increasing [Mat05]. At the time of writing this text, the largest passive tags offer about 32Kb of memory [PS11]. Hence, more and more application-specific data coming from mobile devices can be written on these ubiquitously available tags. Although RFID technology was originally developed and used as an electronic bar code, the communication between mobile devices and tags that are interconnected by mobile ad hoc networks gives rise to *mobile RFID-enabled applications*. These applications use RFID technology in a radically different way than RFID systems deployed today, which only use RFID tags as digital barcodes and almost never exploit the writable memory on these tags [PS11, DMS07, MTCS07].

Usually, RFID technology provides some low-level protocols to deal with issues such as acknowledgements, security [PKH05], collision detection [SSYH06], etc. Although there are many important issues still to be resolved, they are not in the scope of this dissertation. We focus on the higher level software engineering issues instead. It should come as no surprise that in mobile applications where no assumptions are made with respect to communication ranges and available infrastructure, these peculiarities of the underlying hardware or low-level communication protocols can never be hidden from the software level.

Hardware Characteristics

Interactions with RFID hardware exhibit a number of phenomena that cannot be hidden from the application level. Just like with mobile ad hoc networks, they stem from the fact that mobile devices are interacting over unstable wireless connections. Below,

we argue that RFID technology gives rise to highly volatile types of networks which exhibit similar phenomena as mobile ad hoc networks, but with an increase in scale of the fluidity and dynamicity of the network topology.

Volatile Connections Although in some cases mobile devices might have the possibility to connect to a reasonably stable network (e.g. the internet through an internet access point), we take as the base case a much more light-weight type of communication links to cater to much more spontaneously formed connections, such as maintaining a communication link with an RFID tag by periodically broadcasting a radio signal and listening for its response. Abstractly spoken, we consider wireless communication among mobile devices to have a limited range, in which case communication links can break at any point in time because of the roaming of devices. This can range from bad 3G coverage on a mobile phone to an RFID-tagged object which is moved away from an RFID reader.

These disconnections are not always permanent and, especially in more light-weight types of communication links such as RFID, network connections can exhibit very intermittent behavior. A metal object passing through the line of sight between an RFID tag and RFID reader is enough to temporarily disrupt communication. Network connections in both mobile ad hoc networks and RFID systems are hence said to be *volatile*. Of course, the intermittent failure of volatile connections should not halt or break a distributed application communicating over these connections with other (possibly mobile) devices. In classical distributed computing systems this is often the case because failures are rare, and if they occur, they are permanent or long-lasting. Both in mobile ad hoc networks and RFID systems, where volatile connections exhibit intermittent behavior, software should be resilient to these failures by considering them the rule instead of the exception.

Little or No Infrastructure Just like mobile ad hoc networks, RFID networks are formed spontaneously by the mere colocation of mobile nodes. This means that these networks are not globally administered by a central authority because this would hinder the spontaneous emergence of these networks. Hence, RFID-tagged objects present in the network cannot be looked up by means of such a central authority, for example a backend database which stores the properties associated with RFID-tagged objects. Just like in mobile ad hoc networks, RFID-enabled applications must instead discover RFID-tagged objects at runtime by scanning the physical environment. Furthermore, objects can in their turn roam, causing them in their turn to move in and out of range at any point in time. This means that the set of nearby RFID-tagged objects is in constant flux.

Implications on Software Engineering

In this section, we will discuss the implications that the hardware characteristics of RFID have on programming technology.

Mobile RFID-Enabled Applications are Mobile Ad Hoc Network Applications Communication with RFID tags is prone to many failures. Tags close to each other can cause interference and can move out of the range of the reader while communicating with it. These failures may be permanent, but it may be that at a later moment

in time the same operation succeeds because of minimal changes in the physical environment. For example, a tag moves back in range or suddenly suffers less from interference. These failures in combination with the fact that maintaining a connection to a tag must happen by broadcasting a signal and listening for the tag's reply cause connections to RFID devices to exhibit very intermittent behavior. Just like in mobile ad hoc networks, communication primitives used to communicate with RFID tags have to be resilient to volatile connections.

Before being able to communicate, RFID-tagged objects announce their presence by simply broadcasting to all reachable devices equipped with an RFID antenna. Hence, RFID-tagged objects could be considered as a simple remote services that have to be dynamically discovered in a mobile ad hoc network.

These two observations lead us to the conclusion that RFID technology is a restricted form of mobile ad hoc networking technology and that programming technology for mobile RFID-enabled applications should offer the same support as programming technology for mobile ad hoc networking applications for dealing with the issues described above.

Mobile RFID-Enabled Applications are Highly Event-Driven Applications Because the set of RFID tags in the range of an RFID reader is continuously fluctuating, it becomes impractical to use polling on the application level as the mechanism to detect the current set of tags in range. As a result, mobile RFID-enabled applications have to be conceived as event-driven applications from the ground up. Phenomena that cannot be abstracted away – such as the appearance and disappearance of tags and the reception of acknowledgments from these tags – become the driving forces of the application, contributing to the highly event-driven nature of RFID-enabled applications.

This, however, is where mobile RFID-enabled applications are set apart from classic mobile ad hoc networking applications. Mobile RFID-enabled applications can interact at any point in time with a number of RFID tags that is orders of magnitude larger than the number of mobile devices in classic mobile ad hoc networking applications. Moreover, because RFID connections exhibit a higher volatility than more heavy-weight wireless connections, the number of connection and disconnection events per remote connection is much higher as well.

Conclusion

RFID technology exhibits two properties that mobile ad hoc networks exhibit as well: connections between RFID-tagged objects and an RFID-enabled device are volatile and RFID-tagged objects are detected spontaneously be devices without relying on additional infrastructure. What sets apart mobile RFID-enabled applications is that, due to the possible ubiquity of RFID-tagged physical objects, the scale in which objects are detected, connections are formed, broken and restored again is much greater. Even to such an extent that the set of nearby RFID-tagged objects cannot be treated as an enumerable structure, but as a pool of objects that is in constant flux and of which the changes drive the application's execution.

Still, much work has been carried out in coming up with suitable programming models for mobile ad hoc networks. In this work, we investigate how such an existing programming model can be adapted to mobile RFID-enabled applications. Our work is based on the ambient-oriented programming paradigm, which is discussed in the next section.

2.2 Ambient-Oriented Programming

Any application that is to be deployed on a mobile ad hoc network has to deal with its hardware phenomena. What is important to note is that the phenomena are universal, in the sense that they do not depend on the particularities of a specific application. Exactly because of this universality, it is worth investing in a computational model that facilitates distributed programming for mobile ad hoc networks by taking these phenomena into account from the ground up. Such a computational model could be implemented as a new language, library or middleware platform. However, because the effects engendered by partial failures and the absence of remote services tend to affect and pervade the entire application, the above phenomena are not easily hidden behind traditional library abstractions. Therefore, distribution is often dealt with in dedicated middleware or programming languages. This formed the motivation to define ambient-oriented programming as a new programming paradigm tailored towards mobile ad hoc networks and the design of new languages fitting this paradigm.

Ambient-oriented programming is a novel programming paradigm proposed by Dedecker et al. [DVCM+05, DVM+06]. Prior research on ambient-oriented programming led amongst others [BVT+09, VGC+10, VCVCDM09] to AmbientTalk [VMG+07], a programming language that adheres to the original ideas of the paradigm. The work described in this dissertation builds upon both the concepts of ambient-oriented programming and the technical foundation laid by the AmbientTalk language. Therefore, before presenting our work in the later chapters of this dissertation, we first give an overview in this chapter of the ambient-oriented programming paradigm. In chapter 4 we discuss the second incarnation of the AmbientTalk language: AmbientTalk/2.

As discussed in the introduction, mobile ad hoc networks constitute the network technology of ubiquitous computing. In a nutshell, the most difficult problems are that network connections between devices are unstable due to the limited wireless communication range of participating devices, that the network is open – (new) devices frequently join and leave the network – and has little or no infrastructure (e.g. to support globally accessible servers).

Contemporary distributed programming languages, middleware and libraries offer abstractions that are built with different assumptions of the properties of the underlying communications network. For example, in a conventional distributed system, a network partition is regarded as a failure, i.e. an exceptional event. In a mobile network, disconnections between devices become the norm, rather than the exception. This change in the physical nature of the network percolates all the way up to the application layer [MCE02]. Current abstractions cannot optimally cope with these changes. For example, handling failing communication using the classic programming language abstractions of exception handling results in application code that is literally polluted with failure handling. An ambient-oriented language foregoes such issues by offering abstractions in order to fit the hardware phenomena of the underlying network itself [DVM+06].

2.2.1 Ambient-Oriented Programming Criteria

As our work builds on the ambient-oriented programming paradigm, we revisit the criteria put forward by the ambient-oriented programming paradigm aimed at tackling the aforementioned phenomena. In his dissertation, Dedecker extensively motivates why previously developed programming languages and middleware are not readily suitable

for mobile ad hoc networks, because they do not directly address the hardware phenomena described above [DVM+06]. In reaction, he defines the ambient-oriented programming paradigm. A language is ambient-oriented if it exhibits a specific set of characteristics, which are extensively discussed elsewhere [DVCM+05, DVM+06, Ded06]. Additional to these basic criteria, Van Cutsem introduced in his thesis additional criteria for *coordination* in mobile ad hoc networks [Van08]. Similarly, the motivation behind criteria for coordination is that, in mobile ad hoc networks, the state of the network is in constant flux because devices move about in unpredictable ways. Abstractions for mobile ad hoc networks should allow applications to abstract from this underlying physical state when communicating because it can make communication more resilient in the face of temporary disconnections.

We summarize these criteria below and discuss how they prove useful for mobile RFID-enabled applications.

Classless Object Models

An ambient-oriented language disallows the use of classes as they are employed in traditional class-based languages like Java and Smalltalk. In such languages, when objects are copied between devices (to share information, e.g. using parameter-passing), the class has to be copied as well. Hence, a single class can become duplicated across several devices in the network. In a volatile network like a mobile ad hoc network, it becomes impossible to implicitly ensure that all of these duplicate classes are kept in synchronization. However, this impossibility breaks the invariant that all instances of the same class should behave the same way: two instances of the same class may behave differently on different machines.

An ambient-oriented language avoids these problems by requiring objects to be entirely self-sufficient (containing their own code and data). When objects are copied between hosts, they are self-descriptive and need no shared external entity (the class) to be duplicated. Of course, this solution is not a "silver bullet" either: if a class-like sharing relationship between objects is required, it must now be explicitly encoded by the programmer, who is himself responsible for the consistency of this relationship in the face of volatile connections.

Classless object models usually fall under the prototype-based category. Other than the reasons dictated by the ambient-oriented programming paradigm to select a classless object model, table 2.1 compares the class-based and prototype-based model by means of their other advantages and disadvantages.

In this work, we take no stance towards a particular approach, but follow the ambient-oriented programming paradigm in its requirement for a classless object model. Some situations work better with one approach and other ones with the other, as mentioned by Stein, Lieberman and Ungar in their "Treaty of Orlando: A Shared View of Sharing" [SLU89]:

> "[...] that no definitive answer as to what set of these choices is best can be reached. Rather, that different programming situations call for different combinations of these features: for more exploratory, experimental programming environments, it may be desirable to allow the flexibility of dynamic, explicit, per object sharing; while for large, relatively routine software production, restricting to the complimentary set of choices – strictly static, implicit, and group-oriented – may be more appropriate."

Class-based	**Prototype-based**
Classes can denote types in strongly-typed and statically-typed languages.	No strong or static typing.
State easier to encapsulate and protect, classes specify and enforce scoping.	State harder to encapsulate and protect, the uniformity of the object model makes it harder to find a consistent place to specify and enforce scoping (e.g. relying on lexical scoping of ”constructor” procedures).
Inheritance chain cannot be changed at runtime: less flexible with respect to sharing (only anticipated sharing is supported), but safer.	Inheritance chain can be changed at runtime: more flexible with respect to sharing (unanticipated sharing is supported as well), but less safe.
Objects are not self-contained, but copying objects does not require copying their behavior.	Objects are self-contained, but copying objects requires copying their behavior.
More structural guarantees, not flexible with respect to structure, changing the class changes all instances.	Fewer structural guarantees, more flexible with respect to structure, allows to change the structure of a particular object “on-the-fly”.
Unique objects (such as booleans) need a class.	Unique objects (such as booleans) can be constructed without building a class.

Table 2.1: Comparison of the class-based model with the protoype-based model.

Relation with mobile RFID-enabled applications For mobile RFID-enabled applications, it makes sense to represent an RFID-tagged physical object as a software object. Spontaneously detecting and interacting with such a software object representing an RFID-tagged object should be possible without first installing the required class hierarchy. Hence, a classless object model is useful to deal with the open nature of mobile RFID-enabled applications by making RFID-tagged physical objects self-contained.

Non-Blocking Communication Primitives

All distributed object-oriented programming languages have primitives for sending and receiving messages across the network. An ambient-oriented language requires these primitives to be “non-blocking”: a process or thread of control should not be suspended if the operation cannot be completed immediately. This requirement is based on the fact that in a wireless ad hoc network, communicating parties can often be unavailable, and making a communication operation block until the communicating party is available may lead to unacceptable delays. Furthermore, blocking primitives increase the risk of distributed deadlocks which are hard to detect in an ad hoc network, because the network lacks a global coordinator to detect or break the cycle.

Non-blocking message sending is better known under the term asynchronous message sending. More specifically, it is the form of asynchronous message sending where the sender does not even have to wait for the message to be transmitted to the receiver. Non-blocking message reception is better known as event-driven computation. A non-blocking receive operation enables a process to register its interest in a certain type of

message without blocking the control flow. In an event-driven application, the focus of control lies outside of the application; the control flow is determined by external events, rather than being encoded within the application itself.

Relation with mobile RFID-enabled applications Similarly, for mobile RFID-enabled applications, communication with an intermittently connected RIFD-tagged physical object should not block the application when the RFID-tagged object is temporarily out of communication range.

Reified Communication Traces

The above requirement of making all communication between processes non-blocking implies that devices are no longer implicitly synchronized while communicating. If communication primitives no longer synchronize processes implicitly, other means must be provided to do so for some distributed computations to make sense. The ambient-oriented programming paradigm re-introduces synchronization by requiring information about the communication traces of processes to be reified (i.e. made explicit). By communication traces, we mean for example the messages that a process successfully received in the past or the messages that are still pending to be sent, etc.

Decentralized Service Discovery

As mentioned earlier in this section, mobile ad hoc networks have no infrastructure, which requires devices to detect services dynamically as they are roaming. In addition, services may be "anonymous": they have no a priori known address or URL by which they can be addressed. An ambient-oriented language should therefore include language features to communicate with anonymous services.

Decentralized service discovery also entails a mechanism that allows a program to keep track of which services become available and unavailable as devices roam. An ambient-oriented language requires this mechanism to be peer-to-peer because of the lack of infrastructure: devices must be able to advertise their own services directly, without reference to third party servers. Note that this does not imply that an ambient-oriented program must be peer-to-peer as a whole: it is always possible to structure applications according to a client-server pattern. Ambient-oriented programming only states that applications should not be forced to use a client-server setup in order to interact.

Relation with mobile RFID-enabled applications Mobile RFID-enabled applications naturally accommodate decentralized service discovery. When in range of an RFID-enabled device, the RFID tag of an RFID-tagged physical object will be powered up and spontaneously initiate interaction. A mobile RFID-enabled application must be able to keep track of which RFID-tagged objects are currently in and out of range.

Decoupling in Time

The volatile connections in mobile ad hoc networks require communication models that can abstract from the network connectivity between communicating processes. It should be possible for two processes to express communication independently of their connectivity. This significantly reduces the case-analysis for the programmer, which

can reason in terms of a fully connected network by default, and can deal with border cases in an orthogonal way. Hence, **communicating processes do not necessarily need to be online at the same time**.

Decoupling in time is achieved either by synchronizing processes until a connection is available (which, as is argued below, is not a very scalable solution in a mobile ad hoc network) or by storing sent messages in an intermediary data-structure. This makes it possible for communicating parties to interact across volatile connections, because the logical act of information sending is decoupled from the physical act of information transmission, allowing for the information to be saved and transmitted at a later point in time when the connection between both parties is restored.

Relation with mobile RFID-enabled applications Exchanging data with an intermittently connected RFID-tagged physical object is prone to many failures. In many cases, multiple attempts at reading to or writing to an RFID tag's memory are needed before an operation succeeds. This should happen without immediately signaling an error for every single fault to the programmer. Instead, the implementation should retry these operations without blocking the application of signaling an error. Due to physical phenomena, a small change in the physical environment may cause these operations to succeed after a short timespan.

Decoupling in Space

Decoupling in space implies that communicating processes do not necessarily need to know one another's exact address or location in order to collaborate. It is directly motivated by the scarcity of infrastructure in a mobile ad hoc network, making the reliance on servers to mediate collaborations impractical. A second motivation for space-decoupled communication is that it enables applications to adapt more gracefully to changes in their physical environment. In mobile networks, equivalent services may be hosted by different devices. As a device roams, it may use different instances of conceptually the same service. For example, in a city tour application, the handheld device of a tourist may connect to the same tourist information service via different access points physically dispersed throughout the city. A communication model that is decoupled in space supports such transitory relationships, because it allows one to make abstraction from specific service instances. Hence, **communicating processes do not necessarily need to know each other beforehand**.

Decoupling in space implies a form of anonymous communication, often implemented by a form of communication where senders and receivers of data are matched on the content of the data itself, such as in content-based publish/subscribe. A prototypical example of this is communication via tuples in tuple spaces. Gelernter refers to space decoupling in the context of tuple spaces as "distributed naming" [Gel85].

Finally, decoupling in space is closely related to decentralized discovery, in the sense that decentralized discovery is usually implemented in terms of communication which is decoupled in space. However, decoupling in space does not imply a decentralized form of discovery. For example, in the Linda coordination language [Gel85], processes are decoupled in space but there is no form of discovery to connect decentralized distributed processes.

Relation with mobile RFID-enabled applications RFID-tagged objects are spontaneously discovered by RFID-enabled devices. It is up to the application to decide based

on the data on the RFID tag whether to interact with the object. To not rely on additional infrastructure, this should be possible without contacting a backend database.

Synchronization Decoupling

In a mobile ad hoc network, an application may find itself deprived of access to a certain service or resource for extensive periods of time. For example, in a collaborative meeting application, the application only has access to the calendars of other people when they are physically nearby. Such extensive periods of time between potential periods of interaction suggest that synchronization between different parties should be performed without blocking their control flow (i.e. without suspending their thread of control). Blocking synchronization can lead to applications which remain unresponsive for extensive periods of time. A reactive synchronization style is more appropriate in a mobile ad hoc network, as it leaves processes responsive to other events while waiting for (information provided by) another process. Therefore, **the control flow of communicating processes must not be blocked upon sending or receiving**.

Synchronization decoupling implies that a sender can employ a form of asynchronous message passing, such that the act of message sending becomes decoupled from the act of message transmission. Likewise, allowing recipients to process messages asynchronously decouples the act of message reception from the act of message processing. Message transmission and reception require a connection between sender and receiver, but message sending and processing can be decoupled, allowing communicating processes to abstract from the fact that the other process is online or not.

Relation with mobile RFID-enabled applications Exchanging data with an intermittently connected RFID-tagged physical object is prone to many failures. In many cases, multiple attempts at reading to or writing to an RFID tag's memory are needed before an operation succeeds. This should happen without immediately signaling an error for every single fault to the programmer. Instead, the implementation should retry these operations without blocking the application of signaling an error. Due to physical phenomena, a small change in the physical environment may cause these operations to succeed after a short timespan.

Arity Decoupling

In mobile networks, groups of devices are often not statically determined, but are rather formed ad hoc as devices roam. One is often interested in communicating with only the proximate devices. The number of such proximate devices is not a priori known to the application. It is therefore important that interactions can be expressed without explicit reference to the number of participants. A good coordination mechanism should enable the programmer to express interactions with a varying number of participants, e.g. one-to-one, one-to-many or many-to-many interactions. In short, **Processes do not necessarily need to know the total number of processes communicated with**.

Van Cutsem uses the term arity decoupling [Van08] to attribute forms of communication that can target more than one recipient, without explicitly specifying the total number of recipients. Arity decoupling is a special case of space decoupling. Space decoupling can be regarded as abstracting from both the identity of receivers and the total number of receivers [EFGK03]. These two aspects are treated separately because both concepts are orthogonal in principle. A communication abstraction can be anonymous yet specify a maximum number of receivers.

Finally, note that the arity of the interaction influences the synchronization of the participating processes. In a point-to-point interaction, a process often needs to await a reply from a previous request before being able to continue. However, in a one-to-many interaction, different kinds of synchronization abstractions are called for.

Relation with mobile RFID-enabled applications In mobile RFID-enabled applications, the set of reachable RFID-tagged objects is in constant flux. To prevent the programmer from having to enumerate all these tagged objects extensionally in collections and having these collections to be constantly kept manually in sync with the physical environment, it should be possible to designate RFID-tagged objects both individually and using intensional descriptions. In the latter case, the set of matching objects must be automatically kept in sync without programmer intervention.

2.3 Programming Model Requirements

Given the properties of mobile ad hoc networks and mobile RFID-enabled applications described earlier in this chapter, in this section we establish the programming model requirements for mobile RFID-enabled applications. These requirements will be used in the next chapter to evaluate how or to which extent the current state of the art in different research areas manages to address them.

Our requirements can be classified into the three purposes listed below. First, we look how the ambient-oriented programming paradigm can be used to abstract over the bare RFID hardware, given the similarities that we identified between mobile ad hoc networks. Subsequently, we take into account the decoupling requirements of mobile ad hoc networks themselves. Finally, the highly event-driven nature of mobile RFID-enabled applications presents a number of challenges which have to be addressed as well.

2.3.1 Ambient-oriented Programming with RFID-tagged Objects

As we mentioned earlier in section 2.1.2, we consider mobile RFID-enabled applications as a special case of mobile ad hoc networking applications in which some software objects are used to represent real-world physical objects tagged with a (passive) RFID tag. In order to abstract over RFID hardware details, ambient-oriented programming for mobile RFID-enabled applications should allow:

Addressing physical objects. RFID communication is based on broadcasting a signal. However, to be able to associate a software entity with one particular physical object, it is necessary to address a single designated physical object and maintain a (conceptual) connection with the object.

Storing application-specific data on RFID tags. In the spirit of mobile ad hoc network applications, we assume as little infrastructure as possible to implement mobile RFID-enabled applications. Hence, mobile RFID-enabled applications should be able to work without relying on a backend database and therefore it should be possible to store the application data in the writable memory of the RFID tags themselves [RK09, FL05, PS11, MTCS07].

Reactivity to appearing and disappearing objects. It is necessary to observe the connection, reconnection and disconnection of RFID tags to keep the mobile RFID-enabled application synchronized with its physical environment. Differentiating

between connection and reconnection is important to preserve the identity of physical objects in the application. Furthermore, it should be possible to react upon these events from within the application. Notification of these events should be possible in a mobile ad hoc network (i.e. without assuming a fixed infrastructure).

Asynchronous communication. To hide latency and keep applications responsive in the face of intermittent connections, communication with RFID-tagged physical objects should happen asynchronously. Blocking communication will freeze the application as soon as one tag is unreachable.

Fault-tolerant communication. Treating communication failures as the rule instead of the exception allows applications to deal with temporary unavailability of the RFID-tagged physical objects and makes them resilient to failures. For example, read/write operations frequently fail due to hardware phenomena. Handling all these failures individually or each time they occur is not necessary and significantly complicates the development of mobile RFID-enabled applications.

Data consistency and security. Different mobile applications might concurrently read and – more importantly – write data to a number of tagged objects all within their proximity. This can lead to data races that have to be prevented. Similarly, in some scenarios data stored on RFID tags may not be read or modified by unauthorized users.

2.3.2 Loose Coupling

Since mobile RFID-enabled applications are distributed in a mobile ad hoc network, the different distributed components of such applications can be disconnected from each other at any point in time. To prevent the application from halting or even breaking, it should be possible to dynamically discover a replacement component for a disconnected component. This is only possible if distributed application components are not hard-wired in the application code by means of a fixed network address or location. As mentioned earlier, this type of decoupling is called *decoupling in space* [EFGK03]. On the other hand, if replacing a disconnected component is impossible (e.g. because of application-specific reasons, untransferable state...), the fact that communication is (temporarily) impossible should not block the entire application. Progress should still be possible where possible and communication among disconnected components should not be lost for when these components are reconnected. This type of decoupling is called *decoupling in time*.

Note that such loose coupling is advocated by the ambient-oriented programming paradigm. Below, we summarize these decoupling requirements into the following three properties.

Decoupling in Space

In a mobile ad hoc network, where no infrastructure is assumed, decoupling in space requires a form of *anonymous communication* where application components or services announce their presence to all reachable parties. It is then up to these parties to decide if interaction should occur. This should happen without relying on a third party, such as a naming server, event broker, etc. This boils down to broadcasting announcements on the one end and subscribing and reacting to these broadcasted announcements on

the other end. Programming abstractions for both these activities are needed to shield programmers from low-level implementation details.

Decoupling in Time

In a mobile ad hoc network, distributed application components or services should always run concurrently and not wait for each other to prevent one component or service to block the other when the network connection between them is broken. Decoupling in time means that the logical act of information sending is decoupled from the physical act of information transmission, allowing for the information to be saved and transmitted at a later point in time, when the connection between both parties is restored. This style of communication is called *asynchronous communication*. In the face of volatile connections, it makes sense to assume network disconnections are the rule rather than the exception to reduce the exceptional cases a programmer needs to address by providing fault tolerant communication. Programming models incorporating fault tolerant and asynchronous communication primitives are desirable in mobile RFID-enabled applications.

Arity Decoupling

As already mentioned above, broadcasting is a communication paradigm that should be supported to allow decoupling in space. Additionally, some types of hardware only support broadcasting, such as RFID technology. Hence, it should be available as a communication abstraction. On the other hand, employing broadcasting as the only communication abstraction can cause network congestion in some settings. Additionally, in some situations broadcasting simply is not the most appropriate abstraction. Hence, point-to-point communication is a communication paradigm that should coexist with broadcasting and the appropriate programming abstractions should be available to integrate both paradigms into a single useful application. In short, the arity of communication should be decoupled as well. Similarly to broadcasting, point-to-point communication should be possible without relying on a third party, and hence should be peer-to-peer.

2.3.3 Highly Event-driven Code

Although in classic software, applications have to be made responsive to external events such as user input or interaction with hardware, these applications are largely driven by a single control flow that is determined by the static structure of the program text. In mobile RFID-enabled applications, the set of tags that are reachable and with which the application is interacting is in constant flux. The result is that almost the entire application logic is event-driven. Hence, different code is triggered in response to different events by means of *event handlers* or *callbacks*, requiring careful management of how the application responds to events to keep the application responsive and correctly functioning. This introduces a number of software engineering issues.

Event Representation

First, events have to be represented as a data type such that they can be communicated to (separately developed) interested application components. Many event communication systems operate only on specific data types (e.g. event structs without

methods as opposed to full blown objects in an object-oriented language) which lack some of the expressive power typically conveyed by high level programming languages [VC03, Eug07]. Consequently, application data types that correspond to events must adhere to additional criteria to allow mapping application data to events and vice versa. To allow ambient-oriented programming with RFID-tagged objects, we require that these RFID-tagged objects can be represented as full-blown software objects.

Inversion of Control

Second, by adopting such an event-driven architecture, the application logic becomes scattered over different event handlers or callbacks which may be triggered independently [CM06]. The control of the application is no longer driven by an explicit control flow determined by the programmer, but by external events. This is a phenomenon known as *inversion of control* [HO06, PE02]. Control flow among event handlers has to be expressed implicitly through manipulation of shared state. E.g. unlike subsequent function calls, code triggered by different event handlers cannot use the runtime stack to make local variables visible to other executions (stack ripping [AHT+02]), such that these variables have to be made instance variables, global variables, etc. Finally, in more complex systems it is not always clear in which order different event handlers will be triggered, which can be critical in programming languages that allow side effects. Because the primary way of compositing separate event handlers is through side effects, interleavings and interactions become the developer's responsibility [MGB+09].

In short, making code reactive to events in a traditional imperative computational model is hard because of the lack of a *seamless integration of the event paradigm with a high level programming model* [VC03], rendering complex event-driven architectures hard to develop, understand and maintain [LC02, KR05, MRO10, Mye91].

Event-driven Control Flow Management

The unstructured execution of event-driven programs originates from the fact that the control flow that is automatically managed by the computational model of the programming language is largely abandoned in favor of one manually managed by the programmer [LB07]. As such, the flow of control is dictated by how event handlers are put in data structures, in which kind of data structures, how they are triggered, the level of concurrency, etc. This leads to a clumsy programming style where program state is explicitly encoded in globally visible variables to coordinate the unstructured execution of event handlers and can cause unexpected effects in imperative languages where side effects are caused by event handlers. In a distributed setting, this is even more complicated because the event handlers can be distributed over different distributed application components that can interact in unexpected ways. Hence, there is a need for *coordination* of reactive applications such that the coordination model [GC92] can take care of automatically managing the control flow. Furthermore, the coordination model should take the characteristics of applications distributed in a mobile ad hoc network into account.

2.3.4 Conclusion

In addition to the necessary abstractions over the bare RFID hardware mentioned earlier in section 2.3.1, suitable programming model for mobile RFID-enabled applications should provide:

- Decoupling in space and arity.
- Asynchronous (decoupling in time), fault-tolerant communication primitives.
- Rich representation of events.
- No inversion of control.
- Global and automatic control flow management.

Some of these properties should be combinable along different dimensions. For example, some existing systems provide fault-tolerant peer-to-peer communication, but no fault tolerance when one-to-many communication is employed. Other systems provide powerful event processing and communication capabilities, but require events to be represented as severely restricted data types.

When taking some of these properties together, we see that *distributed event-driven architectures* partially exhibit these properties. The traditional way of conceiving an event-driven system in a setting where the interacting entities change at runtime and communication must be time- and arity-decoupled is by adopting a *publish/subscribe architecture* [MC02b, MC02a, BJdL+04], where event *producers* publish events and event *consumers* subscribe and react to events, either using a topic-based or content-based subscription [CRW00, CNF01, EFGK03]. Not surprisingly, the highly dynamic nature of mobile ad hoc network applications caused preceding research to adopt similar architectures [KB02, MC02a, MPR01, Gri04].

In classic fixed-network publish/subscribe architectures, event producers and consumers have to register themselves to some centralized entity called an event broker that routes events between the different registered parties. Clearly, this assumes infrastructure in the form of a reliable server or network of servers that can act as the event broker. In the case of mobile ad hoc networks, registering event consumers to event producers should happen without relying on infrastructure such as event broker nodes, which means that either event advertisements or event subscriptions have to be broadcasted to reachable nodes [CRW01]. Hence, we postulate that mobile RFID-enabled applications should be structured as distributed event-driven architectures that do not rely on infrastructure by broadcasting event advertisements and/or subscriptions.

Publish/subscribe systems usually provide decoupled communication primitives, but do not provide programming tools for managing the control flow of an event-driven application. *Dataflow languages* [JHM04], on the other hand, do provide global and automatic control flow management. Other research areas, such as *sensor networks*, have to deal with similar problems and exhibit some of these properties as well.

When we look at RFID systems deployed today, we see that due to the massive deployment of RFID tags and the events they generate, traditional RFID middleware offer advanced event processing constructs.

For these reasons, in the next chapter, we survey RFID programming middleware, publish/subscribe systems, dataflow programming languages and sensor network programming technology with respect to the requirements listed above. The point of surveying these technologies is to extract how different programming technologies deal with the requirements listed in this section. Our aim is an integration of matching techniques such that all requirements are satisfied.

Concretely, we take as a starting point the ambient-oriented programming paradigm. It already offers the decoupling properties to deal with the hardware phenomena of mobile ad hoc networks. However, it does not take into account the inversion of control problem and offers no global and automatic control flow management.

Chapter 3

Related Work

In the previous chapter, we have observed that mobile RFID-enabled applications are highly event-driven applications running on top of mobile ad hoc networks. In section 2.3.4, we listed a number of programming model requirements to be able to concisely implement such applications. In this chapter, we survey various programming technologies that address one or more of these requirements with respect to the full list of requirements.

We start by looking at programming technology that specifically target applications that have to deal with RFID technology in section 3.1. These systems however, only target classic RFID-enabled applications instead of mobile RFID-enabled applications. Therefore, we subsequently broaden our scope to programming technology that specifically targets distributed event-driven software in section 3.2. However, these systems do not take into account the inversion of control problem. For this reason, in the subsequent section (section 3.3) we look into dataflow programming technology that allows to declaratively write event-driven code such that inversion of control does not occur. Finally, as mobile RFID-enabled applications are a specific type of sensor-driven software, in section 3.4 we look at how sensor network programming technology satisfies the requirements that we put forward. Finally, section 3.5 concludes this chapter by summarizing the techniques that are used in the surveyed work to satisfy our programming requirements. In that section, we point out which of these techniques we selected to integrate in the ambient-oriented programming paradigm to be able to concisely implement mobile RFID-enabled applications.

3.1 Survey of RFID Programming Technology

In this section we survey work on raising the abstraction level at which programmers interact with RFID hardware. We do not consider programming RFID hardware on the driver level by implementing low level protocols. We identify two categories of RFID programming technology. The first are backend-based RFID middleware implementing the Electronic Product Code (EPC) standard [EPC10b] tailored towards applications such as retail, product tracking, manufacturing, supply chain management, access control, etc. [LKKP10]. The second are decentralized programming abstractions targeting general mobile RFID-enabled applications.

3.1.1 Backend-Based Middleware

RFID middleware usually sits between the reader and software application which uses the EPC data. The main objective of RFID middleware is to collect large amounts of raw data coming from a heterogeneous RFID environment, filter them, compile them into a useable format and dispatch them to the application [AJAM09], while abstracting over concrete RFID hardware implementation details. It is most likely used in the cases when the data needs to be shared at more than one location at a time, such as in supply chain systems where many readers are distributed across factories, warehouses and distribution centers. Such middleware usually focus on connecting RFID technology to an enterprise backend and assume a stable network infrastructure, as is the case for the Sun Java System RFID Software [Ora]. Some middleware focus more on the filtering and aggregation of events produced by RFID readers and assume an underlying implementation of low-level RFID operations, as is the case for MIT Auto-ID Center's Savant middleware [OM02].

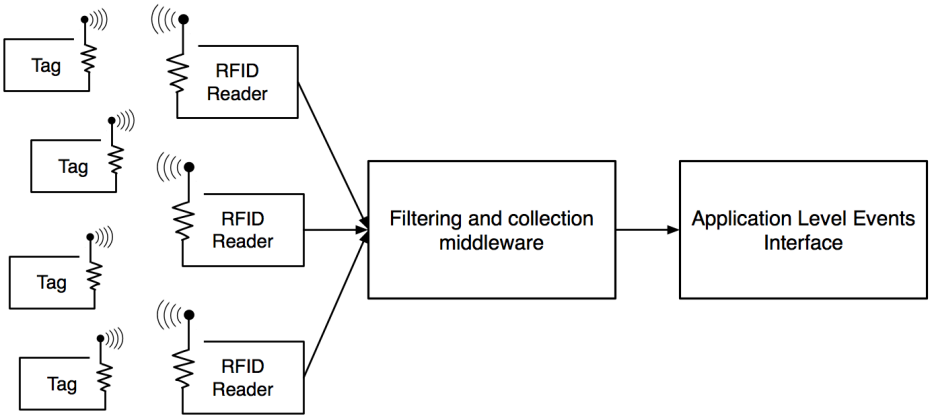

Figure 3.1: Typical architecture of EPC RFID middleware.

A highly simplified schematic of typical EPC middleware is shown in figure 3.1. Middleware implementing the full EPC standard and its protocols allows notifying registered application components of appearing and disappearing tags by means of the EPCGlobal Application Level Events interface (ALE) [EPC10a]. The ALE specification defines a SOAP message transport binding for the subscription communication channel and an XML and TCP/HTTP message transport binding for the notification channel. For this subscriptions and notification infrastructure to work, a reliable network is assumed, which is feasible in traditional RFID middleware applications where RFID readers are connected with applications over reliable connections. However, for mobile RFID-enabled applications we cannot rely on such infrastructure.

WinRFID [PSR+08] is an RFID middleware that is entirely based on the .NET Framework and Windows services, which are specified in XML. Services can read from and write data onto RFID tags through an object-oriented interface. The tag data is also specified in XML and is converted back and forth to a simplified and compressed format when written onto tag memory. Additionally, Windows services are autonomously running components that can run on mobile devices and are very loosely-coupled: they can interact using both synchronous and asynchronous messaging and publish/subscribe. The downside is that an approach based on web services

is more heavy-weight and does not offer the same performance as more light-weight approaches [RSMD04]. The main drawback of WinRFID however is that the devices and/or services have to be explicitly registered into a registry component, such that the services can contact this registry to interact with for example RFID readers that were a-priori registered. This means that there is no decentralized service discovery which makes WinRFID unsuitable for mobile RFID-enabled applications.

Fosstrak [FRL07] (formerly named Accada) is an open source RFID middleware that employs a virtual tag memory service (VTMS) that facilitates writing to a tag by shielding the application from the characteristics of RFID tag memory: limited memory size, different memory organizations, reduced write range. Hosts simply provide key-value pairs that should be written to a set of tags. The service then checks with the VTMS for the appropriate tag memory block and page to write to. If the write succeeds, the reader module acknowledges this to the host and stores a backup copy of the data in the virtual representation of the tag in the VTMS. If the host wants to access the memory of the tag while the tag is outside the range of any reader, the data can be made available via this virtual memory. If a write to the tag fails, the key-value pair is be stored in the VTMS and flagged as "open". The reader will retry the write command at a later point in time. The host can also indicate that the virtual memory of a tag can only be accessed once the tag is in the read range of the particular reader. Hence, Fosstrak allows addressing individual tags and storing application-specific data on these tags using asynchronous and fault-tolerant communication. For this however, the application data has to be converted back and forth to simple key-value pairs of which the values can only be of a restricted set of simple data types.

AspireRfid [KLS+08] partially caters to mobile RFID-enabled applications, by offering support for mobile RFID-enabled devices (such as smartphones) and other types of sensors (e.g. temperature). Mobile RFID-enabled devices communicate with backend infrastructure that runs the AspireRfid middleware by means of wireless networking technologies. Applications running on mobile devices have to explicitly send an intermediary representation of the scanned objects to this infrastructure which will wrap it to comply with and be processed using EPC protocols. Because there the middleware needs a connection with the backend, it is impossible to run the middleware as a stand alone application on the mobile device itself, which makes AspireRfid unsuitable for mobile RFID-enabled applications.

3.1.2 Decentralized Middleware

At the time of writing, the only decentralized RFID programming technology is one based on tuple spaces [MQZ06]. The concept of tuple spaces was originally introduced in the coordination language Linda [Gel85]. In the tuple space model, distributed processes communicate by inserting and reading and/or removing tuples of primitive data types to and from a shared tuple space, which acts as a globally shared memory. Because tuples are anonymous, they are extracted by means of pattern matching on their content. Tuple space communication is decoupled in time because processes can insert and retract tuples independently and without being connected at the same point in time. It is decoupled in space because the publisher of a tuple does not necessarily specify, or even know, which process will extract the tuple.

In [MQZ06] this concept is applied to RFID technology by dynamically constructing a distributed tuple space of all tuples stored on the tags that are in reading range. Applications running on mobile devices or robots equipped with an RFID reader can control their own RFID reader to access the environmental memory via traditional tu-

ple space operations. These operations provide applications the capability of storing new tuples in the environment and of reading/extracting tuples in an associative way. Blocking read/extract operations and asynchronous notification mechanisms (to notify agents when a matching tuple is detected in one or more of the tags in range) enable applications on different devices to coordinate and synchronize their activities with each other through the environmental memory.

From the viewpoint of the application, the perception of the environment is that of a normal tuple space that contains all tuples stored on the RFID tags in rage at that point in time. There is no such concept as a single global tuple space or as a multiplicity of localized tuple spaces. Tuples are spread everywhere in the physical space, without being at all pre-organized in logical containers and rather defining a sort of "continuum" of tuples. The contents of the tuple space is subjective, depending on the current position of the device hosting the application and on the range of its RFID reader. The same application perceives different information in different physical positions, while two applications in the same position may perceive different information depending on the reading range.

The middleware offers no way to control on which specific tag the inserted tuples will be stored, or even if the write operation will succeed when no tag is in range. Hence, with this approach RFID tags cannot represent physical objects as there is no way address one specific RFID tag and write operations are not fault tolerant.

3.1.3 Evaluation

We summarize our survey on RFID programming technology by means of the desired features for implementing mobile RFID-enabled applications listed earlier in section 2.3.1. Table 3.1 gives an overview of the systems discussed above and indicates to which extent they meet these requirements.

The main conclusion we draw from this overview is that existing RFID middleware simply implementing EPC standards and protocols requires a central backend infrastructure to control the RFID readers. Therefore, these systems are not directly usable for developing mobile RFID-enabled applications that are autonomous and use mobile ad hoc network technology to communicate. Furthermore, traditional RFID middleware is typically rather heavy-weight and not tailored to mobile devices [LKKP10]. AspireRfid does allow mobile RFID readers and offers support for other types of sensor devices, but still uses a backend architecture to control these devices and to process and communicate their events to the application level. These restrictions stem from the very specific application domains of these systems: they specifically target classic RFID applications where RFID tags are used for identifying and tracking objects. They contact a centralized database storing the information associated with each RFID tag [PS11, DMS07, MTCS07].

The only other approach (except from ad hoc implementations directly on top of the hardware) at the time of writing is a tuple space-based approach. Although tuple spaces are attractive for mobile ad hoc network applications [MPR01, MZ04], the concrete RFID-enabled implementation discussed earlier does not allow designating a single RFID-tagged object and only provides asynchronous and fault-tolerant communication with RFID-tagged objects for read operations.

A final interesting insight is that all investigated systems lead to event-driven architectures where client applications subscribe to events that represent changes in the physical environment. Hence, they achieve a causal connection with the physical

	WinRFID	Fosstrak	AspireRfid	Tuple Spaces
Addressing physical objects	Yes	Yes	Yes	No
Storing application data on RFID tags	Yes	Only key-value pairs	Yes	Only key-value pairs
Reactivity to (dis)appearing objects	Yes	Yes	Yes	Yes
Asynchronous communication	Yes	Yes	No	Only read operations
Fault-tolerant communication	Yes	Yes	No	Only read operations
Data consistency and security	Yes	No	No	No
Support for mobile ad hoc networks	No	No	No	Yes
Additional remarks	Loosely-coupled, but heavy-weight web services		Support for mobile readers and other sensors	

Table 3.1: Survey of RFID programming technology.

world, but do not specifically fully cater to mobile RFID-enabled applications. Hence, in the next section we survey distributed event driven architectures.

3.2 Survey of Publish/Subscribe Systems

In section 2.3.4, we already mentioned that because of the distributed and inherently event-driven nature of mobile RFID-enabled applications, they accommodate distributed event-driven architectures. Distributed event-driven architectures are typically implemented by publish/subscribe technology. This survey does not aim to be an exhaustive survey of all work done in this field, but tries to evaluate support for these levels in the light of mobile RFID-enabled applications by taking into account the requirements formulated earlier in section 2.3.4.

The publish/subscribe paradigm is attractive for mobile RFID-enabled applications because it inherently provides support for loosely coupled interaction between event producers and consumers [HGM04, BJdL+04]. Event subscriptions can happen without having producers to need to know consumers (and how many of them there are) a-priori and vice versa.

Different strategies for specifying event subscriptions are possible [EFGK03]. In topic-based publish/subscribe, events are categorized in topics or event types (which can be hierarchical) and subscribing happens by simply matching on the topic. In content-based publish/subscribe, matching happens on the actual content of the event data type. Of course, this places some restrictions on the structure of this data type

such that subscribing can happen expressively and matching can be performed efficiently. Hence, content-based publish/subscribe systems usually restrict events to tuples or XML documents with a fixed structure [SFR05]. Sometimes event subscriptions are extended with domain-specific information, such as geographical distance between event producers and consumers [MC02a] in traffic control systems.

Different subscription strategies also have their impact on the routing of events, i.e. delivering the events from producers to consumers [MÖ1]. A centralized solution relying on a single event broker is neither scalable nor fault-tolerant. The alternative is to use a set of distributed brokers in which each broker has a set of local clients and is additionally communicating with some of the other brokers. In that case, events are propagated from producers to consumers along a path of interconnected brokers. Such an architecture is depicted in figure 3.2. Most approaches either flood events to the

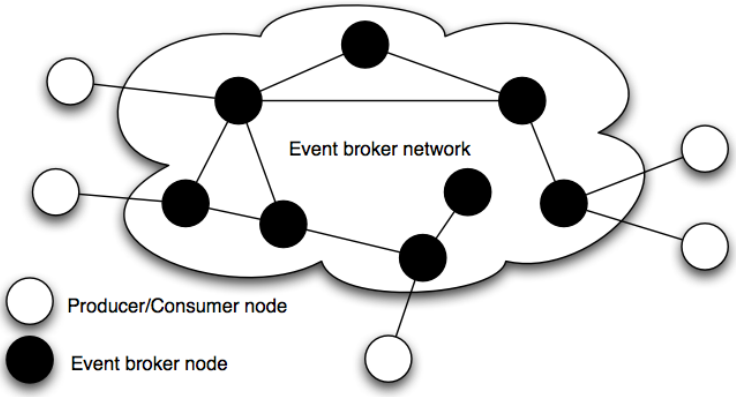

Figure 3.2: Distributed publish/subscribe architecture.

broker network or, if filtering is performed at intermediate brokers, it is assumed that each broker has global knowledge about all active subscriptions (e.g. by maintaining routing tables which map event subscriptions to brokers to which events should be forwarded). This knowledge is constructed by flooding subscriptions filters into the broker network. It enables each broker to decide to which brokers it must forward incoming notifications. Therefore, subscriptions are forwarded to ensure the delivery of all matching notifications.

As an alternative to the naive solution to event routing mentioned above, **SIENA** (Scalable Internet Event Notification Architectures) [CRW00, CRW01] reduces the amount of knowledge needed by a broker to make event forwarding decisions by applying selective filter forwarding (i.e. subscription and advertisement forwarding). SIENA's routing algorithms are based on covering relations among filters. For example, a subscription does not need to be forwarded by a broker to another broker if it has already forwarded a subscription that covers the former because in that case already all matching events are delivered. IBM's Gryphon [BCM+99] uses similar mechanisms. These filter expressions are not only used for event subscriptions, but also for event advertisements. The motivation for advertisements is to inform the event notification service about which kind of events will be generated by which event producers, so that it can best direct the propagation of subscriptions. The idea is, that while a subscription defines the set of interesting events for an interested party, an advertisement defines the set of events potentially generated by a producer. Therefore, the advertise-

ment is relevant to the subscription only if these two sets of events have a nonempty intersection. The use of advertisements is not necessary, but allows for the routing optimizations described above. Additionally, SIENA allows a combination of multiple events to be matched using patterns, which are multiple filters applied in sequence on event attributes of the corresponding events. For the ordering of these events, SIENA provides best-effort semantics.

Although SIENA relaxes the requirement of a fixed, reliable network of event brokers, it still assumes some form of centrally administered network infrastructure to form a broker network, which cannot be assumed in spontaneously forming mobile ad hoc networks [SGP04]. Instead, SIENA focuses on providing efficient and scalable notification routing over a wide-area network.

JEDI (Java Event-based Distributed Infrastructure) [CNF01] is a publish/subscribe system that supports mobile clients. For this, JEDI defines two functions – `moveOut` and `moveIn` – that a client can use to explicitly detach from the event broker network, and to reconnect to it, possibly at a different location (and hence to a different node of the broker network). When a client is disconnected, the event broker network stores all the events in which the client is interested. The client can then reconnect to the same node of the broker network or to another node of the network on a different location by calling `moveIn`. The effect of the `moveIn` operation is to retrieve all events buffered by the broker network in which the client was interested. JEDI still relies on a centrally administered infrastructure to host the broker network, since it must have a global knowledge of all the events that are generated and of all the subscriptions that are issued, but offers fault-tolerant event delivery to weakly connected mobile clients. However, the programmer has to manually call the `moveOut` and `moveIn` operations when relocation of the client is expected. In intermittently connected systems such as mobile RFID-enabled applications, this leads to complicated exception handling code as disconnections can be unpredictable and can be the rule rather than the exception. In [CRW01] this process is automated by letting the middleware monitor the connection status of the client and as soon as a disconnection is observed queue all messages sent to that client in the broker network. When the client reconnects, the queued messages are automatically delivered. This approach offers support for intermittently connected clients.

JEDI's computational model is based on the notion of active objects. An active object is an autonomous computational unit performing an application-specific task. Each active object has its own thread of control and interacts with other active objects by explicitly producing and consuming events.

Location-based Publish/Subscribe (LPS) [EGH05] is a content-based publish/-subscribe middleware designed for mobile computing applications. In order to scope interactions between devices, event propagation and reception is bounded by physical space: a producer defines a publication range and a consumer defines a subscription range. Both are independent of the devices communication range. An event is only propagated to the consumer if the publication range of the producer and the subscription range of the consumer physically overlap.

Events are represented as tuples of primitive values and are offered for distribution to all interested subscribers located or entering the publication space before the event is unpublished. A persistent event can be unpublished either by the publisher or by the service after a determined period known as its time-to-live. The publication space is centered around the publisher, implying that the publication space moves along with the publisher between the time the event is published and the time the event is unpublished. Content-based subscription happens by pattern matching on the event tuple's attributes.

At the time of writing, the implementation of LPS relies on a web service accessed via HTTP by clients. This web service plays the role of communication backbone and performs the matching between published events and subscribers. This centralized architecture where permanent connections are assumed makes the current implementation not suitable for mobile RFID-enabled applications.

STEAM (Scalable Timed Events And Mobility) [MC02a] is an event-based middleware service that has been designed for mobile ad hoc networks, more specifically traffic management systems. In this context, STEAM has been used as the underlying publish/subscribe middleware for the CORTEX project [BC04]. In STEAM it is assumed that the closer event consumers are located to an event producer, the more likely they are to be interested in the events propagated by that producer. This implies that events are valid within a certain proximity of a producer. Propagating events within a certain area surrounding producers limits forwarding of event messages, and therefore reduces the usage of communication and computation resources, but also renders system-wide services virtually impossible to implement.

The notion of proximity involves both geographical and functional aspects, i.e. the geographical area and the type of events that are communicated. An application component must be located in the geographical area corresponding to the group and additionally be interested in the event type to be subscribed. Because of this mechanism and by broadcasting event messages within the area, STEAM allows consumers to subscribe to particular nearby event types without relying on fixed event brokers and exploits the reliable delivery of messages among proximate application components to provide end-to-end guarantees when delivering events. This is possible by wirelessly broadcasting events to reachable nodes.

Additionally, STEAM event consumers can filter events based on their content. Filter expressions may contain equality, magnitude and range operators as well as ordering relations. In contrast to the subscription matching on the event type and proximity with the consumer, these filter expressions are applied on the consumer side.

EMMA (Epidemic Messaging Middleware for Ad hoc networks) [MMH05] is a publish/subscribe middleware based on the Java Message Service and specifically targeted towards mobile ad hoc networks. Because the middleware represents events as full-blown Java objects, it does not implement a dedicated subscription language over some restricted data type to allow content-based subscription and hence only allows topic-based subscriptions.

EMMA's communication model is based on the concept of message queues that are used to enable asynchronous communication between the producer of an event and possible different consumers. Message queues periodically advertise their topic to reachable hosts by broadcasting advertisements. It is important to note that, at the middleware level, it is logically irrelevant whether or not the network layer implements some form of ad hoc routing (though considerably more efficient if it does); the middleware only considers information about which nodes are actively reachable at any point in time. The hosts that receive advertisement messages to which they are subscribed store a reference to the advertised message queues. Each reference is characterized by a lease. A lease represents the time of validity of a particular entry. If a lease is not renewed, it expires. Consequently, the reference is deleted and a new one has to be discovered. In other words, the host assumes that the queue will be unreachable from that point of time.

Additionally, in order to deliver a message to a host that is not currently in reach, EMMA uses an asynchronous epidemic routing protocol. A message that needs to be sent is replicated on each host in reach. In this way, copies of the messages are quickly

spread through connected networks, like an infection. If a host becomes connected to another partition of interconnected mobile nodes during its movement, the message spreads through this collection of hosts. Within epidemic routing, each host maintains a buffer containing the messages that it has created and the replicas of the messages generated by the other hosts.

GREEN [SBC05] is a publish/subscribe system featuring reconfiguration capabilities on the architectural level by adopting a component architecture, where each component is responsible for a specific functionality (subscription matching, event routing or event broker network management), allowing dynamic plugging of components inside the architecture. This allows the system to adapt to various deployment contexts (from fixed to mobile networks) simply by implementing and plugging new specific components. Hence, the event subscription component can take many different forms, depending on how much expressiveness is needed. One could implement a component that simply uses topic-based subscriptions, content-based matching on tuples, XML-documents or even full-blown objects in an object-oriented programming language. Additionally, this flexible approach allows to plug in a component that can generate composite events. Additionally, the notion of physical proximity can be taken into account in event subscription, similar to the STEAM middleware discussed earlier.

Event propagation is also a component of which the behavior is entirely open. In fixed networks, a component that sets up a stable event broker network can be used. In mobile ad hoc networks on the other hand, components implementing different strategies can be used depending on the mobility of nodes, such as event broadcasting to nearby nodes.

one.world [Gri04, GDH+01] can be regarded more as an operating system for pervasive computing rather than as middleware, providing a common execution platform for pervasive computing applications. one.world application components interact through asynchronous event notifications only. All data types which are exchanged among distributed application components are represented as tuples. Services export event handlers under symbolic descriptors, i.e. tuples, Clients send events by specifying the symbolic receiver, acting as a both topic-based and content-based subscription mechanism.

Event routing can happen by means of different mechanisms. The first way in which such a mechanism can vary is the binding time. Early binding first binds a producer and consumer and then uses point-to-point communications for delivering events. It is useful when an application needs to repeatedly send events to the same consumer or when services can be expected to remain in the same location. Late binding binds producers to consumers while routing the event. While it introduces a performance overhead for every sent event, late binding also is the most responsive and thus most reliable form of communication in a highly dynamic environment. A second point of variation is the arity of the communication: events can be either multicasted to all matching subscribers or to only a single subscribed client. one.world uses leasing to dynamically replace the binding with distributed components which become unreachable.

Event subscription relies on a centralized server to provide its functionality. This server is automatically elected from all nodes running one.world on the local network, with elections favoring nodes with ample resources and long uptimes, and periodically announces its presence by broadcasting. Although this mechanism works for mobile ad hoc networks, it assumes that all devices are capable enough to act as such a server.

A **Distributed Asynchronous Collection** (DAC) [EGS00] extends the notion of a traditional object-oriented collection to a distributed context. A DAC is shared be-

tween multiple distributed objects. Importantly, a DAC is not a centralized collection accessible by distributed objects but rather an inherently distributed collection. Objects can add other objects to the collection (which is equivalent to publishing an event) and can register a callback to be invoked asynchronously whenever objects are added to the collection (which is equivalent to subscribing to events). This happens by specifying a topic or using content-based subscription techniques over full-blown event objects without breaking encapsulation by relying on reflection [KR91].

While traditional object-oriented collections differ in terms of whether or not they accept duplicates (e.g. bags versus sets) or the order in which they store their elements (e.g. stacks versus queues), DACs differ in terms of delivery guarantees (e.g. at-least-once versus at-most-once delivery) and delivery order (e.g. FIFO versus total order) of events. However, DACs have not been designed specifically for mobile ad hoc networks: DACs do not support time decoupling: messages are considered volatile and are dropped once delivered to all connected subscribers.

EventJava [EJ09] is strictly speaking not a publish/subscribe system, but an extension to Java for distributed event-based programming. It extends Java with asynchronous event methods. The fact that events are directly signaled by invoking event methods means that events are represented as argument lists of these method invocations, which can be lists of arbitrary Java objects (matching the signature of the event method).

Event methods can define a predicate over the event that is passed that makes sure that the event method body is only effectively executed if the event matches the criteria. Complex events can be created by listing different event method signatures into a single event method declaration. If all the event methods are invoked that are in the complex event method declaration, the complex event method will be asynchronously invoked. Just like with basic event methods, complex event methods can have a predicate that is used to filter events. In this predicate, the individual events constituting the complex event can be accessed by the corresponding individual event method names.

EventJava supports event correlation over event streams by annotating event methods with an "`[n]`" array-like syntax (denoting a window of the stream of n events long). In the event predicate and method body, different individual events in the stream can be accessed using the same array-like syntax.

EventJava supports full arity decoupling by allowing directed communication by (asynchronously) invoking an event method on a specific instance, and by allowing broadcasting by invoking an event method on a class. In the latter case, the event methods of all instances of that class (and its subclasses) will be invoked. For achieving this in a distributed context, it relies on the JGroups[1] group communication framework. It offers time- and space-decoupled communication since it uses asynchronous fault-tolerant communication and allows broadcasting an event message to all instances of a class, including late-joining instances.

Since event methods are asynchronously invoked methods that do not return a result, they are in essence callbacks that have to be manually coordinated by the programmer.

3.2.1 Evaluation

Let us now evaluate the surveyed systems with respect to the requirements put forward in section 2.3.4 (namely arity and space-decoupled communication primitives, asyn-

[1] http://www.jgroups.org/

chronous and fault-tolerant communication (time decoupling), a rich representation of events and no inversion of control). The results of our evaluation are shown together in table 3.2. Since all of these systems only deal with the subscription and publication of events and rely on plain callbacks for processing events, we omit the no inversion of control and global control flow management requirements. The properties that are useful for mobile RFID-enabled applications are shown in italic font.

	Space decoupling	**Arity decoupling**	**Time decoupling**	**Event representation**	**Infrastructureless**
SIENA	*Yes*	Broadcast only	No	Key-value pairs	No
JEDI	*Yes*	Broadcast only	*Yes*	String tuples	No
LPS	*Yes*	Broadcast only	*Yes*	Tuples	No
STEAM	*Yes*	Broadcast only	*Yes*	Tuples	*Yes*
EMMA	*Yes*	*Yes*	*Yes*	*Objects*	*Yes*
GREEN	Pluggable	Pluggable	Pluggable	Pluggable	Pluggable
one.world	*Yes*	*Yes*	*Yes*	Tuples	No
DACs	*Yes*	*Yes*	No	*Objects*	*Yes*
EventJava	*Yes*	*Yes*	*Yes*	*Objects*	*Yes*

Table 3.2: Survey of distributed publish/subscribe systems.

Space Decoupling Since space decoupling or anonymous communication is one of the main motivations for publish/subscribe, all surveyed systems offer space-decoupled communication.

Arity Decoupling All systems offer arity-decoupled communication in the sense that event consumers do not have to be specified when signaling an event. However, not all systems support directed communication, which means that the programmer must rely on the event subscription mechanism to filter out unwanted events. Some systems such as EMMA and one.world allow to set up a message queue or channel between two communication partners for directed communication. These communication partners discover each other using the classic publish/subscribe mechanism.

Time Decoupling In the systems discussed above, fault-tolerance is usually provided by buffering event messages such that they are not lost in the face of failures. Since these messages cannot be stored in an event broker network, the communicating parties are responsible themselves for storing these messages and forwarding when connections are restored, or when new subscribers connect. An event broken network would introduce the need for infrastructure which in itself is prone to failures.

Event Representation Most of the systems discussed above represent events as tuples or key-value pair lists of primitive values, in most cases because content-based event subscription languages or techniques require this. This phenomenon is called

the *object-event impedance mismatch* by Pietzuch and Bacon [PB02] and by Van Cutsem in his PhD thesis [Van08], drawing a parallel with the object-relational impedance mismatch [CD96]:

> "The object-relational impedance mismatch is caused by the fundamental differences between modeling data as objects and modeling data as tuples which are part of relations. For example, objects encapsulate their state, enabling operations to be polymorphic. Tuples expose state, enabling efficient and expressive filtering, querying and aggregation of data. Objects refer to one another via references, while tuples are associated with one another via foreign keys. Identity is fundamental to objects, while tuples lack any inherent form of identity [...]."

This notion of identity is important for our work on ambient-oriented programming for mobile RFID-enabled applications, where software objects should always be causally connected to their physical counterparts through their identity.

The work on Distributed Asynchronous Collections and type-based publish/subscribe [Eug07] (as employed in EventJava) has shown that by relying on an object-oriented programming language that supports reflection, content-based publish/subscribe is also feasible when events are represented by arbitrary objects. Carzaniga et al. [CRW01] motivate against representing events as arbitrary objects by arguing that in that case events have an associated type (represented by their class), which requires a global authority for managing and verifying the type space. However, as we will show in chapter 4 in our discussion about the programming language AmbientTalk [VMG+07] in section 4.1, there is a way out by using a prototype-based programming language [Lie86] (such as AmbientTalk), where objects have no associated class, but are self-contained and can be freely exchanged[2].

One of the advantages of representing events as full-blown objects is that by exploiting the encapsulation and polymorphism they offer it is possible to reuse or change different data representations for events without breaking dependent code. For example, removing a field in a tuple-based or key-value-based event representation can cause all application components working with these tuples to crash, if they are not adapted accordingly. If these events are represented as objects, the deleted field could be computed by a method implemented by the event object. This way, representing events as objects further promotes the loose coupling between event producers and consumers, but on the event representation level instead of on the architectural level, by programming against an interface shielding the data from direct access. Furthermore, the notion of unique *references* to other objects (whether they are hosted locally or remotely) allows some patterns to be easily expressed, while without this notion of identity, the programmer must manually generate unique identifiers and use them to refer to other application data (such as a previously generated event).

Infrastructureless Operation The publish/subscribe systems discussed above can be roughly classified as follows: systems having reliable access to local networks or wide-area networks in which an event broker network can be deployed, and systems in which event dissemination happens entirely ad hoc using wireless communication. The second category employs some form of broadcasting: subscriptions (EMMA), events (STEAM) or both must be broadcasted to all nearby parties to make them discover each

[2] Except their enclosing lexical scope, an issue that will be discussed as well in the next chapter of this dissertation.

other. This category is the one targeting mobile ad hoc networks and are interesting for mobile RFID-enabled applications. This does not mean that broadcasting is the only way of communicating, e.g. EMMA uses epidemic messaging to forward events, which is an example of a decentralized communication strategy. Figure 3.3 shows a publish/subscribe architecture for mobile ad hoc networks, contrasting our earlier example of an architecture based on event brokers in figure 3.2.

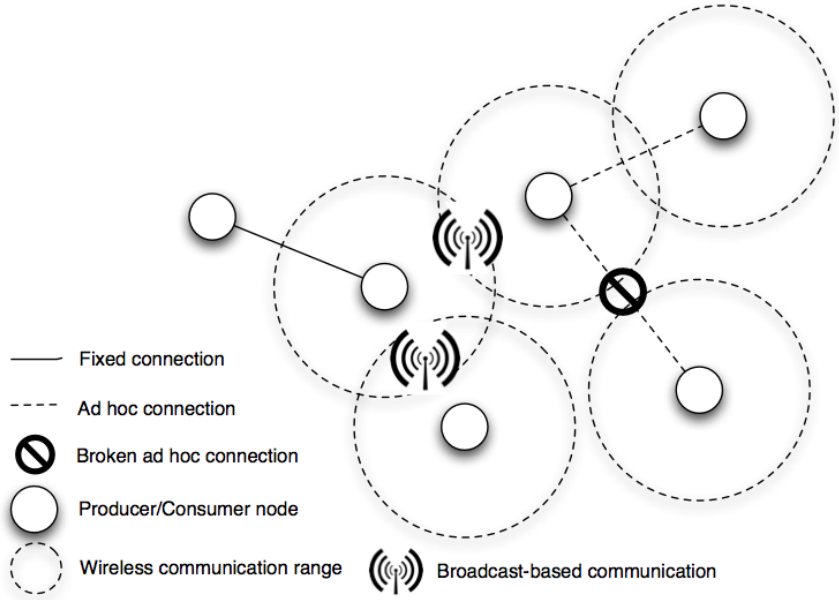

Figure 3.3: Publish/subscribe architecture for mobile ad hoc networks.

Control Flow Management All of the surveyed systems require reacting to events by means of explicit callbacks. This causes the problem of inversion of control. With respect to the systems discussed above, it is observed by Grimm et al. and formulated as follows in their experience report with one.world [GDH+01]:

> "Several event handlers in our applications need to process many different types of events or perform different actions for the same type of event depending on the event's closure. Their implementation requires large if-then-else blocks that use `instanceof` tests to dispatch on the type of event or more general tests to dispatch on the value of the closure. The result is that these event handlers are not very modular and are relatively hard to understand, modify, or extend. This suggests the need for better programming language support to structure event handlers. [...] While we still believe that asynchronous events are an appropriate abstraction for pervasive computing, our experience with event-based programming also suggests that [...] asynchronous events are as hard to program as threads. Just like threads, asynchronous events can result in complex interactions between components."

3.3 Survey of Dataflow Programming Technology

Over the years, different styles of dataflow programming have been developed [JHM04]. The original motivation for research into dataflow programming was the exploitation of massive parallelism. Therefore, much work was done to develop ways to program parallel processors as an alternative to the Von Neumann architecture. Dataflow architectures avoided some of the bottlenecks in Von Neumann architectures by using only local memory and by *executing instructions as soon as their operands become available*. This maps well onto event-driven architectures because the notion of an event can be represented as an operand involved into an instruction becoming available, allowing reacting to such an event by simply executing the instruction.

The name dataflow comes from the conceptual notion that a program in a dataflow computer is a directed graph and that data flows between instructions, along its arcs. However, it was found that parallelism used in dataflow architectures operated at too fine a grain and that better performance could be obtained through some hybrid with classic Von Neumann architectures. In such *coarse-grained* dataflow programming systems dataflow is used at a higher level and the individual instructions are still developed using a conventional programming language and executed by a conventional execution model, such as threads [Jag95]. In fact, over the years the motivation for the dataflow paradigm shifted from performance reasons to software engineering. In this section we will focus on the latter.

The dataflow programming model can be informally described as follows:

- Dataflow programs consist of *dataflow operators* that take a number of input values and return some output values. These dataflow operators are represented as nodes in a *dataflow graph*.

- Dataflow operators communicate with each other over *dataflow edges* (sometimes also called *channels*). These edges represent data dependencies and always flow from the outputs of dataflow operators to the inputs of other dataflow operators.

- When a dataflow operator is fired depends on the concrete execution model used. Some languages only fire dataflow operators once as soon as *all* their input values have received a value. Other languages repeatedly fire dataflow operators as soon as *one* of their input values received a *new* value. Upon firing, the operator is applied to the latest values on its incoming edges (that can be seen as actual parameters) and in response can possibly place new output values on (some of) its output edges. These flow to other dependent dataflow operators.

This allows that different dataflow operators in the dataflow graph can execute in parallel as long as their data dependencies are satisfied. For example a number of operators in a pipeline execute in parallel when the first operator is fed a stream of data. In such a pipeline the first operator is being applied to new data from the stream while operators later in the sequence are being applied to data already processed by earlier dataflow operators in the pipeline.

In this section, we survey dataflow programming technology. We classify dataflow programming into two approaches. The first category constructs a dataflow graph *implicitly* to coordinate the event driven execution of a reactive program. The second category presents dataflow graphs as an explicit abstraction to the programmer.

3.3.1 Functional Reactive Programming

Functional reactive programming is a style of event-driven dataflow programming where events are represented as *time-varying values*. Hence, instead of modeling events as discrete notifications on which the programmer has to explicitly react, events are modeled as state changes of some native data type of the concrete programming model. Low-level events are handled implicitly and update such time-varying values. In many cases, this allows conceiving event-driven applications without resorting to a complex network of callbacks. Additionally, because time-varying values are native data types of the programming model, there is usually explicit support to combine them and integrate them into applications. Dataflow graphs are created *implicitly* by the language interpreter, compiler or virtual machine and are used to track dependencies between reactive expressions. In most cases, this is used to decide when to *recompute* reactive expressions in response to events.

Lucid [AW77] is a functional programming language originally designed for easing the reasoning about and verification of programs by eliminating procedural features such as assignment and goto statements. Procedural features were eliminated by expressing iterative, statement-based procedural programs with declarative expressions. For example a procedural counter in which the value of a variable is updated every iteration of a loop `(i := i + 1)` can be declaratively expressed with the following two definitions:

```
first i = 1
  next i = i + 1
```

Declarative iteration opened up the possibility that operators such as `first` and `next` could be used to describe the dynamic nature of a program and the contextual values of variables. The `next` operator refers to the value of an expression in the context of the next unit of time and `first` refers to the value of an expression in its first ever instance.

Different variants of Lucid have been developed over the years [PMD08], on of which is pLucid [FW87], the first dataflow variant of Lucid. In pLucid, all variables and expressions denote *streams* (i.e. time-varying values), and functions denote *filters* (stream-to-stream transformations). The values of a stream in pLucid are primitive values and arbitrary lists of values (similar to LISP). pLucid is a first-order language: the programmer cannot use or define higher order functions, that is, functions which accept other functions as arguments and/or return functions as results. Without higher order functions the lambda notation (for closures) is of limited use and is therefore not supported by pLucid[3]. In addition to Lucid's iteration primitives, pLucid also provides a `fby` (followed by) primitive, which allows the creation of dataflow programs working on streams. The example given above can be rewritten using `fby` as follows:

```
n = 1 fby (n + 1)
```

pLucid uses a global clock that is used to tag values in a stream according to time, but it does not correspond to physical time, rather to the next batch of time-varying values that are being processed. Furthermore, to achieve reasonable performance, a special centralized datastructure called a "warehouse" to cache calculated values and accelerate future computations is needed. For this reason, pLucid has no distribution support.

Granular Lucid [JDA97] is a coordination language that specifies the parallel structure of the application and imperative functions - written in C - to perform the calculations. This high-level approach was designed to take advantage of coarse-grain

[3] Although higher order functions can be integrated in a dataflow language, as demonstrated in [CGHP04].

data parallelism present in many applications, for example, matrix multiplication, ray-tracing, video encoding, CT-scan reconstruction, etc. The dataflow evaluation mechanism was adapted to distribute these tasks across a shared-memory multiprocessor or a network of distributed workstations, but a centralized architecture is still assumed.

Lucian [OM08] is a variant of Lucid that integrates Lucid and the dataflow paradigm with object-oriented programming. Lucian embeds dataflow into object-orientation via the concept of declarative intensional objects. These objects encapsulate state that varies over time. For example, if one takes an intensional object and treats it as a stream of object snapshots, the first value of the stream is the snapshot of the object just after instantiation. The next value of the stream is the snapshot of the object in the next time step, with possible transformations applied, and so on. Selecting a field of an object again returns a stream corresponding to the varying value of the field over time. Setting a field causes the field to refer to another stream. A limitation of Lucian is that calling methods disrupts this semantics. They return a snapshot return value instead of a stream that is generated by re-invoking the method over time. Another drawback is that classes have to be defined in a classic object-oriented language. They can be defined inline in Lucian code, but this intermingles both syntaxes. There is no distributed implementation of Lucian.

Finally, there is a distributed, decentralized version of Lucid Synchrone [DGP08]. It allows to specify from the program text on which distributed nodes dataflow values are hosted and which dataflow variables can be communicated across nodes. The compiler then takes care of actually distributing the dataflow program. However, this language focuses on embedded systems of which the connections are reliable: the distributed program cannot reconfigure itself from its static structure determined at compile time.

Lustre [HCRP91] is a functional reactive programming language designed for real time applications where the notion of time is critical and originated as a simplified version of Lucid. It is based on the assumption that a Lustre program can react to an external event before any further event occurs, and that this property can be statically verified (hence, reactions should happen fast enough to give the illusion that they happen instantaneously). This is called the *synchrony hypothesis* (therefore Lustre belongs to the family of *synchronous* dataflow languages).

In Lustre, any variable and expression denotes a flow, which is similar to the event stream concept, but differs by internally having a clock that represents the pace at which the stream progresses and at which point in time it has a value. Clocks can be seen as boolean functions that are applied on the stream at every time step and return whether the stream has a value at that point in time or not. The basic clock is the clock that sets the minimal "grain" of time within which a program cannot discriminate external events, and which corresponds to its response time. It can be determined by a real physical clock, and as such obtain statically verifiable real time guarantees.

In Lustre, the following expression returns a new flow, of which the latest value is `0`, until the latest value of the flow `X` becomes greater than `0`, then a new latest value becomes the latest value of the flow `Y` incremented with `1`.

```
if X > 0 then Y + 1 else 0
```

To generate the new flow, expression have to be *reevaluated* when the clocks of the flows on which they depend advance and a new value is generated. Flows can be manipulated and combined using a number of primitive operators. For example, the `pre` operator takes a flow as argument and returns a new flow that always contains the previous value of the original flow (i.e. to remember old values).

Being a synchronous language, Lustre is based on the synchrony hypothesis that states that the reaction time of a particular component, and the communication time between components, are also zero. Synchronous languages are intended to be compiled into centralized sequential code. The parallel composition and the communication mechanism are introduced only for description purposes, at the level of the language; they are compiled into something sequential, hence they do not imply explicit parallelism and communication at execution time. Hence, Lustre does not support distributed programming.

FrTime [CK06] (pronounced "father time") is an extension of the general-purpose programming language Scheme [Dyb09] designed for writing interactive applications. Inspired by functional reactive programming, the language embeds dataflow within a call-by-value functional language. FrTime is a collection of syntactic abstractions and value definitions implemented in Scheme. Executing a FrTime program means running the Scheme evaluator in an environment containing the FrTime definitions. These definitions make executing the program build a graph of its dataflow dependencies, to which evaluated expressions that depend on time-varying values are registered (as nodes). A dataflow engine subsequently reacts to events and propagates changes through this graph of dataflow depedencies (i.e. the edges), triggering reevaluation of dependent expressions in such a way that expressions re-execute in response to an event in expected ways (i.e. in the order they were textually specified by the programmer) and without triggering superfluous re-evaluation. Pure Scheme programs are also FrTime programs with the same meaning they have in Scheme and may be incorporated into FrTime programs without modification.

FrTime considers two types of time-varying values: *behaviors* and *event sources*. Behaviors are values that vary continuously while event sources represent sequences of discrete occurrences, similar to event streams. Unlike with behaviors, primitive procedures cannot be applied to event sources. FrTime instead provides a collection of event processing combinators that are analogous to common list-processing routines, such as filter, map, etc. FrTime provides primitives for converting between behaviors and event streams. One is `hold`, which consumes an event stream and an initial value and returns a behavior that starts with the initial value and changes to the last event value each time an event occurs. Conversely, `changes` consumes a behavior and returns an event stream that emits the value of the behavior each time it changes.

Scheme functions that are called on behaviors are said to be implicitly *lifted*: the expression is registered as a node in the dataflow graph and the result of the call is a new behavior that contains the value of calling the function on the latest values of its argument behaviors. These argument behaviors are the parent nodes of this expression in the dataflow graph. The resulting behavior of the call is its child node.

Scala.React [MRO10] is a framework that allows integrating and composing reactive programming abstractions into the statically typed programming language Scala using native functional and object-oriented abstractions. Scala.React provides several API layers allowing programmers to stepwise port callback-based code to a dataflow programming model. For this, it supports event streams as first class values and offers a number of operators to combine them, such as merge, map, filter, etc. They are implemented in the host language Scala and programmers can easily extend the framework with new operators following the conventions of the framework.

To circumvent the inversion of control problem, Scala.react provides *reactors*. Reactors are objects that are instantiated with a closure that can call two methods dedicated to dataflow: `next` and `delay`. `next` simply suspends the reactor until a given event stream emits a value. Once the stream raises an event, the call evaluates to the value

of the event and the reactor's execution continues. This can be done for example in a loop to periodically execute part of the reactor and continuing with the rest and continue the loop once a new event occurs (effectively implementing state machine-like behavior that would otherwise require complex and brittle orchestration of callbacks). Reactors can depend on multiple event streams. `delay` suspends the current reactor and continues after all pending messages have been propagated by the reactive framework. Reactors can either be specified to execute only once or multiple times.

Additional to datatypes that represent discrete events, Scala.React also provides datatypes that represent continuously time-varying values called *signals*. Signals are represented as a special kind of Scala functions of which the apply method is overridden. Hence, signals can be used as ordinary function calls, but they will be re-applied by the framework as soon as their dataflow dependencies change. What sets apart Scala.React from other reactive programming systems is its reliance on the strong typing of Scala and the fact that the dataflow dependencies of signals can be dynamically switched to other signals, event streams or reactors.

Fran (Functional Reactive Animation) [EH97] is a reactive programming library implemented on top of the functional language Haskell. It is mainly concerned with providing a declarative and first-class representation of time-varying values, which can either vary discretely or continuously, for graphics and animation. The former are represented by *events* and the latter by *behaviors*. Behaviors are seamlessly integrated into the Haskell base language: passing behaviors to function calls as arguments causes these calls to be reevaluated every time the system detects a change to one of the behaviors, and returns a new behavior of which the value will vary over time with the results from these calls. This way, event-driven code can be composed by means of ordinary functional composition. Other than for animation purposes, Fran has also been used for visual tracking [PHRH01]. In this tracking system, low-level C++ tracking components are scripted together using functional reactive programming.

Fran uses a discrete approximation of time using a fixed update interval (in the vein of synchronous languages as discussed earlier) that is used to poll for event and behavior changes and subsequently to update dependent expressions by an interpreter loop (i.e. it *pulls* for updates). This has the advantage that this update interval can be used as the slowest rate of update (depending on hardware and language runtime performance), which is an important property for smooth animation. The downside is that it does not work for distributed systems where events are not generated at a predefined clock rate but arrive at arbitrary times from remote communication partners.

For performance reasons, the Fran library was later reimplemented to in addition to pull-based evaluation also support *push-based* evaluation [Ell09]. Although some time-varying values change continuously, others change only at discrete moments (say in response to a button click or an object collision), while still others have periods of continuous change alternating with constancy. In all but the purely continuous case, pull-based implementations waste considerable resources, recomputing values even when they do not change. In those situations, push-based implementations can operate much more efficiently, focusing computation on updating values that actually change. Another serious problem with the pull approach is that it imposes significant latency. The delay between the occurrence of an event and the visible result of its reaction, can be as much as the polling period (and is on average half that period).

The ideas from Fran were later used to implement a version of functional reactive programming on top of Haskell's arrow combinators and notation [NCP02]. The difference with Fran is that this system does not treat time-varying values as first class citizens in the language to obtain additional operational features such as real-time

guarantees. This version led to the **Yampa** [HCNP03, CNP03] language, which is a domain-specific language for programming mobile robots embedded in Haskell. To meet real-time constraints, it relies on the Arrows framework [Hug00]. This means that time-varying values are not allowed as first-class values. Instead, the programmer has access only to signal functions. A signal function is just a function that maps time-varying values to time-varying values. One cannot directly build signal functions or apply them to time-varying values. Instead, Yampa provides a set of primitive signal functions and a set of special composition operators (or "combinators") with which more complex signal functions can be defined. Additionally, it also provides a number of stateful signal functions, i.e. functions that accumulate information over time, such as `integral`. Stateful functions must either be pre-defined or be defined in terms of other stateful signal functions because they depend intimately on the underlying time-varying semantics.

Additional to classic combinators, Yampa also provides switching combinators, which allow signal functions that are working on time-varying values to be dynamically replaced by other signal functions when certain events are observed in these time-varying values. For this, Yampa provides a number of primitives to generate such events from time-varying values. Switching effectively causes the underlying dataflow graph to be rearranged. However, Yampa is intended for local applications and has no support for distributed programming.

Frappé [Cou01] is an implementation of functional reactive programming in Java following the ideas from Fran. The primary contribution of Frappé is its integration of the FRP event/behavior model with the Java Beans event/property model. At the interface level, any Java Beans component may be used as a source or sink for the event and behavior combinators. This provides a mechanism for extending Frappé with new kinds of I/O connections and allows functional reactive programming to be used as a high-level declarative model for composing applications from Java Beans components. At the implementation level, the Java Beans event model based on classic callbacks is used internally by Frappé to propagate events and changes to behaviors. This allows Frappé applications to be packaged as Java Beans components for use in other applications. For example, Java Beans notify registered listeners whenever a new value is assigned to one of their properties by means of reflection.

To write programs in Frappé, the programmer simply instantiates a number of Java Beans components and connects those components together using the Frappé classes corresponding to functional reactive programming combinators. The program then relinquishes control to the Java runtime library's main event loop. Frappé assumes that event processing is single-threaded and synchronous. That is, all primitive Java Beans events used as event or behavior sources for Frappé must be fired from the system's event dispatching thread, and each event must completely propagate through the dataflow graph before the next event is handled. This makes Frappé unsuitable for mobile RFID-enabled applications where distributed event producers and consumers interact asynchronously and in parallel.

Flapjax [MGB+09] is a reactive programming system targeting event-driven web-based applications. It is built on top of JavaScript and runs on unmodified browsers and readily interoperates with existing JavaScript code. It is usable as either a programming language that is compiled to JavaScript or as a JavaScript library. In the latter case, the programmer must explicitly lift invocations on reactive values. Flapjax borrows the notions of behaviors and event streams of FrTime (discussed above) and uses the same execution strategy of dataflow-dependent expressions. Additionally, it provides abstractions tailored for web development, such as event streams wrapping

asynchronous web server acknowledgements, web-form fields and buttons, etc. Flapjax is a reactive programming system targeting web-based distributed systems, and cannot be used for mobile RFID-enabled applications where communication partners might not implement web services or similar http-based interfaces.

Coherence [Edw09] is a programming language that embodies an object-oriented variant of reactive programming called *coherent reaction*. Coherent reaction is a new model of change-driven computation that coordinates effects automatically. State changes trigger events called reactions that in turn change other states. A coherent execution order is one in which *each reaction executes before any others that are affected by its changes*. A coherent order is discovered iteratively by detecting incoherencies as they occur and backtracking their effects. All the effects of a prematurely executed reaction are rolled back, as in a database transaction, and are reexecuted later.

This mechanism is embedded in the prototype-based (i.e. classless) [Lie86] object-oriented language Coherence by distinguishing two types of fields encapsulated by objects: regular fields (as in classic object-oriented languages), and *derived fields*. The expression that assigns a value to such a derived field is reexecuted every time the field is accessed. This is called *derivation*. This means that the derived field will always be up to date with respect to other fields from which it is derived, and which might have changed after initializing the derived field. Such derivations are the fundamental concept in Coherence and are guaranteed to have no side effects, i.e. derivations triggered by recalculating the derived field are executed *after* the derivation.

Derivation is bidirectional: changes to derived variables can propagate back into changes to the variables they were derived from. This process is called *reaction*, and is used to handle external input. The Coherence runtime makes certain structures visible to certain external interfaces. All input takes the form of changes to such visible fields, which react by changing internal fields, which in turn can react and so on. Multiple input changes can be submitted in a batch, and the entire cascade of reactions is processed in a transaction that commits them atomically or not at all. Output consists of reading visible fields, which are rederived if necessary from the latest changed state. How a field reacts to a state change can be overridden by the programmer, to support custom reactions. Such a custom reaction is specified as a set of Coherence statements that execute when the field is changed. In these reactions, both the pre-state as the post-state of the involved fields can be accessed: the post-state is represented by the same field name but with a prime appended. Since Coherence relies on arbitrary expressions that are run in transactions to find a coherent execution order and that can be (transitively) aborted in case of incoherencies, Coherence programs hard to distribute in mobile ad hoc networks.

Approaches that do not rely on dataflow to circumvent inversion of control are rare. **Synchronous C++** extends C++ with concurrent active objects that synchronize by means of the rendez-vous protocol. With Synchronous C++, one can use rendez-vous together with a special `select` construct. This construct can have a number of `case` branches and causes the active object to wait for another active object to call a specific method (one method for each branch). Its thread of execution remains blocked until one of these methods are invoked by other active objects. As soon as this happens, the thread is woken up and executes the corresponding branch of the `select` construct with the result of the method call. Applying this to event-driven programming is done by representing events as method calls on which is blocked in a `select` statement in an infinite loop. If an event is signaled by calling such a method, the corresponding branch of the `select` statement is executed. These concepts were later applied to Java as well [PE02]. The explicit blocking contradicts our time decoupling requirement.

Glitch Prevention in Functional Reactive Programming

For the languages discussed in this section, the interpreter, virtual machine or compiler of the language *implicitly* constructs the dataflow graph representing the running program behind the scenes. The nodes of this graph represent expressions to be evaluated, which depend on values that vary over time (e.g. driven by external events), such as event streams that discretely signal events, synchronous values that update themselves at fixed rates (in synchronous languages), or continuously varying values. These values can be the results of other reactive expressions in the program, which act as parent nodes in the dataflow graph.

The notion of parent nodes in the graph implies that the nodes of the graph must be topologically sorted in some way. Indeed, if the execution order of reactive code depending on time-varying values is not carefully coordinated, *glitches* will occur. For example, consider the expression below, where `seconds` is a time-varying value that contains the current number of seconds and is incremented every second, signaling a new event.

```
seconds < (1 + seconds)
```

This should always evaluate to `true`, since `n` is always less than `n + 1`. In languages that automatically *reevaluate* reactive expressions, each change in `seconds` triggers recomputation of the overall expression and the inner `1 + seconds` reactive expression, and the order in which these values are recomputed affects the answer. If it updates `1 + seconds` first, then the top-level `<` compares up-to-date versions of `seconds` and `1 + seconds`, yielding `true`. On the other hand, if it updates the top-level reactive expression first, it then compares the up-to-date `seconds` with the stale `1 + seconds` – which is equal to the new value of `seconds` – yielding `false`.

To prevent glitches, reactive programming runtimes must implement a traversal strategy for evaluating dataflow code in the dataflow graph. The crucial property is that no time-varying value should update until everything on which it depends is also up-to-date. The concrete traversal strategy depends on the expressivity of the language: languages that allow side effects, cyclic dependencies, higher order functions etc. require a more complex strategy, as is the case in for example FrTime. In FrTime, each reactive expression in the dataflow graph is assigned a *height*, which exceeds that of all its event producers. Expressions of the same height are evaluated (conceptually) simultaneously, and only if their evaluation completes, the next batch of expressions of the next height (that might depend on the previous batch) is evaluated. The dataflow traversal is said to be *stratified*.

3.3.2 Programming with Dataflow Graphs

Explicit programming with dataflow graphs is currently used mostly in the form of the *coarse-grained* dataflow model. In such models, the dataflow paradigm is used to orchestrate the control flow between different modules (possibly running in parallel and/or distributed) that can be of an arbitrary level of abstraction, usually implemented in a conventional programming language.

When looking at the characteristics and requirements of mobile networks and the applications running on top of them, we observe that the dataflow model may provide a very suitable coordination model for this kind of applications. These applications consist of different distributed components running in parallel that in many cases have to be invoked whenever some external data is fed to them (event-driven architectures).

Hence, the driving force for program execution in such applications is not the control flow, which is explicitized by the order of statements in an imperative textual program, but the data flow, which is implicit in an imperative textual program. Furthermore, in many cases these data come in the form of streams, such as continuous sensor readings. *Explicitly* specifying dataflow graphs has the advantage that the coarse-grained control flow, which in conventional languages would become very implicit in a complex interplay of different event handlers, is now represented in a very explicit (sometimes visual) notation based on the dataflow coordination model.

Since dataflow programming is not a new concept, over the years many different systems that allow manipulating dataflow graphs have been conceived. Industrial examples are LabVIEW and Prograph. LabVIEW was the first software program to include graphical, iconic programming techniques to make programming more transparent and the sequence of processing visible to the user [Kal95]. LabVIEW is based on the G visual dataflow language and the concrete implementation in the LabVIEW environment is primarily used for data acquisition, processing and monitoring in a lab setting. LabVIEW does not use dataflow for expressing distribution and/or parallelism, but for the graphical composition of software components that interact with lab hardware (inherently resulting in sensor-driven systems).

Another language that does use dataflow for expressing distribution and concurrency is Distributed Prograph [CGL96]. In Distributed Prograph, program code of dataflow operators is dynamically sent to remote processing units. The scheduling of the execution of these operators happens at runtime, but the processing units themselves have to be known at compile time, which is unrealistic in mobile ad hoc networks.

A more recent domain where dataflow proved its use is web mashups [BP07]. Similarly, workflow management systems sometimes provide an explicit view on data flow of a workflow to further orchestrate web services. Current workflow engines however require heavy-weight infrastructure and map poorly on an entirely event-driven execution [BPA06].

Below we discuss a selection that takes into account at least some of the mobile RFID-enabled application requirements mentioned in section 2.3.4 more elaborately.

Hyperflow [Kim93] is a visual dataflow language for a pen-based multimedia computer system designed for school children. Yet it is versatile enough to be used as a system programming language. Dataflow (concurrent) processes are represented as boxes that communicate over the links that interconnect them. Dataflow processes communicate over these links through the exchange of signals, either discrete or continuous. Each process communicates with the user through its own interface box by displaying on the box information about the process and by receiving information pen-scribed on the box.

Hyperflow supports four different communication modes: synchronous discrete signal communication (regular arrow), asynchronous discrete signal communication (dotted arrow), channeling continuous signals (fat arrow), and broadcasting continuous signals (dotted box around producer and consumer nodes). The latter are mainly used for multimedia streaming. Asynchronous discrete signals are buffered in queues when they arrive at the processing elements and are served sequentially to be processed by the processing element in FIFO manner.

The body of and the commands supported by Hyperflow processing elements can be encoded individually either in Hyperflow's dedicated syntax or in any textual programming language such as C, Pascal, or assembly language. Hyperflow does not support distributed programming.

Aurora [ACc+03] is an event stream processor in which data is assumed to come from a variety of data sources, such as computer programs that generate values at regular or irregular intervals or hardware sensors. A data stream is a potentially unbounded collection of tuples generated by a data source. Unlike the tuples of the relational database model, stream tuples are generated in real-time and are typically not available in their entirety at any given point in time. Aurora processes tuples from incoming streams according to a specification made by an application administrator. Aurora uses a boxes and arrows paradigm found in most process flow and workflow systems to allow the application administrator to specify dataflow graphs that process event streams. Nodes in the dataflow graph represent stream operators that accept one or more input streams and output one or more processed streams. Aurora offers a number of predefined stream operators, such as filter, union, windowed sort, map, intersection, join... Aurora utilizes a centralized architecture that hosts a scheduler that determines which stream operator to run. Aurora must constantly monitor the quality of service of output tuples. This information is important since it drives the scheduler in its decision-making.

Aurora was later extended to **Aurora*** [CBB+03]. Aurora* does not require a centralized architecture, such that the dataflow program can be distributed. Every network node hosting an Aurora operator is now responsible itself for continuously monitoring its local operation, its workload, and available resources (e.g. CPU, memory, bandwidth, etc.). If a machine finds itself short of resources, it will consider offloading operators to another appropriate Aurora node. All dynamic reconfiguration will take place in such a decentralized fashion, involving only local, pair-wise interactions between Aurora nodes.

Medusa [CBB+03] is another variant of Aurora of which participants can represent a collection of computing devices administered by a single entity. Hence, participants range in scale from collections of stream processing nodes capable of running Aurora and providing part of the global service, to PCs or PDAs that allow user access to the system (e.g. to specify queries), to networks of sensors and their proxies that provide input streams. Participants provide services to each other by establishing contracts that determine the appropriate compensation for each service. Medusa uses a market mechanism with an underlying currency that backs these contracts. Each contract exists between two participants and covers a message stream that flows between them. One of the contracting participants is the sending participant; the other is the receiving participant. Medusa models each message stream as having positive value, with a well-defined value per message; the model therefore is that the receiving participant always pays the sender for a stream. In turn, the receiver performs query-processing services on the message stream that presumably increases its value, at some cost. The receiver can then sell the resulting stream for a higher price than it paid and make money. Some Medusa participants are purely stream sources (e.g. sensor networks and their proxies), and are paid for their data, while other participants (e.g. end-users) are strictly stream sinks, and must pay for these streams. However, most Medusa participants are interior nodes (acting both as sources and sinks). They are assumed to operate as profit-making entities; i.e. their contracts have to make money or they will cease operation.

Both in Aurora* and Medusa, each node hosting an operator sends periodic heartbeat messages to its upstream neighbors. If a server does not hear from its downstream neighbor for some predetermined time period, it considers that its neighbor failed, and it attempts to connect to another node hosting the same operator. Still, all Aurora nodes have to be known and specified into a catalog before deploying the dataflow program into the network, rendering Aurora* and Medusa unsuitable for mobile RFID-enabled

applications.

Later, the features of Aurora* and Medusa were merged in a second generation query processor called **Borealis** [ABC+05]. Borealis additionally supports the dynamic revision of query results (i.e. go back in time and change a tuple at a certain point in time in an event stream) to apply corrections to old tuples and the dynamic modification of running queries and incorporates several optimizations.

Solar [CLK04] is an object-oriented, distributed graph-based abstraction for collecting, aggregating and disseminating context information in pervasive computing applications. Solar models context information as events, which are produced by sources, flow through a directed acyclic graph of event processing operators, and are delivered to subscribed applications. Each input port of an operator has a unique identifier to distinguish it from other ports. Connected operators communicate events either in a *push-based* or *pull-based* fashion. In the first case, the event source voluntarily passes event data units to its subscribers, while in the latter case subscribers query the event source at their own pace for new events.

Applications describe their desired event stream as a tree of operators that aggregate low-level contextual information published by existing sources into the high-level context information needed by the application. Event processing operators emit a single event stream to an arbitrary number of subscribed applications, or other event processing operators. Solar distributes these operator trees into the network and is designed to minimize the traffic across the network edge and to allow the computation to be distributed. While each application can build its own operator tree, to scale to a large number of applications these operators are reused between applications' operator trees. Upon arrival of a new subscription tree, Solar attempts to identify subtrees that match a subtree of an already present subscription tree. This matching is defined recursively: two operators match if they are objects of the same class, have the same parameters and have the same subscriptions. When a match is found, the subtree is clipped from the new subscription tree, replacing it with a subscription to the output of the existing subtree; thus the two subscriptions share the subtree.

Solar requires a centralized server (called a star) that maintains a representation of the operator graph and service requests for new subscriptions. When the star receives a new subscription tree description, it parses the description, and matches the subscription tree against its internal data structure representing the operator graph to see whether there are some existing operators that can be shared and reused. When it decides to deploy a new operator it instantiates the operator's object on one of the available hosts (called planets), which periodically register themselves with the star. The star determines which planet should host the new operator by considering the planet's load and network traffic between planets, and moves the operator as mobile code to that planet. In essence, it attempts to map the operator graph onto the planetary network to distribute load and avoid congestion. When deploying new subscriptions, the star tells the planets to arrange a subscription from one of its operators to another operator, possibly in another planet. Thus the planet maintains all the subscriptions for each of the operators it hosts. When an operator publishes an event, the hosting planet delivers the event to all the subscribed operators (which may reside on different planets) and applications. When a planet receives an event, it dispatches the event to the appropriate operator it hosts.

Although this architecture allows the operator graph to reconfigure at runtime in the face of network failures, network load changes, changing subscriptions etc., it still relies on a centralized star to which all planets have to periodically connect to maintain the dataflow graph. This makes Solar unsuitable for mobile RFID-enabled applications.

Planets maintain an outbound event queue for each event source they host, to buffer event messages when the planet is disconnected from its subscribers and resend them later when it is again connected. This way, Solar supports fault tolerant event communication. Each operator has a dedicated thread to process incoming event notifications serially.

Solar represents events as arbitrary serializable Java objects. Operators are also serializable Java objects that implement a simple publish/subscribe interface, such that they can be serialized and moved to another planet if required. Subscription trees are specified as XML-documents, in which operators are referred to by giving them unique names. These names can be reused in different subscription trees. These names could be regarded as topics and in publish/subscribe terminology we could say that Solar offers topic-based subscriptions.

iQL [CLC+02, CPWY02] is a nonprocedural event processing language based on entities called composer functions (or composers in short) and data sources. A composer has a current value computed from input values, in a manner determined by a composer specification. A composer's input values come from data sources. Some data sources are pervasive networked sources such as web services and sensors; some are other composers. Data sources are advertised to the runtime system. A composer specification includes requirements on data sources. The runtime system discovers advertised data sources satisfying these requirements, binds them, and executes protocols that deliver their data to the composer. As quality-of-service and quality-of-information properties of the data source change (e.g freshness of data, confidence in data, or precision of measurement...), these changes are advertised to the runtime system, which may rebind to different data sources that better meet the requirements. Binding can either happen to a single data source or to all data sources which match the composer's requirements. In the latter case, the values produced by the different data sources are put in a list with for each entry the latest value produced by one of the data sources.

Composer functions work on streams of values. They are either autonomously generated by data sources or by polling the data sources. iQL provides language constructs to explicitly wire a dataflow graph of composers and their data sources, which feed each other's input values. When one input value of a composer function changes, the composer function is reevaluated, and subsequently all composer functions to which it acted as an input value as well, propagating the change event throughout the dataflow graph. Events are propagated by means of asynchronous message passing to cater for temporarily disconnected data sources.

Due to its restricted nonprocedural nature, some patterns are hard to express in iQL and makes iQL hard to integrate with general-purpose programming language. E.g. since there is no notion of variables and scoping, only parameters representing new values of data sources can be used in composer functions. Hence, programmers have to rely on explicit caching language constructs to remember previous values of streams in composer functions, for example to implement an average of previous values. Similarly, iQL provides dedicated operators to generate compound events over events detected in the past. Custom operators, which have to be implemented in Java, can be plugged in.

iQL's runtime system is based on a centralized server that contains all data source advertisements and subscriptions and acts as a broker among data sources and composers. Therefore, iQL's current runtime system cannot be used for mobile RFID-enabled applications.

3.3.3 Evaluation

In this section, we give an overview of the surveyed dataflow technologies and evaluate their distinct properties with respect to the requirements for mobile RFID-enabled applications discussed in section 2.3.4. The results are summarized in table 3.3. Desirable properties for mobile RFID-enabled applications are again shown in italic font.

Control Flow Management The programming systems discussed above can be categorized in dataflow systems that use a dataflow graph as an internal representation of the dataflow program, and dataflow systems in which the dataflow graph *is* the program which the programmer has to construct explicitly.

In the first category (comprising synchronous languages, functional reactive languages, and parallel languages that use dataflow to coordinate side effects), the interpreter, virtual machine or compiler of the language constructs the dataflow graph representing the running program behind the scenes. The nodes of this graph represent expressions to be evaluated, which depend on values that vary over time (e.g. driven by external events), such as event streams that discretely signal events, synchronous values that update themselves at fixed rates (in synchronous languages), or continuously varying values. These values can be the results of other reactive expressions in the program, which act as parent nodes in the dataflow graph. The dataflow graph is traversed in response to events in such a way that glitches are prevented (see section 3.3.1).

The second category comprises languages that use programmer-defined dataflow graphs as an idiom to express an event-driven (Hyperflow) or event processing (Borealis, Solar, iQL) program. In these languages, dataflow is neither in the core of a dedicated general-purpose language nor embedded in a host language, but used as a higher level domain-specific language that steers the execution of modules implemented in one or more lower-level general-purpose languages.

Event Representation Reaction is the core execution driver of dataflow programs and hence does not require special programming techniques such as callbacks. In older dataflow languages, which were more concerned about parallelism then reacting to external events, the programmer had to explicitly advance the state of the program by calling a `next` primitive on a stream, that produces the next value in the stream by executing all dependent code. In most functional reactive programming languages, reaction happens by reevaluating function calls that depend on time-varying values. This can also be applied to method calls in object-oriented languages, but care must be taken when introducing side effects (since reactive code can be executed unexpectedly). Although not a typical dataflow language, the Coherence language discussed above caters for this case by employing versioned objects that are updated in parallel when dependent object fields are updated by other objects, and incoherent states are automatically detected and rolled back. The downside is that it relies on transactions over these objects that cannot be distributed in an unreliable network such as a mobile ad hoc network.

Event processing languages react similarly, but event stream operators are usually restricted in expressivity to be efficiently scheduled in the face of a large number of events to be processed. They act on one or more input streams and produce an output stream that can be used as an input stream to one or more subsequent stream operators. These stream operators are usually implemented in a lower level general-purpose programming language and can in many cases be connected in a dataflow graph using a visual syntax.

	Space decoupling	**Arity decoupling**	**Time decoupling**	**Event representation**	**Infrastructureless**	**Control flow management**
Lucid	No	No	No	*streams*	No	*implicit dataflow*, `next` operator
Lustre	No	No	No	*streams*	No	*implicit dataflow, reevaluation*
Scala.React	No	No	No	*streams, time-varying values*	No	*implicit dataflow, reevaluation*
Fran	No	No	No	*streams, time-varying values*	No	*implicit dataflow, reevaluation*
Yampa	No	No	No	signal functions	No	*implicit dataflow, reevaluation*
Frappé	No	No	No	signal functions	No	*implicit dataflow, reevaluation*
FrTime	No	No	No	*streams, time-varying values*	No	*implicit dataflow, reevaluation*
Flapjax	No	No	*Yes*	*streams, time-varying values*	No	*implicit dataflow, reevaluation*
Coherence	No	No	No	versioned objects	No	*derivation, reaction*
Hyperflow	No	*Yes*	*Yes*	*streams, time-varying values*	No	*explicit dataflow*
Borealis	*Yes*	No	*Yes*	tuple streams	No	*explicit dataflow*
Solar	*Yes*	*Yes*	*Yes*	*object streams*	No	*explicit dataflow*
iQL	*Yes*	*Yes*	*Yes*	*object streams*	No	*explicit dataflow*

Table 3.3: Survey of dataflow programming technology.

Space, Arity and Time Decoupling Functional reactive programming languages where originally not intended for distributed computing. Only Flapjax uses Javascript as a host language and supports time decoupling through asynchronous AJAX calls. Dataflow languages that make the dataflow graph explicit do this in many cases for the coordination of distributed event-driven software. It must come as no surprise that all of these languages offer some decoupling features. An interesting feature of Solar and iQL is that although they require the dataflow graph to be specified up front, they allow events to be broadcasted to a number of late-bound event consumers. This is possible because they rely on an underlying publish/subscribe architecture to subscribe to and deliver events. Solar's distribution model requires it to move mobile code representing dataflow operators to different nodes in the network while executing. Hyperflow also supports broadcasting events to a number of event consumers, but they have to be specified up front in the dataflow graph using a visual syntax to denote the set of interested event consumers.

Infrastructureless Operation All surveyed systems require additional infrastructure outside of the node running (part of) the dataflow program itself.

Distributed Reactive Programming

There are two main issues in distributing dataflow programs. The first is that glitches cannot be avoided that easily in a distributed setting. The second is that maintaining the dataflow dependencies in a distributed dataflow program tightly couples the dependent distributed application components, and hence renders them less resilient to network failures and reduces overall scalability.

Avoiding glitches in a distributed setting Glitches occur because of dependent code that is executed in the wrong order. This can easily happen in a distributed setting where events are communicated over the network and are hence delivered with a delay of which the severity depends on different factors, such as the underlying network technology, network congestion, network failures, etc. Hence, *time-stamping* of events is necessary to allow them to be correctly ordered at the receiver side. The problem here is that distributed clocks can diverge, which can be problematic for ordering events that happen in close succession.

A possible solution could be to use a centralized entity to which all parties involved in the distributed dataflow program connect and that is solely responsible for ordering events. This centralized approach of course introduces a single point of failure and possibly a considerable communication overhead as all parties involved in the system have to communicate with a single host for every event they signal and to receive every event propagated to them as well, potentially limiting scalability.

A decentralized solution could be to accept that events in close succession cannot be ordered as a fact of life and take into account a minimum time interval in which events are considered to occur simultaneously. This is similar to ideas found in real-time synchronous languages where the system is assumed to react atomically to events *before any other events occur* and a global clock with a minimal tick rate determines the time interval. This minimal tick rate could be used as the maximal amount of which distributed clocks may diverge in a distributed dataflow program. As long as it can be guaranteed that all the clocks in the system do not diverge more than this time interval, glitches can be prevented while keeping a decentralized architecture. The

applicability of this assumption depends on a number of factors such as the number of parties involved in the distributed interaction, the amount of clock divergence, the quality of the network, and, most importantly, the nature of the dataflow program. For programs that have to quickly react on events occurring in very close succession this approach might not be feasible, while for programs that work on human time-scales (such as seconds, minutes...) this assumption might be acceptable. Decentralized solutions to mitigate this problem exist, but they highly depend on the physical layer of the mobile ad hoc network [EGE02].

Coupling Maintaining dataflow dependencies among dependent reactive application components tightly couples these application components. Since in most cases this coupling does not have to be explicitly managed by the programmer, this is a non-issue for local applications. However, for distributed applications, a loose coupling of the communicating parties is required to achieve scalability in very large systems [CRW00] or to be applicable in mobile ad hoc networks [HGM04]. In these cases, network failures prevent events to be propagated among distributed application components that depend on each other, causing the application to halt or causing glitches.

In section 3.2 we have concluded that publish/subscribe can offer the necessary decoupling in space, time and arity. One of the pillars of this dissertation is the integration of publish/subscribe with dataflow programming.

3.4 Survey of Programming Technology for Sensor Networks

In the world of sensor networks, reacting to sensor readings is everyday business. The difference with mobile RFID-enabled applications is that there are only a handful of applications: sensing structure rigidity, sensing environmental conditions such as temperature and humidity, capturing object or creature location and movement, etc. These applications require a closed system that is specially built for these types of applications. Hence, sensor network programing technology usually focuses on making sensor nodes work together while saving power and reducing network usage as much as possible (*node-centric*) on the one hand, and querying sensor readings observed in the entire sensor network from a user level (*network-centric*) on the other hand. Still, sensing and reacting to changes to what is being sensed in a wireless network is at the heart of sensor network programming technology and hence can offer interesting ideas that can be applied to mobile RFID-enabled applications as well.

Sensor networks consist of a (usually large) number of sensor nodes. These are tiny devices with enough processing power to gather sensor readings, process them, and make decisions how to communicate and jointly process these readings with other reachable sensor nodes to maximize data quality and minimize network usage and power consumption. Communication happens via a wireless radio that allows sensor nodes to spontaneously form a mobile ad hoc network. One or more edges of the mobile ad hoc network communicate with a base station: a fixed infrastructure that allows collecting processed sensor data from the sensor nodes, further processing it, deploying queries in the sensor network, etc., allowing the user to interact remotely with the sensor network. However, such a base station can only communicate with one or a few sensor nodes: it cannot control the entire sensor network. Furthermore, if all raw data is sent to base stations for further processing, the volume and burstiness of the

traffic may cause many collisions and contribute to significant power loss. This means that in-network processing is often necessary: to achieve energy-efficiency, the amount of data transmission must be reduced by way of summarizing and compressing the data inside the network.

In this section, we survey existing programming technology for sensor networks. We will not cover issues such as power consumption and routing. Instead we will evaluate the different technologies with respect to the requirements for mobile RFID-enabled applications. The surveyed systems are classified as by Suighara and Gupta [SG08]. *Node-centric* abstractions offer programming primitives to program a single node and make it communicate with other sensor nodes. *Group-centric abstractions* provide a set of programming primitives to handle a group of nodes as a single entity. These define APIs for intragroup communications and thus make it easier for the programmers to describe collaborative algorithms. *Network-centric* abstractions, or equivalently *macroprogramming*, treat the whole network as a single abstract machine.

3.4.1 Node-Centric Programming

We will not cover all sensor network programming technology, but focus on event-driven programming abstractions instead. Kasten and Römer observe the following problems in complex event-driven programs for sensor networks [KR05]:

> "Event handlers must not monopolize the CPU for any significant time, thus operations need to be non-blocking. Therefore, at any point in the control flow where an operation needs to wait for some event to occur, the operation must be split into two parts: a non-blocking operation request and an asynchronous completion event. The completion event then triggers an action that continues the operation. As a consequence, even a seemingly simple operation can lead to event cascades - an action calls a non-blocking operation, which causes an event to occur, which, in turn, triggers another action. Breaking a single conceptual operation across several actions also breaks the operation's control flow and its state. This has two implications for the programmer. Firstly, as breaking operations into multiple functions effectively discards language scoping features, programmers need to manually manage the operation's stack. This is called manual stack management. Secondly, programmers must guarantee that any order of events is handled appropriately in the corresponding actions. This is called manual flow control."

To tackle these problems they propose the **Object State Model** (OSM), which is represented in a textual programming language for sensor networks. OSM is based on finite state machines. Finite state machines are based on the concepts of states, events, and transitions. A FSM consists of a set of states and a set of transitions, each of which is a directed edge between two states, originating form the source state and directed towards the target state. Transitions specify how the machine can proceed form one state to another. Each transition has an associated event. The transition is taken (it "fires") when the machine is in the transition's source state and its associated event occurs. FSMs can be thought of as directed, possibly cyclic graphs, with nodes denoting states and edges denoting transitions. As in event-based programming, actions specify computational (re)actions. Conceptually, actions are associated with transitions or states. States can be hierarchically organized: a state can encompass a smaller partial FSM. Through the explicit notion of states, the association of events to actions is no longer

static. Rather, the invocation of actions becomes a function of both the event and the current program machine state.

OSM extends the notion of finite state machines with state variables, which hold information that is local to a state or state hierarchy. Their scope is limited to a state (and its substates), thus allowing to reclaim memory upon leaving the state. Additionally OSM allows to guard transitions by a predicate over event parameters as well as over the variables of the source state. The transition then only fires if the predicate holds (i.e. evaluates to true) on the occurrence of the trigger event.

FSMs rely on a discrete representation of time, similar to the synchronous languages discussed earlier in section 3.3.1. Since the execution of actions consumes a non-zero amount of real time, events may arrive in the system while some action is currently being executed. Those events cannot be handled immediately. Therefore, arriving events are always inserted into a FIFO queue. In the discrete time model, events can occur concurrently. The event queue should preserve such sets of concurrent events, rather than imposing an artificial total ordering on concurrent events by inserting them one after another into the queue. For this purpose, the event queue operates on sets of concurrent events rather than on individual events. The enqueue operation takes a set of concurrent events as a parameter and appends this set (as a whole) to the end of the queue. The dequeue operation removes the set of concurrent events from the queue that has been inserted earliest. Whenever the system is ready to make a transition, it uses the dequeue operation to determine the one or more events to trigger transitions. After each discrete time step, the earliest set of concurrent events is dequeued unless the queue is empty. The set of dequeued concurrent events could trigger different transitions in a single state machine. To resolve such ambiguous cases, priorities can be assigned to transitions, such that the transition with the highest priority is triggered in each state machine. The combination of concurrency and hierarchical scoping of state variables in hierarchical FSMs can lead concurrent writes of these state variables. It is up to the programmer to synchronize them manually.

TinyLIME [CGG+05] is an extension of LIME (Linda Meets Mobility) [MPR01], a tuple space middleware for mobile ad hoc networks, targeting sensor networks and implemented in nesC on top of TinyOS [GLvB+03]. Just like in Linda [Gel85] and LIME, coordination among distributed processes occurs through the writing and reading (and/or removing) of tuples into a conceptually shared memory called a tuple space. Manipulating the contents of the tuple space happens through pattern matching on tuples using template tuples. If multiple tuples match a template, the one returned is selected non-deterministically. A process attempting to access a non-existent tuple in the tuple space in Linda remains blocked until the tuple is added to the tuple space, leading to synchronous access. LIME extends this model with asynchronous *reactions*: other than being blocked, processes can also be notified asynchronously by triggering a callback when a matching tuple arrives.

Communication in Linda is decoupled in time and space, i.e. senders and receivers do not need to be available at the same time, and mutual knowledge of their identity or location is not necessary for data exchange. This decoupling makes the model ideal for the mobile ad hoc environment where the parties involved in communication change dynamically due to their movement through space. To support mobility, the LIME model breaks up the Linda tuple space into multiple tuple spaces each permanently attached to a mobile component, and defines rules for the sharing of their content when components are able to communicate. LIME's transient sharing of tuple spaces provides a view where it is as if all tuples were in the same tuple space, and accessible to all processes. This shared tuple space is called the *federated tuple space*. However,

when one process disconnects, its tuples are no longer accessible to the others, but remains accessible to itself.

3.4.2 Group-Centric Programming

The basic idea of group-centric abstraction is to provide a language construct that handles multiple nodes collectively and a set of operations on it so that people can program the behavior of a group. This way, application design can be simplified by abstracting over the details of low-level communication, data sharing, and collective operations.

Smart Messages [KBX+04] is a modification to the Java virtual machine targeting mobile ad hoc networking applications (instead of pure sensor networks). Instead of passing data end-to-end between nodes, a Smart Messages application *migrates* to nodes of interest named by content and executes there. Each node hosts a virtual machine that executes smart messages in their own thread and a name-based memory, called tag space. The smart messages use the tag space for content-based naming and as a shared memory persistent across smart message executions.

Executing a smart message can lead to the execution of one or more other smart messages and can be bounded in time by sending a timeout value with it. Although once migrated the execution of a smart message happens asynchronously with respect to the sender, the migration itself happen synchronously and must be invoked explicitly by the programmer, who also must explicitly send along any data needed for the remote execution. This means that the local part of the application blocks until a suitable node is detected to migrate the smart message to.

Although smart messages are executed in their own thread, the default scheduler will put them in a queue and execute them in sequence to prevent race conditions on the data that is shared among them. For coordinating asynchronously arriving smart messages, the platform offers a simple synchronization mechanism that allows a smart message to block on a tag in the tag space until another smart message performs a write on it. A blocked smart message is appended to the message queue, such that the scheduler can proceed with the next message in the queue.

Smart messages carry their own application-specific routing code and route themselves to each node of interest, by hopping from reachable node to reachable node. An example of a simple content-based routing strategy could be to only migrate to the nodes that contain the tags in the tag space that are needed for the execution of the smart message. More complex routing strategies based on these tags can be devised, e.g. by communicating location information via these tags to obtain geographical routing.

Hood [WSBC04] is a grouping abstraction for sensor networks implemented on top of TinyOS in nesC. Hood allows a node to identify a subset of nodes around it by a variety of criteria and share state with those nodes. For example, Hood can define a one-hop neighborhood over which light readings are shared and a two-hop neighborhood over which both locations and temperature are shared. Once the neighborhoods are defined, Hood provides an interface to read the names and shared values of each neighbor (represented as key/value pairs). Beneath this interface, Hood is automatically discovering neighbors and caching the values of their attributes while simultaneously sharing the values of the nodes own attributes.

Key to the implementation of Hood is the *broadcast/filter* mechanism used for both data sharing and neighborhood discovery. When attributes are shared they are always broadcast. The receiving nodes “filter” the incoming attributes to determine which nodes are adequate neighbors and which of their attributes should be cached. Hence,

updating cached values is achieved behind the scenes using a broadcast-based publish/-subscribe architecture, where subscriptions are broadcasted and maintained locally by each node and broadcasted events update cached values if necessary. This decouples the owner of an attribute from the observers of the attribute. Hood promises only the weak sharing semantics that unreliable, low-bandwidth networks can provide; the neighbors in the neighbor list and the cached values of a neighbors attributes represent only the best or most recently observed. Any stronger guarantees about consistency, coherence or reliability are intentionally deferred to the application level.

Abstract Regions [WM04] are a family of spatial operators that capture local communication within regions of the network and are implemented on top of TinyOS in nesC. Abstract regions may be defined in terms of radio connectivity, geographic location, or other properties of nodes. They provide interfaces for identifying neighboring nodes, sharing data among neighbors, and performing efficient reductions on shared variables. In addition, abstract regions expose the trade-off between the accuracy and resource usage of communication operations.

Before performing other operations on an abstract region, each node initiates the process of discovering neighboring nodes. Depending on the type of region, this may require broadcasting periodic advertisements, collecting information on node locations, or estimating radio link quality. This is a continuous process, and each node is informed of changes in the region membership set, due to nodes joining, leaving, or moving within the network. A node may terminate this process at any time to avoid additional discovery messages. When terminated, the neighbor discovery operator returns a quality metric that measures the accuracy of the region formation, such as the percentage of candidate nodes that responded to the discovery request. At all times, a snapshot of the abstract region can be taken, returning a static enumeration of nodes. Hence, abstract regions abstract over one type of events: the fluctuating membership of nodes to abstract regions.

The data sharing operator allows variables, represented as key/value pairs, to be shared among nodes in the region. Data sharing is achieved using a tuple space-like programming model [Gel85]. The reduction operator takes a shared variable key and an associative operator (such as `sum`, `max`, or `min`) and reduces the shared variable across nodes in the region, storing the result in a shared variable. However, region membership events do not cause the resulting shared variable to be updated. The programmer must manually reapply the reduction to obtain up to date results.

SpatialViews [NKSI05] is not targeted at a pure resource-constrained sensor network environment, but is rather a macroprogramming extension to Java designed for programming mobile devices connected through a wireless ad hoc network. Under the hood, it uses the Smart Messages platform discussed earlier. SpatialViews' main programming abstractions are spatial views. A spatial view allows to describe physical spaces (such as circles, rectangles, etc.) and desired services within such spaces, hosted on and shared by mobile or stationary devices. Spaces can be combined using operators such as union, intersection, etc. Spatial views are instantiated by an iterator that discovers nodes that provide the services and reside in the space. When iterating, this iterator will send mobile code to the nodes in the space in which their services are invoked. Recursive calls from within an iterator are not allowed, but iterators can be nested. The iteration procedure may be limited by a time constraint which represents a time budget that once expired will lead to the termination of the iteration procedure (there could be spatial views where iteration is infinite when nodes keep joining the network or keep moving around), but cannot force a rollback of the procedure on nodes that did not respond within the time budget. The spatial view is thus the collection of nodes that

provide the specific services and are confined to a space-time region defined by a spatial view and an iterator. As time changes, the same node may be visited multiple times by the same iterator. If a node occupies multiple locations due to its mobility, it may also represent different nodes at the same time, depending on the modeling granularity of both space and time.

The SpatialViews compiler can be configured to generate different iteration strategies for the iterators in a SpatialViews program. The simplest strategy is simple iteration, where the currently reachable nodes are sequentially visited taking time and space constraints into account. More complex strategies allow flooding the network with the iteration procedures that are executed in parallel and of which the results are afterwards propagated back to the node initiating the iteration. The drawback of both approaches is that they may visit too many closely located nodes in close succession that, because of this, produce identical sensor readings. This is a waste of resources. Therefore, as a last strategy, SpatialViews can be configured to use geographic iteration. With geographic iteration, the target space is divided in cells recursively into a quadtree of smaller child spaces. The nodes are evenly distributed over the leaf cells of the quadtree. The iterator can then iterate over these leaf cells.

SpatialViews offers best-effort semantics for executing iterator code and propagating back results. It offers no fault tolerance constructs and assumes that sensor nodes are densely enough populated to be able to drop results. Therefore, iterating can only happen within a time budget specified up front: the asynchronous reception of results that are received with an arbitrary delay is not supported. Its synchronous programming model integrates poorly with the asynchronous, event-driven nature of mobile ad hoc networking applications.

3.4.3 Network-Centric Programming

In network-centric abstractions, a sensor network is treated as a whole and is regarded as a single abstract machine. Most (but not all) network-centric abstractions fall into the category of query processors [MFHH05, BGS01, YG02, LLS+04, SJS00]. Query processors use a traditional database abstraction to represent sensed data in the sensor network. The programmer queries these data using a declarative query language, typically a variant of SQL. Query processors strive to process such queries as energy-efficiently as possible. Typical strategies include minimizing expensive communication by applying aggregation and filtering operations inside the sensor network. This is somewhat similar to some content-based publish/subscribe systems or event processing systems that allow complex queries to be specified over observed events in a distributed system. Since these systems' expressiveness is limited to data gathering queries, we will not discuss all of them here. We will limit our survey on query processors to **Fjords**. Fjords (Framework in Java for Operators on Remote Data Streams) [MF02] is an object-relational query processor based on the Telegraph [CCD+03] dataflow processing system, implemented in Java. This underlying dataflow architecture allows Fjords to treat sensor data as continuous, never ending streams. Because streams are infinite, query operators (such as select, project, join, etc.) can never compute over an entire streaming relation: i.e. they cannot be blocking. Fjords modifies traditional operators into dataflow variants such that they deliver results incrementally, processing streaming values one at a time or in small blocks. These operators can be both pull-based and push-based. It is also possible to define aggregate operators, like count and average, which output results periodically; whenever a value arrives from the stream, the aggregate is automatically updated, and its revised value is forwarded to the user.

If traditional (i.e. blocking) aggregates, sorts, or joins must be used, Fjords allows, similarly to other query processors discussed above, that these operators specify a subset of the stream which they operate over. This subset is typically defined by upper and lower time bounds or by a sample count. Defining such a subset effectively converts an infinite stream into a regular relation which can be used in any database operator. Operators buffer the values they still need to process or need to send to the next operator in queues. This allows that the dataflow architecture is resilient to temporary network failures. Each operator has a set of input queues, and a set of output queues. Each operator reads values in any order it chooses from its input queues and outputs any number of values to some or all of its output queues.

When a new query is deployed, a query plan is constructed consisting of the internal representations of the operators that constitute the query. The code that implements these operators is dynamically sent to the sensor nodes on which they must be executed, which is there executed in a dedicated runtime. However, Fjords are currently non-adaptive; that is, they do not modify the query plan and the deployed query in the face of sensor delays or intermittent failures. To allow the efficient sharing of deployed operators among different queries concurrently, Fjords disallows operators to work on the history of a stream (i.e. its old tuples).

A second category of network-centric abstractions are macroprogramming languages. These languages are usually based on the functional programming paradigm and offer distributed version of lazy operations such as map, filter and fold that can be applied to sensor values.

Regiment [NMW07] is a macroprogramming language that compiles a network-wide dataflow representation of the program into a node-level, event-driven program. Regiment focuses on sensor network applications where spatiotemporal properties are an important part of sensor information. These include nodes' locations and their topological relationships, as well as time-varying sensor data and computational state.

In Regiment, the programmer views the network as a set of spatially-distributed, time-varying signals, each representing either the state of an individual sensor node (e.g., sensor readings or the results of local computation) or an aggregate taken across regions of sensor nodes. These aggregate signals are created using built-in primitives, such as map, filter and fold. This allows to sum all sensor values in a region as follows:

```
sumsignal = rfold((+), 0, inputregion)
```

Each time the contents of `inputregion` change, the call to `rfold` will be reevaluated, resulting in a new value for `sumsignal`. The second argument passed to `rfold` is the initial value for the resulting signal. It is important to note that it is unspecified how many times `rfold` will be invoked, so one must take care with side-effects.

Regions may be defined in terms of geographic area, network topology, or functional capability (e.g. all of the climate sensors within a given area). They are defined using primitives similar to the previously discussed group-centric systems (see section 3.4.2), such as Hoods and Abstract Regions. Regiment abstracts away the details of sensor data acquisition, storage, and communication from the programmer, instead permitting the compiler to map global operations on signals and regions onto local structures within the network. It is important to note that membership in a region may vary with time; for example, the region defined as "temperature readings from nodes where temperature is above a threshold" will consist of values from a varying set of nodes over time. The collection of signals that participate in a region can also vary due to node or communication failures or the addition of nodes in the network.

Sensor nodes are represented as primitive data types that basically are a key-value-

based associative data structures where the programmer can request any property or sensor value of a node by passing it a key. The return value of such an operation is naturally a time-varying signal. A severe restriction is that – to avoid high communication overheads – Regiment enforces the rule that time-varying signals are only accessible on the physical sensor node where they are hosted. To communicate time-varying values, the programmer must first project the sensor readings of interest from the node before attempting network communication.

Flask [MMW08] is a macroprogramming language embedded in Haskell that applies functional programming to collections of sensor nodes and additionally offers a dataflow programming library to wire together node-level application fragments. It is implemented on top of TinyOS and Flask programs are compiled into distributed nesC programs. Flask has ties with the research on Regiment discussed above, but by having two separate idioms for node-level and network-level code, removes the restriction that time-varying sensor values can only be used locally in node-level code.

Node-level code can be written in a language called Red, which is syntactically equivalent to Haskell but disallows closures and recursive functions and data types, thereby eliminating arbitrary allocation. Red offers language interoperability with nesC: nesC functions can be called from within the Red runtime and quasiquoting allows nesC code to be used anywhere in Haskell source files. This is useful to access low level sensor primitives. Flask provides abstractions for wrapping event-driven sensor code written in nesC into time-varying signals.

Haskell serves as Flask's meta language, and its full power is available for expressing how dataflow graphs can be constructed using Yampa-style combinators (discussed earlier in section 3.3.1). The dataflow library differs from Yampa in that it provides first-class signals rather than arrowized functional reactive programming's signal functions. Unlike behaviors in functional reactive programming and Regiment's signals (discussed above), Flasks signals are always discrete event streams and do not automatically trigger reevaluation of dependent code.

Flask cannot make the assumption that all signals are synchronously driven by a global and provides dedicated dataflow combinators to wire together stateful computations while preventing race conditions. Still, care must be taken when using stateful nesC code into a dataflow program, as race conditions are not automatically detected nor prevented by Flask.

Flask constructs a distributed dataflow graph of sensor processing nodes, but in this case at compile-time instead of run-time using the Haskell meta-language. To cope with a dynamically changing network topology in mobile ad hoc networks we require the graph's nodes to be deployed at runtime instead of at compile-time.

3.4.4 Evaluation

In this section, we give an overview of the surveyed sensor network programming technologies with respect to the programming model requirements that we put forward in section 2.3.4. These results are summarized in table 3.4. Desirable properties for mobile RFID-enabled applications are again shown in italic font.

Time Decoupling Just like in mobile ad hoc network applications, sensor network applications must deal with volatile connections. Therefore, almost all surveyed systems employ a time-decoupled communication style that is resilient to network failures (exceptions are Smart Messages and SpatialViews that assume a densely populated

network where it is acceptable to only reach part of the nearby nodes within a certain time budget). Examples are asynchronous messaging, code migration, tuple space abstractions, caching required data broadcasted by peers, etc.

Network-centric approaches have dedicated mechanisms to efficiently route information from individual nodes to the base station and to orchestrate the communication between individual nodes. The programmer is not exposed to their communication infrastructure (except for time-varying values in Regiment, which must be manually sampled and communicated).

	Space decoupling	**Arity decoupling**	**Time decoupling**	**Event representation**	**Infrastructureless**	**Control flow management**
Object State Model	*Yes*	No	*Yes*	nesC values	*Yes*	Finite state machines with concurrent events
TinyLIME	*Yes*	No	*Yes*	Tuples	*Yes*	No
Smart Messages	*Yes*	*Yes*	No	Tuples	*Yes*	Blocking
Hood	*Yes*	*Yes*	*Yes*	None	*Yes*	No
Abstract Regions	*Yes*	*Yes*	*Yes*	None	*Yes*	No
SpatialViews	*Yes*	*Yes*	No	None	*Yes*	Blocking
Fjords	No	*Yes*	*Yes*	*Object streams*	No	Dynamic queries, *dataflow operators*
Regiment	*Yes*	*Yes*	*Yes*	*Streams*	*Yes*	Implicit dataflow reevaluation (local only)
Flask	No	No	*Yes*	*Streams*	*Yes*	*Explicit dataflow*

Table 3.4: Survey of sensor network programming technology.

Space Decoupling Other than time-decoupled communication, space decoupling is considered desirable as well [SG08]. From the node-centric approaches, TinyLIME provides tuple spaces as a simple decoupling mechanism. The rest uses broadcasting to dynamically discover nodes in the sensor network.

Group-based abstractions allow to declaratively specify which nodes belong to a certain group. In some cases, this is purely used as a decoupling mechanism: unlike statically binding to nodes, it allows to enumerate the available set of nodes when they need to be accessed. For example, in Smart Messages, Hood and Abstract Regions content-based filters on data hosted by these nodes determine if these nodes belong to the group. Obviously, these data can contain physical properties of the node, such as its

location. SpatialViews provides dedicated abstractions to compose groups that reflect nodes in a specific region of the network.

Regiment reifies group-centric constructs on a network-centric level.

Arity Decoupling Node-centric approaches only support basic directed and broadcast-based communication primitives (e.g. as offered by nesC).

Other than space decoupling, group-centric approaches allow to easily spawn redundant operations (to deal with failing sensors or low quality of sensed values) and to address a group of nodes (that is automatically updated behind the scenes) by their *role*. This makes it for example possible to maintain a group of all temperature sensors and broadcast them a request for the current temperature. Such group operations obviously return a group of results, which in most cases, because of the asynchronous nature of the communication layer, have to be processed asynchronously (i.e. they do not arrive at the same time, maybe even with a large time-span between them). In most systems, the programmer must manually poll for more incoming results before applying an aggregation operator on the results. Exceptions are the network-centric approaches Regiment and Flask, which allow to represent the incoming results as an event stream or time-varying value, and use dataflow techniques to process them.

Event Representation How events are represented is to a large extent determined by the level of abstraction that is offered. Node-level approaches usually signal events by sending asynchronous messages. The Object State Model uses normal nesC values in predicates to determine to which state to advance using finite state automata. TinyLIME is based on tuple spaces and Smart Messages uses tuples as well to be able to declaratively specify groups of nodes. Other group-centric approaches attempt to abstract over events by making sure that the runtime keeps the specified groups causally connected to the physical sensor nodes by caching sensor values belonging to a group. Since they provide no application-level event handling framework, the programmer must manually poll for changes to these cached values. Macroprogramming abstractions such as Fjords, Regiment and Flask use event streams because they offer limited dataflow programming support.

Control Flow Management Because of the asynchrony that is prevalent in most sensor network operating systems (TinyOS), sensor network software is inherently event-driven. This leads to the same problems we have discussed before in chapter 2 section 2.3.3 and that are observed by Welsh and Mainland [WM04] as well:

> "The TinyOS concurrency model requires each concurrent "execution context" to be implemented manually by the programmer as a state machine, with execution driven by the sequence of commands and events invoked on each software component. If an application is performing multiple concurrent tasks, these operations must be carefully interleaved. In addition, each split-phase operation requires that the application code be broken across multiple disjoint segments of code. The programmer must manually maintain continuation state across each split-phase operation, adding significant complexity to the code. While the logical program may be quite simple, the lack of blocking operations in TinyOS requires that the application be broken into multiple tasks and event handlers."

This calls for dedicated event-driven abstractions that prevent structuring the entire application around (possibly concurrent) callbacks. This is achieved by bringing the global sensor network application to a higher level by means of macroprogramming techniques. Regiment and Flask, for example, offer dataflow programming techniques to alleviate these issues. Still, Regiment's time-varying values have to be manually sampled such that these sampled values can be sent to other nodes: there is no automatic way to notify other nodes of a value change. Flask, on the other hand, does allow events to be communicated using the dataflow framework, but the distributed dataflow graph is statically constructed at compile time and cannot be rearranged to deal with changes in the sensor network (such as node failures or the deployment of additional nodes).

Infrastructureless Operation Most approaches only require sensor nodes and do not rely on additional infrastructure. The only exceptions are the database abstractions, which require a base station from which sensor database queries are disseminated and to which query results are routed back.

3.5 Conclusion

In this section, we revisit our problem statement and integrate the different requirements for mobile RFID-enabled applications with interesting properties from the surveyed state of the art to tackle the now technically grounded problems.

3.5.1 Revisiting the Problem Statement

Now that we presented a survey on related programming technologies, we revisit our problem statement and technically ground it using the results of this survey. Technically, our problem statement is twofold:

- **There are no suitable programming abstractions for mobile RFID-enabled applications.** These applications must be manually implemented directly on top of the hardware level.
- Dealing with sensor technology such as RFID results in highly event-driven applications. Distributed event-driven systems focus on the decoupling requirements, but suffer from inversion of control. Dataflow programming technology on the other hand allows reacting to events without suffering from inversion of control, but does not take the decoupling required for mobile ad hoc networking applications into account. In short, **there exists no distributed event-driven programming model that both satisfies the decoupling requirements and the requirements that allow to declaratively deal with events without suffering from inversion of control**.

3.5.2 Towards Ambient-Oriented Programming for Mobile RFID-Enabled Applications

For mobile RFID-enabled applications, the state of the art provides no programming abstractions tailored towards general, decentralized mobile RFID-enabled applications, but instead relies on application-specific middleware. We repeat our programming

model requirements listed earlier in section 2.3.1. An instantiation of such a programming model is the topic of chapter 5 in this dissertation.

Addressing physical objects. RFID communication is based on broadcasting a signal. However, to be able to associate a software entity with one particular physical object, it is necessary to address a single designated physical object and maintain a (conceptual) connection with the object.

Storing application-specific data on RFID tags. In the spirit of mobile ad hoc network applications, we assume as little infrastructure as possible to implement mobile RFID-enabled applications. Hence, mobile RFID-enabled applications should be able to work without relying on a backend database and therefore it should be possible to store the application data in the writable memory of the RFID tags themselves [RK09, FL05, PS11, MTCS07].

Reactivity to appearing and disappearing objects. It is necessary to observe the connection, reconnection and disconnection of RFID tags to keep the mobile RFID-enabled application synchronized with its physical environment. Differentiating between connection and reconnection is important to preserve the identity of physical objects in the application. Furthermore, it should be possible to react upon these events from within the application. Notification of these events should be possible in a mobile ad hoc network (i.e. without assuming a fixed infrastructure).

Asynchronous communication. To hide latency and keep applications responsive in the face of intermittent connections, communication with RFID-tagged physical objects should happen asynchronously. Blocking communication will freeze the application as soon as one tag is unreachable.

Fault-tolerant communication. Treating communication failures as the rule instead of the exception allows applications to deal with temporary unavailability of the RFID-tagged physical objects and makes them resilient to failures. For example, read/write operations frequently fail due to hardware phenomena. Handling all these failures individually or each time they occur is not necessary and significantly complicates the development of mobile RFID-enabled applications.

Data consistency and security. Different mobile applications might concurrently read and - more importantly - write data to a number of tagged objects all within their proximity. This can lead to data races that have to be prevented. Similarly, in some scenarios data stored on RFID tags may not be read or modified by unauthorized users.

Support for mobile ad hoc networks. For conceiving mobile RFID-enabled applications, it should be possible to embed RFID technology in mobile devices that communicate via mobile ad hoc network technology. Hence, they should not rely on centralized infrastructure for their operation.

Such a programming model is inherently event driven because communication must happen asynchronously and the set of reachable RFID tags from the viewpoint of a single device is in constant flux. To circumvent the software engineering issues with a classic event-based system, we propose to devise a dataflow programming model for mobile RFID-enabled applications, as discussed below.

3.5.3 Towards a Publish/Subscribe-Style Interaction for Dataflow Programs

Dataflow programming technology allows functional composition of event-driven programs, while in programs structured around callbacks, control flow is inverted, evaluation can happen in unexpected orders, and in many cases must be coordinated using globally visible variables, making functional composition a lot harder. On the other hand, maintaining a dataflow graph tightly couples dependent reactive code such that it becomes impossible to guarantee progress in a mobile ad hoc network where volatile connections can prevent events to be immediately delivered.

Mobile publish/subscribe technology however, offers a very loosely coupled way of binding event producers to event consumers (by different kinds of subscription mechanisms and asynchronous event propagation). Hence, we propose to distribute dataflow programs by representing dataflow dependencies as event subscriptions in an underlying publish/subscribe infrastructure. This means that dataflow dependencies can be dynamically filled in as new event producers appear in the mobile ad hoc network, or replaced as event producers leave the network and other ones join. Using subscriptions, the knowledge that event producers and consumers must have about each other can be reduced and factored out to the bare minimum required to subscribe to interesting events. This way, subscriptions may be dynamically bound to a multitude of event producers. We propose to use group-centric techniques as found in sensor networks to asynchronously aggregate time-varying values published by such a group of event producers into a single time-varying value, and additional dataflow operators to perform other processing operators than aggregation (i.e. mapping over results). Since remotely and asynchronously signaled events arrive in an unpredictable, fluid fashion, the model must incorporate programming support to determine which events are used as updates to the result group (similar to sensor network query processors) and which (late-arriving) events are not interesting anymore and can be discarded. We call this mechanism *time sampling*.

To be as general as possible with respect to the representation of events in the programming model (e.g. for representing data structures in an RFID tag's memory), we aim for an object-oriented data type, offering classic benefits such as encapsulation, extensibility, polymorphism, and an easy mapping to real-world concepts.

All this together is what we call *ambient-oriented dataflow* programming. An ambient-oriented dataflow language with group-centric primitives is the topic of chapter 6 of this dissertation.

3.5.4 Towards a Network-Centric Ambient-Oriented Dataflow Language

An issue with all distributed event-based programming models is that reactive code (whether its execution is driven by callbacks or less traditional mechanisms such as dataflow) is triggered at arbitrary points in time by being notified of external events, that arrive unpredictably. This obscures the control flow of an application, as the textual, sequential representation of the program does not reflect the execution order anymore. Network-centric programming systems in sensor networks provide an additional layer of abstraction that allows specifying the global distributed program behavior explicitly and on a higher level. Query processors compile queries into plans that consist of node-level code that implement the global behavior. Flask allows the global program behavior to be specified declaratively in Haskell, that is used as a meta-language to

construct a dataflow program. Underneath, the global program is still driven by node-level events, but the network-centric abstraction takes care of filtering, communicating and aggregating events.

Similarly, for tackling the complex control flow of larger mobile RFID-enabled applications, we propose a dataflow meta-language to construct the global dataflow behavior of distributed applications. Naturally, it should take the requirements for mobile RFID-enabled applications into account. Therefore, it should be resilient to volatile connections, impose a very loose coupling among nodes in the distributed dataflow graph and work entirely decentralized (with as an exception the node launching the program, but only during initialization). This requires making sure that events are not lost in the face of intermittent connectivity, dynamically binding dataflow nodes to physical nodes at runtime and allowing other physical nodes to replace non-responding nodes in the graph.

Furthermore, it should be integrated with the programming model described in the section above. This means it must have network-centric representations for the group-centric communication primitives described in the section above.

Chapter 7 of this dissertation takes ambient-oriented dataflow to the network level and discusses a visual, network-centric dataflow meta-language called AmbientTalk/R^V.

3.5.5 Summary

Viewed from more abstract level, what we are exploring in this dissertation is how the control flow can be made less explicit in favor of an explicit data flow and how the macroscopic view of a distributed program can be shifted from a node-centric view to a network-centric view. These shifts turn the resulting programming models increasingly domain-specific towards mobile RFID-enabled applications. The table below gives an overview.

	Control flow	**Data flow**	**View**
Chapter 4 and 5 (ambient-oriented programming)	explicit (callbacks)	implicit	node-centric
Chapter 6 (node-centric ambient-oriented dataflow programming)	implicit (reactive)	implicit	node-centric
Chapter 7 (network-centric ambient-oriented dataflow programming)	implicit	explicit (dataflow graph)	network-centric

Table 3.5: Breakdown of programming models in chapters.

In chapters 4 and 5 we start out with a programming model in which the control flow has to be very explicitly coordinated through callbacks, but the data flow is very implicit because of the event-driven execution model. We only consider a node-level view on applications. Subsequently, in chapter 6, we turn the control flow implicit by adopting a reactive programming style in which events are processed without relying on callbacks, but are instead used by the interpreter to drive the execution of reactive expressions. The data flow is managed by the interpreter as well and remains implicit. After that, in chapter 7, we introduce the network-centric visual dataflow

language AmbientTalk/R^V, that turns the data flow of the application very explicit by using the dataflow graph itself as the program representation. This dataflow program is a network-centric program that is distributed in loosely-coupled node-centric code fragments that are executed by events that they communicate to one another.

Chapter 4

Ambient-Oriented Programming with AmbientTalk/2

The work described in this dissertation builds upon both the concepts of ambient-oriented programming (described in chapter 2) and the technical foundation laid by the AmbientTalk language. Therefore, before presenting our work in the later chapters of this dissertation, in this chapter we first give an overview of the second incarnation of the AmbientTalk language: AmbientTalk/2. We start this chapter by discussing the features of the language that are important for the implementation sections in the remaining part of this dissertation. Subsequently, in section 4.2 we discuss how the event-driven execution of AmbientTalk programs causes inversion of control. Finally, section 4.3 concludes this chapter.

4.1 AmbientTalk/2

In chapter 2 section 2.2 , we gave an overview of the hardware characteristics of mobile ad hoc networks and how they are tackled by the ambient-oriented programming paradigm. In this section, we give an overview of the concrete ambient-oriented programming language in which we carried out our work: AmbientTalk/2. Although AmbientTalk/2 has a predecessor (the original AmbientTalk introduced by Dedecker et al. [DVM+06]) that spawned many of the original ideas, we will only review AmbientTalk/2, which we will henceforth name AmbientTalk interchangeably.

This section gives an overview of work that is carried out in the past by other researchers. This overview serves three purposes:

- Discuss the computational model of AmbientTalk in which we embedded the different constructs and artifacts described in the remainder of this dissertation.
- Identify where AmbientTalk technically exemplifies the problems that we are tackling in this dissertation.
- Explain the features of AmbientTalk that are used throughout this dissertation in examples or transcripts of certain implementations.

AmbientTalk offers an actor-based, event-driven model of computation that aligns well with the event-driven nature of distributed computations. Next to explaining this event-driven computation model, we focus on AmbientTalk's features for orchestrating service discovery and composition in mobile ad hoc networks. We also describe how the language enables an ambient-oriented programming style in order to treat network partitions as a default mode of operation. We describe AmbientTalk's object model, metaprogramming and reflection facilities, concurrency model and distributed communication model. We include an extension called ambient references [VDM+06, Van08] that is used in the work presented in this dissertation. Since these matters were extensively discussed before in Van Cutsem's PhD dissertation [Van08], the text in this section is heavily based on that dissertation's section on AmbientTalk/2.

4.1.1 Object-Oriented Programming in AmbientTalk

AmbientTalk remains, first and foremost, a language to compose objects (services) across a mobile ad hoc network. Despite the domain-specific nature of its abstractions for distributed programming, AmbientTalk remains a full-fledged object-oriented programming language.

AmbientTalk is a dynamically typed, object-based language. Computation is expressed in terms of objects sending messages to one another. Objects are not instantiated from classes. Rather, they are either created *ex-nihilo* or by cloning and adapting existing objects. The following code illustrates standard object-oriented programming in AmbientTalk.

```
1  def Point := object: {
2    def x := 0;
3    def y := 0;
4
5    def init(newx, newy) {
6      x := newx;
7      y := newy;
8    };
9
10   def +(other) {
11     self.new(x+other.x, y+other.y)
12   };
13
14   def distanceToOrigin() {
15     (x*x + y*y).sqrt();
16   };
17 };
18
19 def origin := Point.new(0, 0);
```

In the above code snippet, a prototypical `Point` object is created ex-nihilo and bound to the variable `Point`. AmbientTalk objects consist of fields and methods, although methods can subsume fields because AmbientTalk unifies field access with nullary method application[1].

Instantiating an object is done by sending it the message `new`, which creates a shallow copy of that object and initializes the copy using its `init` method, which plays the role attributed to "constructors" in class-based languages. Every AmbientTalk object understands this `new` message. Any arguments passed to `new` are passed on to the copy's

[1] This property allows clients of an object to abstract over the fact whether data is stored in a field or calculated by means of a method. This is called the *uniform access principle [Mey88]*.

`init` method such that the copy can be reinitialized with new values. AmbientTalk's object instantiation protocol closely corresponds to class instantiation in class-based languages, except that the new object is a clone of an existing object, rather than an empty object allocated from a class.

AmbientTalk supports both traditional canonical syntax (e.g. `dict.put(key, value)`) as well as keyworded syntax (e.g. `dict.at: key put: value`) for method definitions, message sends and function invocations. As a convention, keyworded syntax is used for control structures (e.g. `while:do:`) or language constructs (e.g. `object:`). The canonical syntax is used for expressing application-level behavior.

By convention, when an object receives a message which it does not understand, it delegates the message to the object sitting in to its slot named `super` (which is implicitly defined on every object, except the `nil` value object). The object stored in the `super` slot is called the *parent object* of the object storing it. Creating an object with an existing parent object happens as follows:

```
1  def SpatialPoint := extend: Point with: {
2    def z := 0;
3
4    def init(newx, newy, newz) {
5      super.init(newx, newy);
6      z := newz;
7    };
8    // ...
9  };
10
11 def sPoint := SpatialPoint.new(20, 10, 30);
```

The semantics for delegating messages between objects follows that of Self [UCCH91] and Act1 [Lie86]: a delegated message is a message that is forwarded to another object, but for which the `self` pseudo-variable remains bound to the delegating object. Hence, AmbientTalk supports object-based (single) inheritance. The `super` slot is assignable, such that the parent of an object may change. This enables dynamic inheritance which is useful for implementing objects with state-dependent behavior [UCCH91].

The example given above defines a `SpatialPoint` object with a `Point` as parent object (`SpatialPoint` is said to *extend* `Point`). In the above example, `SpatialPoint` and `Point` remain separate objects in their own right. The relationship between a child and a parent object defined by `extend:with:` implies that the child's `super` field is initialized to the parent object and that when a child is cloned, the clone's `super` field is bound to a clone of the parent object. Hence, when a `SpatialPoint` is cloned, the clone has its own `Point` parent object with its own copies of the `x` and `y` fields. Thus, `extend:with:` is the object-based equivalent of class-based inheritance.

Block Closures

AmbientTalk provides support for block closures reminiscent of those in Self and Smalltalk [GR83]. A block closure is an anonymous function object that encapsulates a piece of code and the bindings of lexically free variables and `self`. Block closures are constructed by means of the syntax `{ |args| body }`, where the arguments can be omitted if the block takes no arguments. The code excerpt below illustrates a typical use of blocks to map a function over a table of numbers[2]:

[2] Tables are arrays of which the elements are indexed starting from index 1, not 0. The terminology stems from one of AmbientTalk's influential languages, Pico [DDD05]

```
[1,2,3].map: { |i| i + 1 }
// result: [2,3,4]
```

The following code excerpt shows another typical usage of blocks to remove all elements from a collection that fail to satisfy a predicate:

```
1  def from: collection retain: predicate {
2    result := clone: collection; // shallow copy
3    collection.each: { |elt|
4      predicate(elt).ifFalse: {
5        result.remove(elt)
6      };
7    };
8    result;
9  };
10
11 from: [1,-2,3] retain: { |e| e > 0 }
12 // result: [1,3]
```

Note that block closures can be applied using a familiar canonical function call syntax. Alternatively, they may be treated as objects: a block closure is an object with an `apply` method. In the above example, the call `predicate(elt)` can be equivalently expressed as `predicate.apply([elt])`. Block closures are frequently used in AmbientTalk to represent delayed computations, e.g. for implementing control structures.

Scoping, Nesting and Encapsulation

This section describes AmbientTalk's semantics for the resolution of names (scoping), how nested objects behave, and how scoping can be used to encapsulate an object's state.

Lexical versus object scope AmbientTalk is a lexically scoped language, meaning that free variables in a function or method are looked up in their environment of definition. However, AmbientTalk is also an object-based language with delegation, which introduces a second scope in which to resolve names: the object scope. The object scope of an object is the set of all names defined in the object plus the object scope of its parent object (the object referenced by its `super` field). The rules for distinguishing which scope to use when resolving a name are straightforward:

- An unqualified identifier (e.g. `x`) is resolved in the lexical scope.
- A qualified identifier (e.g. `o.x`), is resolved in the receiver's object scope.

These rules have a large effect on programs: lexical variable access can be statically determined, while qualified access is subject to late binding (enabling object-oriented polymorphism). The interplay between object inheritance and lexical scoping is particularly subtle. Consider the following example:

```
1  def obj := object: {
2    def x := 0;
3    def staticAccess() { x };
4    def dynamicAccess() { self.x };
5  };
```

In the code above, `obj` defines two accessors for its `x` field. The first accessor performs an unqualified access and hence looks up `x` in the lexical scope. The second accessor

performs a self-send, looking up `x` in `obj`'s object scope. Both will access the same field. The difference between both only becomes apparent in the context of object-based delegation. Consider the following code:

```
def child := extend: obj with: {
  def x := 42;
};
```

Invoking `child.dynamicAccess()` yields `42`, because `self.x` is late-bound and starts the lookup in `child`. However, invoking `child.staticAccess()` returns `0`: the `x` identifier referred to within the `staticAccess` method is the lexically visible one, and no object can change its resolution. Hence, the resolution of `x` is not the same as that of `self.x`.

Nesting and encapsulation In AmbientTalk, objects may be arbitrarily nested within other objects, functions or methods. Because of lexical scoping rules, this enables nested objects to access the lexically visible state and behavior of enclosing objects. Nesting objects is crucial to achieve encapsulation because AmbientTalk has no notion of visibility modifiers for fields or methods. All fields and methods of an object are considered "public". Nevertheless, a field or method can be made "private" to a scope by means of lexical scoping. The following code shows the definition of an object inside the definition of a function.

```
1 def makeBankAccount(balance) {
2   object: {
3     def deposit(amount) {
4       balance := balance + amount;
5       "ok";
6     };
7   };
8 };
```

Because the bank account object encapsulates the `balance` variable in its private, lexical scope, it cannot be selected from within its object scope, i.e. `makeBankAccount(100).balance` would result in an exception indicating that the name `balance` could not be resolved.

Type Tags

AmbientTalk is a dynamically typed, classless, prototype-based language. This introduces the problem that objects cannot be easily *classified*. In statically typed languages, the static type of the variable holding an object is often used for these purposes. In class-based languages, the class naturally plays the role of classifier. Object classification is useful for a diverse number of reasons. For example, in an exception handler, it is often useful to specify the type of objects that the handler can catch. Section 4.1.4 discusses the use of classification for the purposes of service discovery.

To recover the ability of classification, AmbientTalk introduces *type tags*. A type tag is identified by name (i.e. it is a nominal type) and it can be a subtype of zero or more other type tags. Objects, in turn, can be tagged with zero or more type tags. Type tags are not associated with a set of methods and are not used for static type checking. They are perhaps best compared with empty Java interfaces, like the typical "marker" interfaces used in Java libraries to merely tag objects (prominent examples are `java.io.Serializable` and `java.lang.Cloneable`). The following code illustrates the use of type tags:

```
1 deftype IndexableT;
2 deftype EnumerableT;
3 deftype OrderedT;
4 deftype SortableT <: EnumerableT, OrderedT;
5
6 def Array := object: {
7   // ...
8 } taggedAs: [ IndexableT, SortableT ];
```

Objects can only be tagged with type tags when they are created (via `object: taggedAs:`), and their set of type tags remains constant, to make sure objects distributed across multiple devices remain consistent in this regard.

A primitive function allows the programmer to perform a type test on objects, e.g. `is: Array taggedAs: Enumerable`. The type test determines whether an object *or one of its parents* is tagged with the given type tag *or a subtype* of the type tag. This is very reminiscent of the behavior of the `instanceof` operator of Java.

Type tags are first-class objects. Thus, they can be parameter-passed as arguments, bound to variables, etc. However, type tags do not follow standard object identity semantics. Type tag equality is by their name rather than by their object identity.

4.1.2 Metaprogramming and Reflection in AmbientTalk

Apart from AmbientTalk's role as an ambient-oriented programming language, it also serves as a language laboratory to develop and explore novel language constructs. Therefore, following a long-standing tradition [McA95, BGL98, CBM+02], AmbientTalk is conceived as an extensible research artifact equipped with extensive meta-level and reflective programming abstractions. Since the work presented in this dissertation relies heavily on these facilities, we discuss them here.

First-class Messages

Similar to e.g. Smalltalk, in AmbientTalk, object-oriented messages and methods can be manipulated by the programmer as first-class citizens of the language. Consider the following methods defined on tables (AmbientTalk's arrays):

```
1 def eachSend: message {
2   self.each: { |rcvr| rcvr <+ message };
3 };
4 def mapSend: message {
5   self.map: { |rcvr| rcvr <+ message } ;
6 };
```

The expression `rcvr <+ msg` sends a first-class message `msg` to an object `rcvr` as if the message was literally invoked on the receiver in the source text. Hence, this expression provides the functionality of `perform:` in Smalltalk or `apply` in functional languages. The above methods, together with a literal syntax for messages, can be employed to express higher order messages [WD05] – messages taking other messages as their argument – as shown below.

```
observers.eachSend: <-statusUpdated(newStatus);
[4,5,6].mapSend: .+(2); // returns [6,7,8]
```

A literal message expression is denoted by a message send expression that lacks a receiver. AmbientTalk distinguishes between three kinds of literal messages: asynchronous ones (`<-m()`), synchronous ones (`.m()`) and delegated ones (`^m()`). The `<+`

operator is polymorphic and expresses either an asynchronous, a synchronous or a delegated message send based on the type of its message argument.

A first-class message can be queried for its selector (its name) and the actual arguments it carries. The message object can also be classified according to all type tags with which it was annotated using the `@` syntax. Hence, for a message `msg` constructed as `.m()@Type`, evaluating `is: msg taggedAs: Type` yields `true`.

Reflection

Computational reflection allows programs to reason about themselves [Smi84, Mae87]. AmbientTalk provides extensive support for reflection by means of a *mirror-based* architecture [BU04]. AmbientTalks metalevel architecture combines mirror-based reflection with *intercession* – the ability of programs to change the semantics of the programming language.

AmbientTalks mirror-based architecture has been inspired by that of Self [US87]. The following code excerpt shows how one may reflectively manipulate an instance of the `Point` object defined earlier in section 4.1.1[3].

```
1  def p := Point.new(2,3);
2
3  // retrieve a mirror by invoking reflect:
4  def mirrorOnP := (reflect: p);
5
6  // read the contents of a field via its mirror
7  mirrorOnP.grabField(`x).value; // 2
8
9  // reflectively invoke a field access
10 mirrorOnP.invokeField(p, `x); // 2
11
12 // retrieve a mirror on a method
13 mirrorOnP.grabMethod(`init); // <mirror on method:init>
14
15 // reflectively invoke a method
16 mirrorOnP.invoke(p, `distanceToOrigin, []);
17
18 // print all method names
19 mirrorOnP.listMethods.each: { |method| system.println(method.name) };
20
21 // add a z coordinate
22 mirrorOnP.defineField(`z, 0);
```

A mirror is a metaobject which is *causally connected* [Mae87] to the object it mirrors: if the object is changed by base-level code, the changes can be observed via the mirror. Conversely, changes applied explicitly to the object via the mirror modify the actual base level object. The above examples illustrate various forms of reflection. Using the terminology of Kiczales et al. [KR91], AmbientTalk mirrors support:

Introspection: the retrieval of fields and methods (cf. `grabField`, `grabMethod` and `listMethods`).

Invocation: the explicit invocation of methods (cf. `invoke`). The arguments passed to `invoke` are a receiver (any object), a selector (a symbol) and actual arguments (a table). The receiver parameter is the object to which `self` is bound during

[3] In AmbientTalk, a backquote character is used to quasi-quote an expression (cf. quasi-quoting in Scheme [Dyb09]). A quasi-quoted expression evaluates to an object representing the expression's abstract syntax tree. Quasi-quoting an identifier evaluates to a symbol.

method invocation. If this receiver is the same as the object being mirrored (`p` in the example), the reflective call expresses a standard method invocation. If the receiver object is a different object, the reflective call expresses explicit delegation.

Self-modification: the addition of new fields and methods (cf. `defineField`).

Mirrors on objects are accessed by calling the `reflect:` function. The `reflect:` function in turn creates a mirror by calling a factory method, which can be replaced by metaprograms. Because a mirror on an object `obj` is retrieved via a separate mirror factory (via `reflect: obj`), rather than by querying the object itself (e.g. via `obj.reflect`), the association between objects and their metaobjects can be separated from base level concerns. Separating mirrors from their associated base level objects in this way is what makes the mirror architecture stratified.

Mirages: Mirror-based Intercession

In this section, we describe how an AmbientTalk programmer can provide his own definition for methods of AmbientTalk's metaobject protocol, such as e.g. the `invoke` operation used in the previously shown `Point` example. This form of reflections is called *intercession* by Kiczales et al. [KR91]: the ability of metaprograms to modify the behavior of objects. As a language laboratory, AmbientTalk relies heavily on intercession to develop new language constructs. For example, the implementation of "thing" objects denoting RFID-tagged objects discussed in the next section relies on intercession.

A *mirage* is a base-level object that is causally connected to a mirror object with a customized MOP. In order to clarify this, consider the archetypical example of intercepting and logging all methods invoked on an object. First, we define a prototype mirror object that encodes the logging behavior by overriding the default implementation of the `invoke` metalevel operation:

```
1 def LogMirror := extend: actor.defaultMirror with: {
2   def invoke(rcvr, selector, args) {
3     system.println("invoked "+selector+" on "+self.base);
4     super^invoke(rcvr, selector, args); // perform default behavior
5   };
6 };
```

To facilitate the development of mirror objects which require only small changes with respect to the default language semantics, AmbientTalk actors implement a prototypical mirror object named the `defaultMirror` which encapsulates AmbientTalks default metaobject protocol. The `defaultMirror` makes the native metaobject protocol implementation explicitly accessible while keeping it encapsulated behind the protocol's interface. The `LogMirror` leaves all metalevel operations intact save `invoke`.

A mirror can refer to the object with which it is causally connected by invoking `self.base`. The above mirror is but a prototype implementation: it is not yet causally connected to any object. A mirror object can only modify the interpreter when a mirage object is defined that is explicitly mirrored by that mirror object. The code excerpt below redefines the `Point` prototype from the previous section as a mirage, of which the meta-behavior is now defined by the `LogMirror`:

```
1 def Point := object: {
2   /* the original implementation */
3 } mirroredBy: LogMirror;
```

The `object:mirroredBy:` language construct associates a mirage base level object with its corresponding mirror metaobject. When the mirage is constructed, it becomes causally connected with its mirror. The latter then effectively becomes absorbed by the interpreter. For example, evaluating `Point.new(1,1).distanceToOrigin()` now triggers the custom `invoke` method defined by the `LogMirror`. The details on how the causal connection between a mirage and its associated mirror is constructed and maintained is described in more detail by Mostinckx et al. in [MVTT07].

First-class Abstract Syntax Trees and Environments

AmbientTalk treats syntax trees as first-class values. For example, the code of a method body obtained using the mirror interface described before in section 4.1.2, returns such a syntax tree. Alternatively, reading (i.e. parsing) source code is a reified operation that can be used to construct syntax trees as well. In fact, AmbientTalk reifies the read, eval and print operations of its interpreter, similar to the quoting and quasiquoting mechanism of languages such as Scheme [Dyb09]. This means that one can read any string and get the corresponding syntax tree for it, evaluate any syntax tree and get a value for it, and print any value and get a string representation of the value. Consider the example below:

```
read: "1+2";  // >> 1.+(2)
```

Syntax trees are regular AmbientTalk objects, represented as *symbols* (i.e. quoted expressions). Hence, the value `` `1.+(2) `` is an AmbientTalk symbol that can be further used in metaprogramming operations.

Once a syntax tree object is obtained, it can be evaluated. AmbientTalk goes one step further than Scheme by offering an `eval:in:` construct (as opposed to Scheme's `eval` primitive). This construct takes two arguments. The first argument is the syntax tree object to evaluate and the second object is the *environment* or *scope* in which the code represented by the syntax tree object will be evaluated. Hence, environments are first-class values as well in AmbientTalk. In fact, environments are represented as regular objects in AmbientTalk. Consider the code below:

```
1 def o := object: {
2   def x := 4
3   def sum(x, y) { x+y };
4 };
5
6 eval: `x in: o; // >> 4
7 eval: (read: "sum(5, x)") in: o;  // >> 9
```

The object `o` defined above also represents an environment with two variables defined: the field `x` and the function `sum`[4]. When evaluating expressions represented as syntax tree objects, these variables are dynamically bound to free variables in these expressions.

4.1.3 Concurrent Programming in AmbientTalk

In AmbientTalk, concurrency is spawned by creating actors: one AmbientTalk virtual machine may host multiple actors which run concurrently. AmbientTalk's concurrency model is based on the communicating event loops model of the E language

[4] Naturally, it also contains the variables `super` and `self`.

[MTS05a], which is itself an adaptation of the well-known actor model [Agh86]. The model combines actors and objects into a unified concurrency model. Unlike previous actor languages such as Act1 [Lie87], ABCL [YBS86] and Actalk [Bri88], actors are not represented as "active objects", but rather as *vats* (containers) of regular objects, shielding them from harmful concurrent modifications. Within the confines of one single vat, computation happens sequentially. Incoming messages from objects living in other vats are processed in a serial manner in order to ensure that no race conditions can occur on the internal state of the objects within the vat.

Each vat contais an *event loop*, which is a thread of execution that perpetually processes *events* from its *event queue* by invoking a corresponding *event handler*. Hence communication between two vats happens because their event loops exchange messages. In later chapters, we will use actor, vat and event loop interchangeably to denote the same concept, i.e. AmbientTalk's unit of concurrency. Communicating event loops enforce three fundamental concurrency control properties:

Property 1: serial execution *An event loop processes incoming events from its event queue one by one, i.e. in a strictly serial order.*

As a consequence, the handling of a single event happens in mutual exclusion with respect to other events. Hence, race conditions on an event handler's state caused by concurrent processing of events cannot occur.

Property 2: non-blocking communication *An event loop never suspends its execution to wait for another event loop to finish a computation. Rather, all communication between event loops occurs strictly by means of asynchronous event notifications.*

As a consequence of non-blocking communication, event loops can never deadlock one another. However, in order to guarantee progress, an event handler should not execute e.g. infinite `while` loops. Rather, long-lasting actions should be performed piecemeal by scheduling events recursively, such that an event loop always gets the chance to respond to other incoming events.

Property 3: Exclusive state access *Event handlers and their associated state belong to a single event loop. In other words, an event loop has exclusive access to its mutable state.*

As a consequence, two or more event loops never share synchronously accessible mutable state. Because event handlers are not shared between event loops, they never have to lock mutable state.

Event loop concurrency avoids deadlocks and certain race conditions by design. The nondeterminism of the system is confined to the order in which events are processed. In standard preemptive thread-based systems, the nondeterminism is more substantial because threads may be pre-empted upon each single instruction. In the following section, we describe how the abstract event loop model is incorporated into the AmbientTalk language.

AmbientTalk Actors

As mentioned earlier, AmbientTalk actors are event loops: the event queue is represented by an actor's message queue, events are represented as messages, event notifications as asynchronous message sends, and event handlers are represented as (the

methods of) regular objects. The actor's event loop thread perpetually takes a message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message. Messages are processed serially to avoid race conditions on the state of regular objects.

Each AmbientTalk object is said to be owned by exactly one actor. This ownership relation is established upon object creation and cannot change during the lifetime of the object. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor may communicate using standard, sequential message passing or using asynchronous message passing. AmbientTalk borrows from E the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). It is possible for objects owned by an actor to refer to objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [MTS05a]) and only allow asynchronous access to the referenced object. Synchronous access to an object via a far reference raises a runtime exception. Any messages sent via a far reference to an object are enqueued in the message queue of the actor owning the object and processed by the owner itself.

Figure 4.1 illustrates AmbientTalk actors as communicating event loops. The dashed lines represent an actor's event loop which perpetually takes messages from its message queue and synchronously executes the corresponding methods on its owned objects. The control flow of an actor's event loop never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

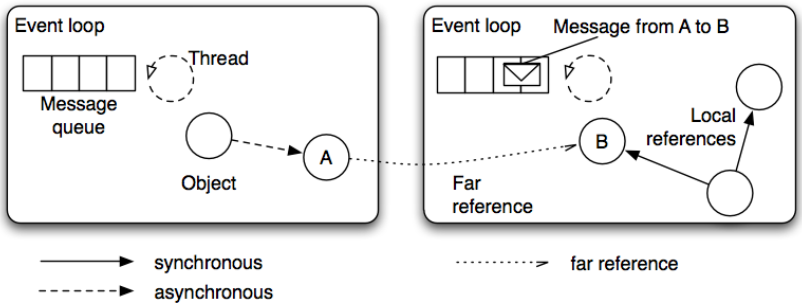

Figure 4.1: AmbientTalk actors as communicating event loops.

A far reference encapsulates a copy of the set of type tags with which its target object is tagged. This implies that a type test can be performed locally on a far reference to an object, which explains why the set of type tags of an object must remain constant: the far reference maintains a copy of that set on a potentially remote device.

Every AmbientTalk interpreter starts its execution with a single actor. An actor can spawn new actors by invoking `actor: { ... }`. When a new, empty, actor is created it evaluates the code passed to the `actor:` primitive in order to construct the first object it will own. The return value of the `actor:` primitive is a far reference to this object, thus allowing the creating actor to communicate with the new object owned by the created actor.

Message Passing Semantics

In AmbientTalk, asynchronous messages can be sent between objects owned by the same or by different actors. In the case where both sender and receiver are owned by the same actor, the message is simply added to the owner's message queue and parameters are passed *by reference*, exactly as is the case with synchronous message sending. For inter-actor message sends, where an object sends an asynchronous message via a far reference to an object owned by another actor, objects are parameter-passed *by far reference*: the parameter of the invoked method is bound to a far reference to the object. In either case, **messages are guaranteed to be delivered to an object in the same order as they were sent**. Consider the following example, assuming that the code on the left-hand and the right-hand side is executed in two different actors:

```
def arg := object: { ... };

obj<-m(arg);
```

```
def obj := object: {
  def m(par) { ... };
};
```

In the method body of `m`, `par` will be bound to a far reference to `arg`.

In some cases, the remotely invoked method may want to access its argument synchronously. To this end, AmbientTalk introduces the notion of an *isolate* object. Isolates are objects that are passed *by copy* over a far reference. This allows the recipient actor to operate on the copy synchronously, without additional inter-actor communication and without violating the exclusive state access property. When an isolate is copied during parameter-passing, all objects it directly refers to are recursively parameter-passed (according to their own semantics). The following code provides an example of an isolate:

```
1 def ComplexNumber := isolate: {
2   def re := 0;
3   def im := 0;
4   def init(r,i) { re := r; im := i; };
5   // ...
6 };
```

Isolates are regular objects, with two notable differences. First, as already remarked, they are passed by copy across far references. Second, isolates cannot use any free lexically visible names. Isolates are thus completely isolated pieces of code, without any implicit dependencies on the surrounding scope – hence their name. Isolates are best thought of as if they were lexically defined at the top-level. Isolates are disallowed access to their lexical scope because they are copied during parameter passing. The restriction avoids having to implicitly copy any lexically visible variables referred to by the isolate.

Futures

By default, an asynchronous message send does not return a meaningful value (i.e. it returns `nil`). However, this can make the processing of return values quite cumbersome, because objects owned by different actors must always explicitly signal their return value as an asynchronous reply message to the sender of the original message. Of course, this original sender must implement the corresponding method (i.e. a *callback* method) to process the return value.

In order to better reconcile return values with asynchronous message sends, AmbientTalk employs the notion of a *future* (also known as a *promise*). This is a frequently recurring language abstraction in concurrent programming languages [BGL98]. In

AmbientTalk, a future is a placeholder for the return value of an asynchronous message send. Once the return value is known, it "replaces" the future object; the future is said to be *resolved* with the value. The example shown below illustrates the creation of a future through an asynchronous message send.

```
def sumFuture := calculator<-add(x,y);
```

This makes the handling of return values syntactically identical to that of the familiar synchronous message send. However, we have yet to explain how objects can synchronize on the actual value represented by the future. In many programming languages, futures act as synchronization barriers while: if code tries to access the futures value before the future is resolved, the thread of control is suspended until the value has been computed. In a language with communicating event loops, however, such a semantics would violate the non-blocking communication property. It would imply that an event loop can suspend in a state other than when its event queue is empty. Hence, the event loop would become unresponsive to other events, and the entire system becomes prone to deadlock once again.

AmbientTalk avoids the wait by necessity semantics and instead employs the semantics first introduced by promises in E [MTS05a]. An actor can register its interest in the resolved value of the future by registering an observer – a closure, to be precise – that will be invoked later, when the future has become resolved. This observer serves as a callback or event handler that will be invoked when the future is resolved. The advantage over explicit asynchronous callback messages is that such a callback closure resides in and captures the lexical scope of the expression in which it is registered, as shown below:

```
1 def sumFuture := calculator<-add(x,y);
2
3 when: sumFuture becomes: { |sum|
4   system.println("result: " + sum);
5 };
```

The `when:becomes:` construct shown above takes a future and a block closure as its arguments, and registers the closure as an observer on the future. If the future is resolved, the closure is applied to the resolved value. Multiple observer closures may be registered on the same future. Note that the `when:becomes:` function itself returns immediately. The code specified in the block closure is always delayed, i.e. it is executed after the code following the call to `when:becomes:`, even if `sumFuture` is already resolved at the time the observer is registered. It is also guaranteed to be executed by the same actor that performed the asynchronous message send. Hence, the execution of the observer is always serialized with respect to other activities within the same actor.

The block closure passed to `when:becomes:` acts as an in-line event handler. It effectively enables the sender of an asynchronous message to synchronize on and process the result of that message in the scope where it was sent. Because block closures close over their lexical scope, all variables in scope at the time the message was sent are still available when handling the return value at a later point in time. The programmer does have to be aware of the fact that the values of the variables in scope may have changed since the time the message was sent, as other code may have run within the actor while the code was delayed.

Asynchronous Exception Handling A final aspect of synchronizing on a future is dealing with exceptions. AmbientTalk features a standard exception model that allows

objects to be raised and caught as exceptions. When an exception is raised inside an asynchronously invoked method, the exception propagates up to the level of the asynchronous invocation. At this point, the only available continuation is the future attached to the message. In order to signal the exception to the sender, the future is *ruined* with the exception. When a future is ruined, regular observers on the future are not triggered. However, a separate exception handler can be specified as follows:

```
1 def quotientFuture := calculator<-divide(x,y);
2
3 when: quotientFuture becomes: { |quotient|
4   system.println("result: " + quotient);
5 } catch: DivisionByZero using: { |e|
6   system.println("error: divided "+ x +" by zero");
7 };
```

In AmbientTalk, exception types are modeled using type tags. Hence, it is assumed that `DivisionByZero` is a type tag. Should `quotientFuture` become ruined with (a subtype of) `DivisionByZero`, the second block closure is applied to the exception. This enables the handling of asynchronously raised exceptions much in the same way as the well-known `try-catch` construct is used for regular exception handling.

The return value of a call to `when:becomes:catch:using:` is itself a future. The future is resolved with the return value of an observer block closure, or ruined if an exception occurs during the execution of that closure. In effect, the future that is returned by `when:becomes:` is dependent on the future on which it operates: resolving or ruining the latter eventually leads to resolving or ruining the former.

4.1.4 Distributed Programming in AmbientTalk

In this section, we proceed from concurrent to distributed programming in AmbientTalk. Actors can be distributed across a network, each hosted by different AmbientTalk interpreters. The major difference between single-machine and distributed programming is the possibility of partial failures, the phenomenon whereby remote objects may not respond to messages, due to either a network or a machine failure. The second important issue is service discovery, the goal of which is to acquire a first reference to a remote object.

Object	Unit of designation
Actor	Unit of concurrency
Interpreter	Unit of partial failure
Java Virtual Machine	Unit of termination

Table 4.1: Overview of AmbientTalk's units of operation.

Before continuing, a brief word on terminology. We have previously mentioned that an actor is said to *own* one or more objects. Likewise, an AmbientTalk interpreter is said to *host* one or more actors. Because AmbientTalk is currently implemented in Java, a single Java Virtual Machine can be said to host one or more AmbientTalk interpreters. Two objects are said to be *local* when they are owned by the same actor. Objects are considered *remote* when they are owned by different actors, even if those actors are hosted by the same interpreter. Within one interpreter there is no notion of partial failure: connections between actors within a single interpreter never fail.

Hence, interpreters are the unit of partial failure. Within one JVM, there is no notion of "crashes": either all interpreters within a single JVM are terminated, or none of them are. Hence, JVMs are the unit of termination. Table 4.1 summarizes AmbientTalk's units of operation.

Recall from section 4.1.3 that references between objects owned by different actors are always far references which only permit asynchronous access to their target object. Because objects residing on different devices (i.e. in distinct interpreters) are necessarily owned by different actors, **far references are the only kind of remote object references in AmbientTalk**. This ensures by design that all distributed communication is asynchronous, as required by the non-blocking communication characteristic of the ambient-oriented programming paradigm.

Far References and Partial Failures

AmbientTalk's far references are by default resilient to failures. When a failure occurs, a far reference becomes disconnected. However, a disconnected far reference buffers all messages sent to it. When the failure is restored at a later point in time (e.g. a network partition is healed), the far reference flushes all accumulated messages to the remote object in the same order as they were originally sent. Hence, messages sent to far references are never lost. Making far references resilient to failures by default is one of the key design decisions that make AmbientTalk's distribution model suitable for mobile ad hoc networks, because temporary network failures have no immediate impact on the application's control flow. This behavior is desirable in mobile networks because it can be expected that many partial failures are the result of temporary network partitions. However, perhaps a machine has crashed beyond recovery, or it has moved out of the wireless communication range and does not return. Such persistent failures also need to be dealt with. We postpone this discussion until the next section.

AmbientTalk allows to monitor connectivity with remote objects by means of two event handlers that are both illustrated in the following example. In the example, the GUI of an instant messenger application uses these event handlers to indicate whether a buddy in the user's buddy list is online or not.

```
1 // buddy is a far reference to an instant messaging peer
2 whenever: buddy disconnected: {
3   gui.markOffline(buddy);
4 };
5 whenever: buddy reconnected: {
6   gui.markOnline(buddy);
7 };
```

The event handlers are registered by applying primitive functions that both take a far reference and a nullary closure as their arguments. The closure is applied *whenever*[5] the interpreter detects the disconnection (respectively reconnection) of the object referred to by the far reference.

The return value of both failure event handlers is a subscription object with a single method, named `cancel`. When invoked, it cancels the registration of the event handler with the interpreter, such that the closure will no longer be triggered.

Ruining futures in response to persistent failures Unfortunately, it is impossible to distinguish network failures from device failures [WWWK96]] and even if it is a net-

[5] AmbientTalk also supports `when:disconnected:` and `when:reconnected:` event handlers that are only triggered *once*.

work failure, one cannot distinguish transient from permanent failures [Wal01]. The best the programmer can do is to choose a timeout period that will treat the failure as "permanent" as soon as it persists long enough. AmbientTalk allows the expression of such timeouts either at the level of far references or at the level of individual asynchronous messages. Timeouts at the level of eventual references are part of a leasing strategy, which are not used in the rest of this dissertation and which we will not discuss for the sake of brevity. Here, we focus on timeouts associated with individual asynchronous messages. If the future associated with a message annotated with a timeout is not resolved within the timeout period, the future is automatically ruined by the system with a `TimeoutException`. The following code shows how a timeout can be specified and dealt with:

```
1 when: buddy<-chat(aTextMessage)@Due(minutes(1)) becomes: { |ack|
2   // message received successfully
3 } catch: TimeoutException using: { |e|
4   // message timed out
5 };
```

In the above example, if the `chat` message is not replied to within 1 minute, the exception handler is triggered (i.e. asynchronously applied). The `@` syntax allows a message to be annotated with one or more type tags. The `Due` type tag can be used to associate a timeout period with a message.

When performing failure handling, one should always be aware of the fact that the message may still have been received by the remote party. It may even be possible that the receiver sends a reply after the timeout period has already expired. In this case, the future will silently ignore the return value, as it has been previously ruined with a `TimeoutException`.

Exporting Objects

Objects can acquire far references to objects or copies from isolates by means of parameter-passing or return values from inter-actor message sends. It remains to be explained how objects can acquire an *initial* far reference to an object by a remote actor. In order to make it possible for an object to be discovered by remote actors, the object must be explicitly *exported*. The same is true for isolate objects, but instead of a far reference, a copy is exported.

An object always has to be exported with a corresponding type tag. The type tag is used to classify what kind of service the object provides. It plays a role similar to the topic in traditional publish/subscribe architectures. The subtyping mechanism of type tags allows objects to be published in a hierarchical classification. It is assumed that all distributed peers attribute the same semantics to the names of type tags, and define the same type hierarchy. The following example shows how to export an object as an instant messenger chat peer:

```
1 deftype InstantMessenger;
2
3 def peer := object: {
4   def chat(textMessage) { ... };
5 };
6
7 export: peer as: InstantMessenger;
```

From the moment an object is exported, it is discoverable by objects owned by other actors by means of its associated type tag. The `export:as:` primitive returns a publica-

tion object `pub` which can be used to "unexport" the object by invoking `pub.cancel()`. An unexported object can no longer be discovered by remote objects. However, far references referring to the unexported object remain valid, so an unexported object can still be remotely referred to.

Service Discovery

The AmbientTalk interpreter has a built-in service discovery mechanism that enables the discovery of remote objects in a peer-to-peer manner. The mechanism is fully decentralized, no servers or other infrastructure are required. Objects that want to be notified by the interpreter of available remote objects do so via a discovery event handler:

```
def subscription := whenever: InstantMessenger discovered: { |messenger|
  buddyList.add(messenger);
};
```

A discovery event handler is registered by calling the `whenever:discovered:` primitive (or the `when:discovered:` variant) which takes as arguments a type tag and a unary closure. Whenever an actor is encountered in the network that exports a matching object, the closure is triggered on (i.e. asynchronously applied to) a far reference to the discovered object. Hence, the parameter of the block closure is the initial far reference to a remote object, from which other far references may be derived by message passing.

An object matches a discovery request if its exported type tag is a subtype of the type tag argument to the discovery event handler. This implies that service discovery is polymorphic: a discovery request for a `Printer` may be satisfied by a `ColorPrinter` object, provided that the `ColorPrinter` tag is a subtype of `Printer`. A discovery request only triggers on objects owned by other actors; an actor does not discover its own exported objects. It is possible for `whenever:discovered:` to trigger on the same remote object multiple times (e.g. when a temporary network partition is healed). The programmer should thus take this possibility into account.

Analogous to the return value of the failure event handlers discussed in section 4.1.4, the discovery event handler returns a subscription object of which the `cancel()` method cancels the registration of the closure. There exists a variant event handler, accessible as a primitive named `when:discovered:`, which only applies the closure to the first matching discovered object, and afterwards automatically cancels its registration.

4.1.5 Designating Groups of Objects with Ambient References

When writing AmbientTalk code to query nearby services for data (e.g. all nearby temperature sensors in a wireless sensor network), a form of multicast is needed. To ease the writing of multicast queries in AmbientTalk, a dedicated data type was introduced called *ambient references* [Van08]. Ambient references denote a collection of nearby services of the same type. This collection is constantly kept up-to-date with the proximate physical environment: newly discovered services are added to the collection, while unresponsive services are removed from it.

Sending a message to an ambient reference can happen in different ways, depending on how the message is annotated with dedicated type tags. A first example is given below.

```
1 def sensors := ambient: TemperatureSensor;
2
3 when: sensors<-getTemp()@Any resolved: { |temperature|
4   // process the sensed temperature
5 };
```

The keyword `ambient:` allows one to create an ambient reference given a type tag. The variable `sensors` contains an ambient reference that constantly refers to all nearby services exported as `TemperatureSensor`. In the above example, the message `getTemp` sent via the ambient reference `sensors` is annotated with `@Any`. This will cause the ambient reference to send the message to *any* discovered object exported as `TemperatureSensor`.

Ambient references, however, allow to vary their message sending behavior by annotating the messages sent via them differently. For example, consider a variation on the example given above where the `getTemp` message should not only be sent to a single temperature sensor, but to *all* sensors in range. This can happen in two ways. Either the message is annotated with `@All`, which will cause the ambient reference to perpetually discover new temperature sensors and sending them the `getTemp` message during its life span[6].

A message can also be annotated with an expiration period. If a message has an expiration period, it will keep discovering new matching services and sending them the tagged message until its expiration period has elapsed. This is shown in the example below:

```
1 def sensors := ambient: TemperatureSensor;
2
3 whenAll: sensors<-getTemp()@Expires(5.seconds) resolved: { |temperatures|
4   // process the sensed temperature values
5 };
```

In the above example, the message `getTemp()` is asynchronously multicast to these services with an expiration period of 5 seconds. This implies that the message may be received by all proximate sensors at the time it is sent, as well as to all additional sensors discovered within the next 5 seconds.

The `whenAll:becomes:` control structure allows the programmer to install an event handler that can be used to gather the results of the query. Within this event handler, `temperatures` refers to an array containing the readouts of the sensors that replied. The event handler is triggered when the message's expiration period has elapsed. Ambient references support only weak delivery guarantees: some sensors may not have received the `getTemp()` message, and some replies to the message may have gotten lost or may arrive too late, in which case they are discarded.

Similarly, there is also a `whenEach:resolved:` event handler that is invoked each time a new reply for the message sent over the ambient reference is received.

The above example shows how ambient references relieve the programmer from having to deal explicitly with the events of discovery and loss of nearby services: ambient references transform these events into additions to or removals from the collection they designate. However, the programmer must still deal with the replies to the query by means of the `whenAll:becomes:` or `whenEach:resolved:` event handlers. For example, to keep an average temperature up to date by means of the temperature readings received from multiple sensors, one must resort to the following event-driven code:

[6] Additionally, the message can be annotated with `@Once` together with `@All`, which causes the ambient reference to multicast the message only to the services in range at that specific point in time

```
def totalTemp := 0;
def counter := 0;
def averageTemp := 0;

def sensors := ambient: TemperatureSensor;

whenEach: sensors<-getTemp()@All resolved: { |newTemp|
  totalTemp := totalTemp + newTemp;
  counter := counter + 1;
  averageTemp := totalTemp / counter;
};
```

4.1.6 Linguistic Symbiosis with the JVM

AmbientTalk objects can access objects in the underlying Java Virtual Machine (JVM) by means of a technique known as a *linguistic symbiosis* [GWDD06]. This will be used in the following chapter to connect an AmbientTalk program to a sensor (an RFID reader) with which AmbientTalk programs interact using a Java API.

AmbientTalk's symbiosis layer offers both a *data mapping* and a *protocol mapping*. The former ensures that data in one language looks like data in the other language, such that the symbiosis becomes syntactically transparent. For example, JVM objects are represented as objects in AmbientTalk, such that messages can be sent to objects regardless of their native language. AmbientTalk's data mapping is similar to that of other dynamic languages implemented on top of the JVM. In a nutshell, JVM values are either mapped to primitive AmbientTalk objects where possible (for example, a `boolean` is mapped onto the `true` or `false` prototype in AmbientTalk) or are otherwise represented as regular AmbientTalk objects.

The latter allows that both AmbientTalk and JVM objects communicate by sending messages, but AmbientTalk is dynamically typed while the JVM is statically typed and exploits type overloading during method lookup. The symbiosis attempts to resolve overloading automatically by inspecting the type and number of actual arguments. If automatic overloading fails, the AmbientTalk programmer is forced to pass type information in the call (how this is done is thoroughly explained in [VMD07]).

Primitive AmbientTalk objects are mapped to primitive JVM values when possible (e.g. an AmbientTalk integer is mapped to an `int`). If such a conversion is not possible, the AmbientTalk/JVM symbiosis can represent the AmbientTalk object as a regular JVM object, but only if the static type of the variable which is to hold that JVM object is an interface type. This means that for every AmbientTalk object used from within Java code, the programmer must explicitly create a corresponding interface.

```
def Button := jlobby.java.awt.Button;
def button := Button.new("Click me");

button.addActionListener(object: {
  def actionPerformed(actionEvent) {
    system.println("The button was pressed");
  };
});
```

To illustrate how AmbientTalk objects can be passed to JVM objects, consider the example shown above. It shows the archetypical pattern of registering a listener object on a button GUI widget to act upon event notifications (written in AmbientTalk, but using

the actual Java AWT framework)[7]. The above code demonstrates that an instance of the Java class `java.awt.Button` appears as an AmbientTalk object `button`. It also demonstrates that AmbientTalk text (`"Click me"`) is transparently converted by the symbiosis into a `java.lang.String`. The `addActionListener` method defined on instances of the Java `Button` class takes a parameter of type `ActionListener` as its argument, which is an interface type. As a result, it is allowed to pass any AmbientTalk object to this method; the object is not even required to implement all declared interface methods, although the anonymous object passed in the above code does properly implement the `ActionListener` interface. The symbiosis transparently wraps the AmbientTalk object into a wrapper implementing the `ActionListener` interface. The AWT framework will invoke the `actionPerformed` method on the wrapper whenever the button is pressed.

For a discussion how the concurrency issues are resolved when combining AmbientTalk's event loop model with the JVM's concurrency model based on threads, we refer to the article "Linguistic Symbiosis Between Event Loop Actors and Threads" [VMD08].

4.1.7 Summary

AmbientTalk is especially suitable for mobile ad hoc network applications due to its strictly asynchronous communication between objects owned by different actors. The built-in message queues of actors and eventual references decouple communication in time and synchronization, making the application resilient to transient network failures. The failure handling strategy of buffering messages while disconnected is a good default if failures are mostly engendered by temporary network partitions.

A traditional RPC or RMI communication model is not able to provide a similar decoupling. To abstract over temporary failures, objects would either remain blocked waiting for an outstanding RPC to a disconnected object (making the application unresponsive), or the RPC would fail, forcing the programmer to deal with every failure, even if it is only temporary. On the other hand, purely undirected communication – such as for example in some tuple space systems [MPR01] and publish/subscribe – makes it harder to designate a single remote object. Asynchronous communication over far references seems a good middle ground.

In mobile ad hoc networks, services have to be discovered in the proximate environment as devices are roaming. Since no shared infrastructure is available objects should not be required to rely on a third party to discover one another. To enable decentralized service discovery, each AmbientTalk interpreter is equipped with a topic-based publish/subscribe engine. The topics are the type tags used to classify objects in a meaningful way, independent of any particular device address, catering to anonymous interactions among objects. Each actor can independently export objects and subscribe to be notified of objects that become available. Hence, AmbientTalk applications are conceived as *decentralized event-driven architectures*, where application components (services) are discovered by means of some abstract description, such as a type tag acting as a topic in AmbientTalk. Such architectures allow loose coupling between these components such that they can be dynamically matched to each other in a mobile ad hoc network and allows them to be resilient to the volatile connections that interconnect them.

[7] The `jlobby` object is a special AmbientTalk object of which the fields correspond to packages and classes available in the underlying JVM.

Finally, AmbientTalk is conceived as an extensible language laboratory. It is extended with new language constructs through its reflection and metaprogramming facilities. Still, it can be used for realistic applications due to its full language symbiosis with the underlying Java Virtual Machine.

For these reasons, we will rely in the rest of this dissertation on (a variant of) AmbientTalk to implement ambient-oriented dataflow programs.

4.2 The Inversion of Control Problem

In the previous section, we have given an overview of the AmbientTalk ambient-oriented programming language and concluded it offers a solid basis for mobile RFID-enabled applications. Still, using such an ambient-oriented programming language comes with a price. Ambient-oriented applications are entirely event-driven. They consist of event handlers that are asynchronously invoked when new services move into range, move out of range, move back in range, or signal requests to each other. Table 4.2 lists the callbacks that must be manually coordinated by the programmer and their role in AmbientTalk programs.

`when:discovered:`	Reacting on the discovery of a new object.
`when:disconnected:`	Reacting on the disconnection of an object.
`when:reconnected:`	Reacting on the reconnection of an object.
`when:becomes:`	Reacting on the reply (future) to an asynchronous message.
`when:becomes:catch:`	Reacting on a distributed exception (future) caused by an asynchronous message.
`whenEach:becomes:`	Reacting on the individual replies to asynchronous messages sent to a group.
`whenAll:becomes:`	Reacting on the replies to asynchronous messages sent to a group as a whole.

Table 4.2: Different callbacks in an AmbientTalk application.

Although AmbienTalk's concurrency model based on event loops and its support for first-class block closures circumvents some inconveniences resulting from event-driven code, from a software engineering perspective, such event-driven code still exhibits a number of undesirable properties. These problems are described by a concept known as *inversion of control* [HO06].

Consider an AmbientTalk application that must react on the appearance of a certain service together with the reception of a reply to an asynchronous message (to a different remote object), as shown below.

```
1  def NOTHING := 'NOTHING;
2  def SERVICE := 'SERVICE;
3  def REPLY := 'REPLY;
4  def READY := 'READY;
5
6  def state := NOTHING;
7
8  whenever: ServiceType discovered: { |service|
9    if: ((state == REPLY).or: { state == READY }) then: {
10     state := READY;
```

```
11      executeReady(); // Execute actual reaction
12    } else: {
13      state := SERVICE;
14    };
15
16    when: service disconnected: {
17      if: (state == READY) then: {
18        state := REPLY;
19      } else: {
20        state := NOTHING;
21      };
22    };
23  };
24
25  when: obj<-message() becomes: { |reply|
26    if: ((state == SERVICE).or: { state == READY }) then: {
27      state := READY;
28      executeReady(); // Execute actual reaction
29    } else: {
30      state := REPLY;
31    };
32  };
```

Such code exhibits a number of software engineering issues that are formulated by Maier et al. in their technical report [MRO10] on a dataflow programming library for the Scala language in the context of a graphical user interface library. We repeat them below, but shift the problem domain to distributed AmbientTalk applications and slightly adapt them to the language features offered by AmbientTalk.

Side-effects or code nesting Event handlers or callbacks promote side-effects. Since callbacks are stateless, we often need several of them to simulate a state machine. Consider the AmbientTalk code given above. Either the different states in which the application can be are encoded in global variables that both callbacks side effect, or, thanks to AmbientTalk's event handlers based on block closures, one event handler is nested in the other. In the second case, the control flow of the application is both driven by the events as well as by how the event handlers are nested. If not carefully designed, one event handler could never be triggered because the enclosing event handler is never triggered. For example, note the subtle interplay between a `whenever:discovered:` event handler and its nested `when:disconnected:` event handler. The outside callback is triggered multiple times, and in response the inner callback is registered again. In many other cases, side-effects and global variables are required to implement the correct behavior.

Encapsulation Since the global variables mentioned in the previous item escape the scope of the event handlers, event handlers break encapsulation.

Composability Multiple event handlers form a loose collection of objects that deal with a single concern (or multiple, see next point). Since multiple event handlers are installed at different points at different times, we cannot, for instance, easily dispose them altogether. For example, the programmer must manually cancel the event handlers of the example given above, or they will keep assigning the global variable of the example. This can interfere with newly registered event handlers, causing subtle and hard to track bugs. Furthermore, if a `whenever:disconnected:` inner callback would have been used, the callback should have been cancelled when it has finished executing, or it will be fired each time the `service` disconnects.

Scalability Adding more event sources to the example increase the complexity of the application quadratically because more states must be encoded and checked, and their transitions must be carefully managed. Furthermore, the code will be dispersed over different event handlers, maybe even separately developed.

Uniformity Although AmbientTalk offers a generic event loop model, the constructs it offers for actually reacting to and processing specific events are ad hoc constructs tied to the specific event they are supposed to be used with. New types of events require again adding new ad hoc constructs. In short, AmbientTalk has no uniform representation for events and no uniform way of reacting to them and processing them. This stems from the initial vision on event-driven ambient-oriented programming that primarily dealt with asynchronously received messages or remote parties appearing and disappearing dynamically. The most apparent issue in this code example are that different methods to install different observers are needed, which decreases code uniformity.

Abstraction In the example above, we cannot abstract over the precise event sources. For instance, replacing one of the event types with for example a user interface event requires rewriting the entire example.

Semantic distance Ultimately, the example is hard to understand because the control flow is inverted which results in too much boilerplate code that increases the semantic distance between the programmers intention and the actual code.

Because of AmbientTalk's event loop concurrency model, the overhead of triggering event-driven code is a fact of life and cannot be circumvented. Hence, we drop the *resource management* item mentioned by Maier et al. Also, in this particular example, the *separation of concerns* item is not applicable and is dropped as well. The first item is concerned with the aggressive cancellation of unneeded callbacks to prevent the runtime to taking them into account and by doing so wasting resources. The second item is concerned with different concerns that are treated in the same code location.

Besides the items listed above, we identify one additional issue not listed by Maier et al.:

Registration management When callbacks have to be explicitly registered and cancelled, as in the example shown above, the programmer is burdened with making sure that every single callback is correctly registered or cancelled at every point in time to not miss specific events or to not execute unintended code in response to an event. The registration of callbacks has to be managed throughout the lifetime of a program. Additionally, the order in which callbacks are registered also determines the order in which they are executed when they are trigged by the same event. Hence, their execution order could also introduce subtle bugs if not carefully managed.

In general, in event-driven applications control flow management is partly the responsibility of the programmer instead of the runtime of the language. The control flow is said to be *inverted* as events cause the asynchronous execution of code as opposed to procedure or method calls which are associated with a calling context to return to by the language runtime. The effect grows as more event types are added.

4.3 Conclusion

Ambient-oriented programming is a paradigm the feature set of which is developed specifically to accommodate the idiosyncratic properties of mobile ad hoc networks. Such networks, composed of mobile devices with wireless communication links exhibit two discriminating characteristics: connections between devices are volatile and infrastructure is scarce or non-nexistent. Networks may be formed ad hoc simply by colocating devices.

This requires distributed applications to be structured as autonomously and concurrently running components that are loosely coupled with respect to time, space and synchronization. In such spontaneously formed networks, in many cases several distributed components are functionally equivalent, or must be semantically addressed as a single group. In such cases, this requires ambient-oriented applications to abstract over communication arity as well.

AmbientTalk/2 is an ambient-oriented programming language embodying the paradigm. AmbientTalk's classless object model, its concurrency model based on communicating event loops, its asynchronous and reliable communication primitives, and its decentralized service discovery mechanism offer a strong foundation for mobile RFID-enabled applications, the type of ambient-oriented applications targeted in this dissertation. AmbientTalk's metaprogramming and reflection facilities together with its linguistic symbiosis with the underlying JVM render it an attractive language for experimenting with new language constructs.

Still, AmbientTalk's event-driven execution model stemming from its adherence to the paradigm forms the culprit of many software engineering issues found in event-driven systems, such as inversion of control. Although many of AmbientTalk's event handling constructs return futures which in their turn will synchronize asynchronously received messages, this requires converting synchronous code to asynchronous future-based code, and ultimately, the introduction of callbacks. The effect is that when larger applications are modularized, the asynchronous interplay between the modularized parts quickly turn into collections of unpredictably firing callbacks that must carefully coordinated by the programmer, and consist of fragile code due to their lack of functional composition and encapsulation. This is aggravated by the lack of uniformity in the language's event handling constructs.

In the next section, we apply the principles of the paradigm through an extension of AmbientTalk to mobile RFID-enabled applications where physical objects have an object-oriented representation in the language. Naturally, this increases the amount of events in ambient-oriented applications interacting with these objects. This leads us to pursue an alternative paradigm for dealing with event-driven applications in the subsequent chapters of this dissertation.

Chapter 5

Ambient-Oriented Programming for Mobile RFID-Enabled Applications

In the previous chapter, we explained AmbientTalk, an existing language for programming mobile ad hoc network applications. In this chapter, we extend the programming model of this language to represent RFID-tagged physical objects as *remotely hosted software objects*, barring the distinction between software and "real" tagged objects. We believe that such a uniform representation of all objects interacting in a software system eases the development of mobile RFID-enabled applications. By doing so, we open up the ambient-oriented programming paradigm from a network of devices to a network of "things".

The fundamental contributions discussed in this chapter are a technical one and a conceptual one. The technical one is the realization of an ambient-oriented programming framework for mobile RFID-enabled applications in AmbientTalk. The conceptual one is the extension of ambient-oriented programming to a setting where remote objects can be hosted on devices which have no autonomous processing power, such as RFID tags. Both contributions do not change the fundamental criteria of ambient-oriented programming, but instead allow these criteria to be applied to a novel type of applications.

First, in section 5.1 we motivate our design choices by mapping the programming model requirements for mobile RFID-enabled applications put forward in chapter 2, section 2.3 to a real-world example. Thereafter, section 5.2 discusses a number of changes to the base AmbientTalk language to be able to implement mobile RFID-enabled applications. Building upon these changes, section 5.3 discusses how our requirements for mobile RFID-enabled applications are satisfied by AmbientTalk constructs extended towards mobile RFID-enabled applications. Section 5.4 gives an overview of the implementation. Finally, section 5.5 concludes this chapter.

5.1 Motivation

In this section, we motivate the main design choice of integrating the RFID programming model with the ambient-oriented programming model and representing RFID-

tagged objects as true mutable software objects that are presented to client applications as remote references to these objects. We call these software representatives for physical objects *things*. We first give a scenario – that we will use as a running example throughout this chapter – and subsequently use this scenario to distill the requirements we put forward for a programming model for mobile RFID-enabled applications earlier in chapter 2 section 2.3.1. Additionally, we discuss how the AmbientTalk language should be extended to support each requirement.

5.1.1 A Mobile RFID-enabled Application Scenario

The scenario consists of a library of books that are all tagged with writable passive RFID tags. The user of the mobile RFID-enabled application carries a mobile computing device – such as a smartphone or tablet PC – that is equipped with an RFID reader. On this device, there is software running that allows the user to inspect the list of books that are nearby (i.e. in the reading range of the RFID device) sorted on various properties of the books (e.g. author, title, ...). This list is updated with the books that enter and leave the range as the user moves about in the library. The user can select a book from the list of nearby books, after which a dialog box opens. In this dialog box, the user can write a small review about the book. This review is stored on the tagged book itself. Other users can then select that same book from their list of nearby books and browse the reviews on the book, or add their review. Figure 5.1 shows a screenshot of the mobile RFID-enabled application.

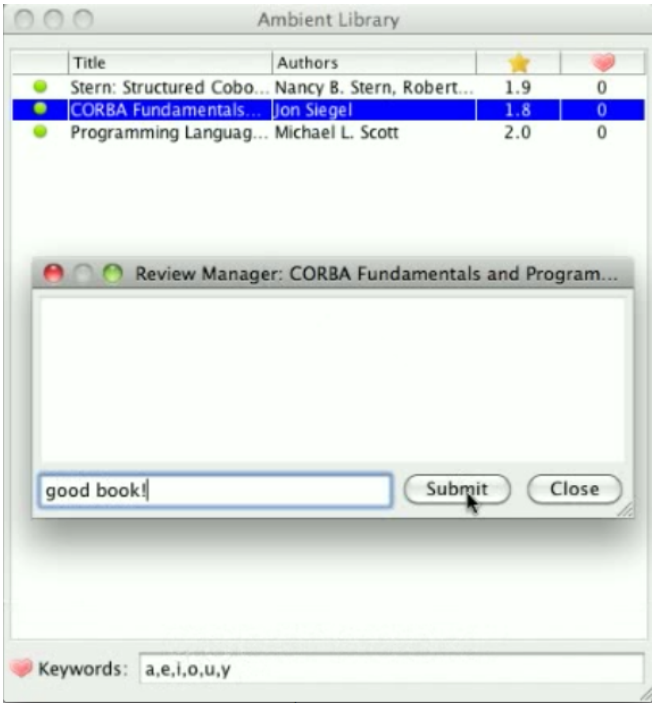

Figure 5.1: Screenshot of the mobile RFID-enabled library application.

5.1.2 Ambient-Oriented Programming with RFID Tags

In the mobile RFID-enabled application introduced in the previous section, mobile devices hosting the application move throughout an environment of tagged books. These books dynamically enter and leave the communication range of the mobile devices and interact spontaneously. As we pointed out earlier, these properties are very similar to the ones exhibited by distributed applications in mobile ad hoc networks. Similar to mobile devices in mobile ad hoc networks RFID tags and readers should be able to interact spontaneously when their ranges overlap. Below, we put the requirements for mobile RFID-enabled applications (discussed in chapter 2 section 2.3) next to parts of the solutions provided by ambient-oriented programming (discussed in the previous chapter). A straight-forward adoption of the model still leaves some issues to be solved, which are discussed below.

Addressing physical objects. In the ambient-oriented programming model, remote objects can be addressed using directed communication primitives. We propose to represent RFID-tagged physical objects as far references to distributed objects called *things*. In the scenario, this makes it possible to send messages to and to be notified of events coming from an individual book.

Storing application-specific data on RFID tags. In the ambient-oriented programming model, objects are self-contained and classless such that they can be easily transmitted across the network and can naturally encapsulate mutable state. We propose to represent RFID-tagged objects as true mutable software objects (things) that encapsulate the state on the bare memory of the tag and make it accessible through a well-defined interface. In the scenario, this means there is a prototype object representing books with a well-defined interface which can be used to query or change its internal state.

Reactivity to appearing and disappearing things. In the ambient-oriented programming model, disconnections are not considered as errors nor are they hidden from the programmer. They reflect the physical connectivity state of remote objects. We propose to encapsulate things in a dedicated RFID event loop that signals connectivity changes to client applications (RFID event loops and client applications can be hosted on the same device). In our scenario, this means that books of which is detected that they are out of range, can be removed from the user interface.

Asynchronous communication. In the ambient-oriented programming model, communication with remote objects can only happen asynchronously. The dedicated RFID event loop must take care of buffering incoming messages, processing them sequentially, and signaling back replies. This means in the scenario for example that a user storing a new review on a book must not deal with a number of error messages when the connection between his mobile device and the book is temporarily broken. Instead, the system keeps attempting to transfer the message, until some application-specific timeout is reached. Additionally, reading from and writing to RFID tags are operations that are slow in comparison to general computations. Asynchronous communications allows the latency caused by these operations to be hidden by executing them in the background while other computations are performed.

Fault-tolerant communication. In the ambient-oriented programming model, messages that are not acknowledged are not lost. They can be resent later. The dedicated RFID event loop must take care of synchronizing the thing with its physical counterpart (e.g. when methods perform side effects on its internal state) and do this in a fault-tolerant way. In our scenario, reviews being stored on the book should be stored completely, or not at all. In the latter case the client application should be notified of such a failure.

Data consistency and security. The dedicated RFID event loop must take care of orchestrating access to the bare RFID memory such that no data races occur. In our scenario, this means that different users could use their mobile devices to wirelessly manipulate the same set of books without experiencing conflicts. Similarly, in some scenarios data stored on RFID tags may not be read or modified by unauthorized users. In this work, we have not tackled security yet. Security is an issue that to be solved requires securing the entire RFID technology stack, down to the hardware level, to be completely covered [RK09], which is out of the scope of this dissertation.

5.2 Changes to AmbientTalk and Its Interpreter

In order to support the requirements listed in the previous section, we first had to adapt the AmbientTalk language and its interpreter.

5.2.1 Fine-Grained Connectivity Handling

In the communicating event loops model, an event loop forms the unit of distribution. Because of that, AmbientTalk only provides a primitive to take online and offline an event loop as a whole, which entails that either all of the published objects hosted by the event loop are made available or all or none of them are made unavailable. In the setting of mobile RFID-enabled applications however, the RFID event loop hosting the RFID-tagged objects has to be able to take online and offline the corresponding things *individually*, since the connectivity of RFID tags varies individually. The alternative is spawning an event loop per scanned tag, which is clearly not scalable. To allow this, we extended the event loop model to enable a more fine-grained control over the connection status of objects. Event loops remain the unit of concurrency – and conceptually: distribution – but in addition it is possible to programmatically disconnect or reconnect a single exported object. To this end we introduced a new operation `disconnect:` that logically disconnects the published object it receives as an argument.

```
1 thing.disconnection := disconnect: thing; // RFID tag leaves range
2
3 thing.disconnection.reconnect();          // RFID tag reenters range
```

The above code shows the use of a *disconnection object* returned by disconnecting an exported object. Calling this object's only method `reconnect` reestablishes the exported object's connection.

5.2.2 Maintaining Thing Identity Using Multiway References

In section 5.1.2, we proposed to represent RFID-tagged objects as remote software objects which can be accessed via an AmbientTalk far reference. The connectivity to a

physical RFID tag is reflected by the connectivity state of the far reference. In the case the mobile device (such as an RFID-enabled smartphone) has discovered a tag using an internal reader, the thing representing the tagged physical is hosted locally on the device. The RFID event loop of the device then offers this thing as a far reference to applications locally hosted on the device, but also to any other application running on another device that is interconnected with the device hosting the thing.

For (mobile) devices that rely on one or more external RFID readers (such as smartphones without an integrated RFID reader), the connectivity state of a far reference reflects both the physical presence of the respective RFID-tagged physical object in range of the reader as well as the connectivity to the device hosting the thing representing the physical object. Figure 5.2 illustrates a scenario where a mobile device acquires far references to the same thing via a remote fixed reader and via its integrated reader. Things that are referenced via different RFID event loops arise when these objects roam through the physical space in which multiple RFID readers are present, which in their turn are used by mobile devices to detect things.

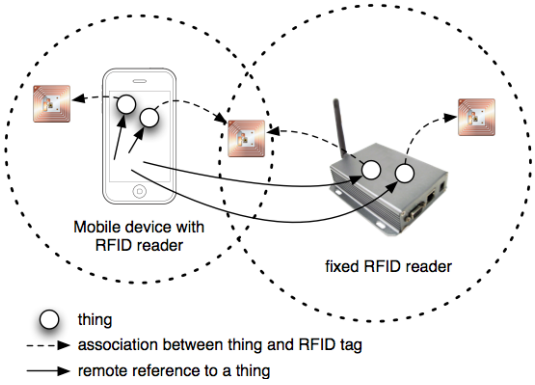

Figure 5.2: Using multiway references to abstract over multiple connections.

Our approach installs an RFID event loop per RFID reader. This entails that per tag discovered, per RFID reader, an AmbientTalk object (thing) for this tag is constructed. Therefore, an application that discovers RFID tags from more than one reader might end up with two *disjoint* references referencing the *same* thing, through *different* AmbientTalk objects. This is illustrated by figure 5.2. However, to be able to **address physical objects** – one of the requirements listed above in section 5.1.2 – thing identity must be maintained. We addressed this issue by relying on an abstraction called *multiway references*, first introduced by Pinte et al. [PHD11].

Multiway References

Multiway references are a generic abstraction that collapses far references considered "equal" based on an equivalence relation associated with every single reference. For things, far references are considered "equal" if they point to things with the same unique serial number. To applications that acquire references a multiway reference appears as a single far reference. However, the multiway reference incorporates the different *ways* a physical object can be reached.

Figure 5.3 shows a multiway reference to a book object that can be reached in two ways. The client application however discovers the book only once: when new copies

of things denoting the same physical book are discovered, the multiway reference is extended with the references to these new copies.

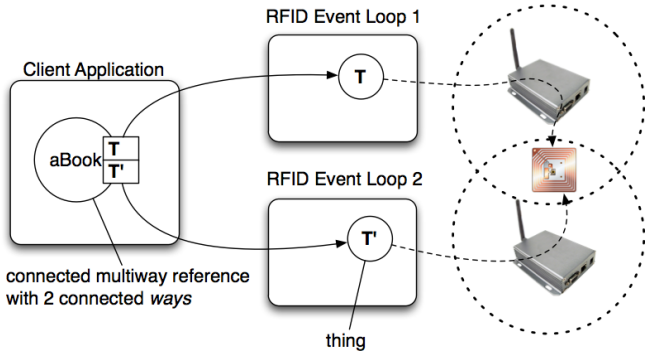

Figure 5.3: A multiway reference to a single physical book through multiple things.

A multiway reference is thus able to reach a single RFID tag via multiple paths and does the bookkeeping of the connection status of each of the paths. A multiway reference is only disconnected if each single reference it incorporates is disconnected. This is illustrated in figure 5.4. The picture shows a multiway reference referencing a

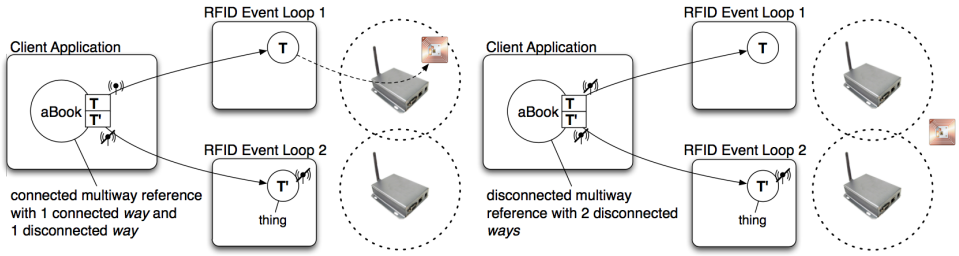

Figure 5.4: A multiway reference is connected up until all of its composing remote references are disconnected.

thing via two RFID event loops. In the left-hand side the multiway reference is marked as connected because the thing can be reached via one of the two RFID event loops. In the right-hand side the multiway reference is marked as disconnected because the thing is not reachable anymore via any RFID event loop.

As with far references, all messages sent to a disconnected multiway reference are buffered in the reference. As soon as one of the encapsulated far references reconnects or a new path is discovered, the stored messages are flushed over the connected path. Each multiway reference keeps one of its references as the *active* path. When sending messages to the multiway reference, the messages will flow over this active path to the destination. Selecting the reference that becomes the active path happens by ranking the encapsulated remote references based on the priority carried by each of them. This priority can be anything from a simple number indicating e.g. the number of hops a reference has to make to reach the device hosting the target AmbientTalk object, to a full fledged object that can embody more detailed information such as the range of the reader, the signal strength, etc.

Property References

A multiway reference is a higher order reference that clusters references to different objects that are *equivalent*. To implement this behavior, multiway references rely on a small extension to the bare AmbientTalk far references. *Property references* extend far references with two functions: an equivalence relation to check wether two references are equivalent, and a priority function that is used within the multiway reference to order the references and select the default path. Property references were previously introduced by the same researchers that introduced multiway references [PHD11].

A property reference is a construct to extend a reference to an exported object with additional information. Exporting an object in AmbientTalk is performed using the `export:as:` operation which takes the object to export as a first argument and a type tag as its second argument. Property references extend this construct such that a third argument (*property*) can be specified that will be locally accessible on the device on which a far reference to the object is obtained. In the example below, a book thing is exported with a type tag `Book` together with a property that contains two functions which implement the equivalence relation and the priority function (greater than) to support multiway references:

```
1 deftype Book;
2 export: book as: Book with: {
3   def serialNr := book.serialNr;
4   def equivalent(otherRef) { serialNr == otherRef.serialNr };
5   def >(otherRef) { ... };
6 };
```

Multiway references use property references as follows. A multiway reference internally holds a message queue that is used to buffer messages in case all of the internal property references it contains are disconnected. When a new property reference is obtained that is deemed equivalent by the equivalence relation it encapsulates, the property reference is added to that multiway reference. If the reference matches no known equivalence relationships, a new multiway reference is constructed containing the property reference as its only path. Upon acquiring a new property reference or upon changes of the connection status of one of the property references that belong to the multiway reference, the multiway reference will recompute the default path by means of its priority function. This happens by simply ordering all encapsulated property reference using the “>” (“greater than”) operator that they implement.

Except the semantics described above, multiway references behave exactly as ordinary far references do. For the remainder of the text we will employ the term “far reference”, but note that they are actually multiway references.

5.2.3 Serializing Things

Things must be serialized into a self-contained representation that includes their method definitions in order to store them on RFID tags. We extended AmbientTalk with the `asCode:` primitive, that when it is passed an object, generates a string representation of the object. It can be regarded as a self-contained version of the JavaScript object serialization offered by JSON (JavaScript Object Notation) [Cro06] for AmbientTalk. The difference with JSON is that it includes a textual representation of the serialized object’s methods. Just like with JSON, it suffices to evaluate the generated string to obtain a copy of the object, as shown below:

```
def iso := isolate: {
  def x := 5;
  def print() { system.println(x) };
};

def str := asCode: iso;

def copy := eval: str in: self;
copy.x;          // Returns 5
copy.print();    // Prints 5
```

The primitive is recursively applied to objects referenced from within the object passed to the primitive. Of course, it makes more sense to use this functionality on isolates than on normal objects (which additionally to their object scope also have access to their surrounding lexical scope). Serializing far references using is not supported by this mechanism

5.3 Programming Mobile RFID-Enabled Applications in AmbientTalk

In this section, we explain our RFID programming model in more detail using the scenario presented in section 5.1.1. The model is conceived as a set of AmbientTalk language constructs offered by the language or implemented on top of it that correspond to the properties listed in section 5.1.2.

5.3.1 RFID-tagged Objects as Things

As already mentioned, we represent RFID-tagged objects from the client application's perspective as far references to things. Our basic model supports passive, writable RFID tags without computing power whatsoever. For this case, the thing representing the RFID-tagged physical object must be a proxy or stand-in object: the RFID tag itself is not capable of executing method calls. This means that client applications interact with the physical object through a far reference to such a thing.

An example of a book thing is given below. It contains slots for the ISBN, title and reviews and provides two *mutator* methods to update the book's title and add reviews:

```
deftype Book;
def aBook := object: {
  def ISBN := 123;
  def title := "My Book";
  def reviews := Vector.new();

  def setTitle(newTitle)@Mutator {
    title := newTitle;
  };

  def addReview(review)@Mutator {
    reviews.add(review);
  };
} taggedAs: Book;
```

The limitations of RFID tags render it impossible to deploy a full fledged virtual machine hosting objects on the tags themselves. We thus store a serialized data representation of a thing on its corresponding tag, including its methods. Because of Ambi-

entTalk's prototype-based object model, objects are self-contained: there is no separate class that defines their behavior. Upon deserialization, the object's behavior (its methods) is preserved and used to reconstruct the thing (see section 5.3.2). Since we cannot rely on classes to categorize objects, we use AmbientTalk's *type tags*: here they act as a lightweight classification mechanism (comparable to empty "marker" interfaces in Java such as for example `Serializable`) that are attached to an object to identify its "type". In the above example, we define a type `Book` on line 1 and attach that type to the `aBook` object on line 14. In section 5.3.3 we use the type tag to discover objects representing tagged books.

The book object thing given above is in standard AmbientTalk. Of course, the data stored on the tags has to be synchronized with the state of these things. Methods that change the state of the book things are annotated by the programmer with the special `Mutator` annotation[1] that will cause the RFID implementation to treat this methods when invoked specially. These annotations are used by the RFID implementation to detect when things change and have to be written to the corresponding tag. For example, calling the `addReview` mutator method on a book thing first updates the `reviews` field by adding the new review. Subsequently, the system entirely serializes the modified book thing and stores it on the correct RFID tag.

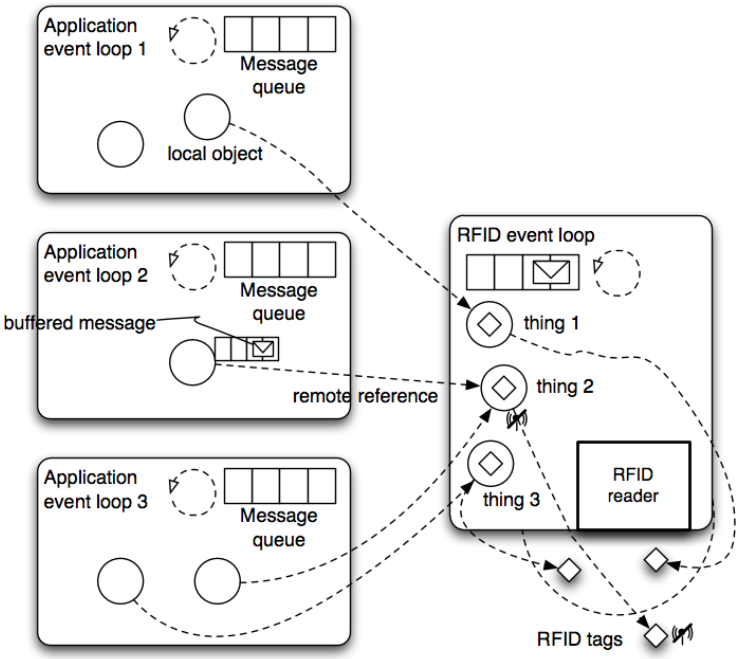

Figure 5.5: Overview of the RFID event loop.

Things are managed by a special kind of AmbientTalk actor that we called the *RFID event loop*, as shown in Figure 5.5. It controls an RFID reader to detect appearing and disappearing tags and it associates things with them. These things can then be used

[1] AmbientTalk's symbiosis with Java makes it hard to determine from the source code if mutating operations are going to be invoked. Additionally, allowing the programmer to specify which methods are mutators and which are not leaves room for optimizations or other reasons not to (immediately) write through the changes to the thing.

by other AmbientTalk event loops – after acquiring far references to them – to interact with the tags as if they were mutable software objects. Far references reflect the state of the corresponding RFID tags. When a tag moves out of range of the reader the far reference is signaled of this disconnection; conversely, when a tag moves back in range the far reference is signaled of the reconnection.

Abstracting Over Specific RFID Hardware

Embedding RFID technology in the ambient-oriented programming model requires abstracting away the hardware-specific protocols in our generic abstraction called the RFID event loop (see section 5.4 for an overview of the architecture). This even allows multiple RFID readers to be used in one or more mobile RFID-enabled applications using the same abstractions. Figure 5.6 depicts a mobile device that hosts three different applications that interface with RFID hardware by means of two RFID event loops steering different types of RFID hardware.

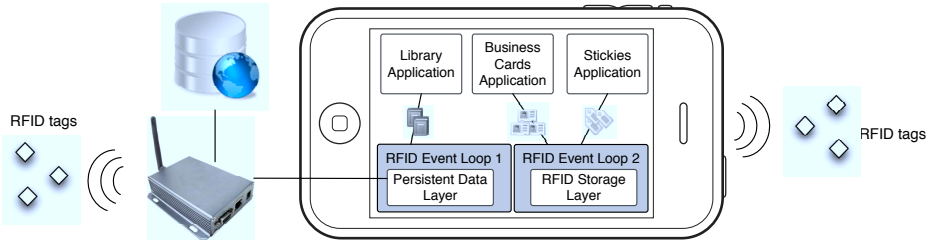

Figure 5.6: RFID event loops and different applications interfacing with it.

This not only allows abstracting over hardware protocols, but also over where the data associated with the RFID tag is physically stored. Although we are focusing on pervasive applications that do not assume any infrastructure, our abstractions do not limit programmers to this kind of application. In fact, when using the same programming model it can be made transparent whether the data associated with an RFID tag is stored onto the tag itself or is stored in a (potentially remote) external database (see section 5.4). The key point is here that although the data itself may be permanently accessible (using a stable internet connection to the database for example), the application interacting with the physical world might still be interested in whether the physical object denoted by the tag is in communication range (see requirement 5.3.3), and in addition the same interface to program RFID applications is offered to the application programmer.

Figure 5.6 depicts a mobile device that hosts three different applications that interact with two differently configured RFID event loops that each interact with their own RFID reader. The library application looks up information associated with tagged books in a centralized database by means of the serial number on the tag, which is scanned by a stationary reader to which the mobile device is connected (for example using a local WiFi network). The other two applications directly use the RFID reader built into the mobile device and do not assume any infrastructure by storing all necessary data on the tags themselves. The first approach can utilize well-understood database technology and the latter approach is the topic of the next section, in which we will discuss a mobile ad hoc version of the library application. This architecture allows different hardware abstraction layers and/or persistence layers to be encapsulated by the

same interface towards applications: the RFID event loop. All applications are encapsulated in their own event loop. The RFID event loop notifies client applications of the appearance and disappearance of RFID tags and takes care of (sequentially) scheduling messages sent to things hosted by the event loop and of writing the necessary data on the concrete storage infrastructure associated with the thing.

5.3.2 Storing Objects on RFID Tags

When the RFID event loop detects a blank RFID tag, the tag is represented by a generic object which responds to only one message: `initialize`. The code below shows how a blank tag is initialized as a book thing:

```
when: tag<-initialize(aBook) becomes: { |book| book<-getTitle() };
```

The RFID event loop generates a data representation of the `aBook` thing by serializing it and stores this data on the RFID tag that corresponds with the `tag` object. For storing objects on RFID tags, we currently employ a custom representation that includes the method implementations of things in their serialized representation (see section 5.2.3). The serialization strategy is not interleaved with the rest of the implementation to allow more standardized object representations to be used, e.g. encoded as an XML document. Note that these representations have to be adapted to encode prototype-based objects (by including method implementations into the serialized representation) to obtain the same functionality that we are proposing here.

The reference `tag` to the generic tag object is obtained using the discovery constructs we explain in the next section. When storing the object on the tag succeeds, the call to `initialize` returns with a new remote reference `book` that points to a newly constructed thing (the `when:becomes:`-construct is explained in the previous chapter section 4.1.3) representing the book. From this point on, the RFID tag is no longer "blank" as it contains application-specific data. The RFID event loop keeps track of the unique link between a thing and a tag by means of the serial number that each tag carries. Note that the concrete behavior of performing side effects on such a thing depends on the underlying hardware implementation. Using an entire ad hoc implementation where all the data is stored on the tag memory itself behaves differently than when the data is stored in a relational database to which the mobile device has access. However, the interface provided to the application programmer is the same.

5.3.3 Reactivity To Appearing and Disappearing Things

As explained in section 5.3.1, the RFID event loop notifies other event loops of the appearance and disappearance of the things they have far references to. In the code example shown below, an event handler that will execute a block of code each time a thing of type `Book` is discovered is installed using the standard AmbientTalk `whenever:discovered:` construct. The registered code block is parametrized by the far reference to the book thing (which is also used to send it asynchronous messages).

```
whenever: Book discovered: {|book|
  whenever: book disconnected: { // react on disappearance };
  whenever: book reconnected:  { // react on reappearance };
};
```

Once a far reference to a book is obtained, within the `whenever:discovered:` callback, two more event handlers can be registered on the `book` far reference using the

`whenever:disconnected:` and `whenever:reconnected:` constructs. These allow one to install a block of code which is executed as soon as the thing denoted by the `book` far reference moves in or out of range of the reader. Notice that upon reconnection the thing maintains its identity through the `book` reference.

5.3.4 Asynchronous Communication

Just like with normal AmbientTalk objects, applications that acquire a far reference to a thing can communicate with it via asynchronous message sending. Messages sent to things are handled sequentially by the thread encapsulated in the RFID event loop. This ensures that all things hosted by the RFID event loop are protected against race conditions (for preventing race conditions on an RFID tag's memory when it is accessed by two RFID event loops, we refer to section 5.3.6). When the far reference to a thing is disconnected, all messages sent to it are locally buffered in the far reference. When the connection is restored, the messages are flushed to the RFID event loop's message queue. This means that a message sent to a thing of which the RFID tag temporarily suffers from interference or is temporarily unavailable will eventually be processed.

Messages sent to things can either retrieve data (read operations) or trigger behavior that causes side effects (write operations). Both kinds of operations aim to keep the tag synchronized with the thing. The difference with messages to normal AmbientTalk objects is that performing a read operation on a thing causes the thing to be updated with the data on the corresponding tag. We delay the actual retrieval of data from the tag's memory to the point that such a message invoking a read operation is received to obtain up-to-date data from the connected tagged object. This is important to maintain a consistent view when such a thing is being referenced via different RFID event loops in a single application, as discussed earlier in section 5.2.2. Performing write operations first causes a side effect on the thing, thereafter the corresponding RFID tag is updated to contain the modified thing. Reading and writing tags is thus done by sending messages to the thing.

The following example asks a book for its title and displays it:

```
when: book<-getTitle() becomes: { |title| system.println(title)};
system.println("here first!");
```

This example thus immediately prints `"here first"`! and only after the `title` future signals the reply, it prints the title of the book. If the RFID tag corresponding to the book thing has disappeared upon sending the message, the far reference buffers the message until the tag reappears. This message will only be sent when the RFID tag represented by the far reference is back in range.

5.3.5 Fault-tolerant Communication

Conform to AmbientTalk's semantics, buffering an asynchronous message to a thing ensures that the message will eventually be sent if the tag moves in range. This makes the communication fault-tolerant as no exception is raised when the thing is unavailable for a short period of time. However, failures may not be temporary, a tag may move out of range and never return again. Reusing AmbientTalk's abstractions, the `Due` annotation can be used to annotate the message send with a duration that controls how long a message is buffered before timing out. For example, we can add short reviews to a book:

```
def myReview := "not suitable for beginners";
when: book<-addReview(myReview)@Due(10.seconds) becomes: { |ack|
  // message processed successfully
} catch: TimeoutException using: { |e|
  // message timed out
};
```

Suppose the RFID tag corresponding with `book` would leave the reader's range before the `addReview` message is received by the book thing. Then the message is buffered for at most 10 seconds. If the tag does not respond in time, a timeout exception is raised. If the tag reappears in range within this time frame, the message to add the review `myReview` is delivered to the RFID event loop and the corresponding book thing is updated and stored on the RFID tag. Remember from section 5.3.1 that `addReview` was annotated as a mutator method. This means that first the `reviews` field of the thing is updated by adding the new review. If the method would have not been annotated, it would have been treated as an accessor method and the change would not be written in the RFID tag's memory (this mechanism is specifically for things, other AmbientTalk objects do not require the annotation of methods). Subsequently, the RFID event loop serializes the changed book thing and attempts to store it entirely on the correct RFID tag (which might involve multiple tries because of a bad connection). Only after both of these operations complete successfully, the future object triggers all its registered `when`-observers. If this did not happen within the 10 second timeframe, the exception is signaled to client applications and their registered `catch`-blocks are invoked. This is a stronger condition to resolve a future than the condition checked by these AmbientTalk constructs for "normal" objects, which do not have to take into account the storage of data on a passive memory such as a passive RFID tag.

5.3.6 Data Consistency

Passive tags like the ones we have used in our experiments offer no opportunity to programmatically coordinate read and write operations. They do offer low-level protocols (e.g. different variants of the ALOHA protocol [FW06]) for preventing read and write collisions, but these are automatically exploited by the RFID reader when reading the data from or writing it on tag memory. For our implementation, this means that objects will always be consistently written by the hardware each time a mutator method returns (i.e. data cannot be garbled because of concurrent writes). However, on a finer granularity, successive read and write operations within a single method invocation, such as incrementing a counter based on a previously read value, can lead to data loss (as the thing containing the counter might have been concurrently updated by another RFID-enabled device before the method returns).

Hence, ensuring data consistency with passive RFID tags requires coordination among writers before physically writing the data on the tags. This is not a problem in a setting where RFID-enabled applications can coordinate by means of a shared centralized entity, e.g. by all connecting to an internet server that manages a certain type of tagged objects. The location of such servers could be stored on the tags to minimize configuration.

In the type of applications that we consider in this dissertation, where no additional infrastructure such as a centralized server is assumed, the only way of coordinating writes is by means of the tag itself. Our solution is to allow an RFID event loop to function in two modes. In the first mode, the RFID event loop only signals the RFID

reader to power up nearby tags when they have to be detected, read or written. Since requests from different readers might interleave, distributed race conditions can occur in this mode. In the second mode, the RFID event loop instructs the RFID reader to keep its RF field active, permanently powering all reachable tags, unless a client application explicitly signals the RFID event loop to shortly power down its RF field. The result is that RFID tags that are powered by the RF field of a single reader, grant exclusive access to that reader and ignore commands from all other readers. When the tag moves out of range of the reader or when the reader temporarily switches off its RF field, it loses its exclusive access to the tag and, by doing so, grants exclusive access to other nearby RFID devices willing to interact with the tagged object.

The following code snippet shows how losing exclusive access to a tagged book in the time span between sending the `getRating` message and the resulting future being resolved raises a `NoExclusiveAccessException` (specific to mobile RFID-enabled applications) in the client application such that defensive code can be triggered:

```
when: book<-getRating() becomes: { |rating|
  book<-updateRating(rating + 1);
} catch: NoExclusiveAccesException using: {|e|
  // handle lost exclusive access
};
```

Note that the `updateRating` message is immediately scheduled in the RFID event loop after the resolution of the future. No other externally sent asynchronous messages sent to the RFID event loop can be scheduled before it because the `becomes:` clause of the callback registered on the future is executed synchronously.

We have found this approach satisfactory in small scale scenarios, such as the tagged library used as a case study in this paper. In larger scale applications, keeping a number of tags powered to obtain exclusive access to them may cause performance issues, since potentially a large number of tags can cease responding and become invisible to other RFID readers.

When looking at more advanced hardware such as active RFID tags, there are more possibilities. Their autonomous nature allows to for example assign time slots (i.e. leases [GC89]) in which applications are allowed exclusive access to its memory before either aborting all operations or committing them before giving another application write access. In this dissertation, we only consider passive tags.

5.3.7 Addressing Specific Groups of Things

RFID tags are typically used in large quantities, e.g. in warehouse applications. In mobile RFID-enabled applications it is often necessary to address a specific group of things. E.g. we may want to update the price for all tags that represent a certain product. However, such a collection of things has a highly dynamic nature due to the volatile connections with the RFID tags and the mobility of the application. At any point in time, tags move out of range and new tags move in range. Instead of forcing the programmer to manually manage collections of nearby things, AmbientTalk has a dedicated abstraction to discover and address a group of objects: ambient references [VCDM+06] (discussed before in chapter 4 section 4.1.5). At any point in time, an ambient reference designates the set of proximate objects of a certain type. This abstraction is applicable because we represent physical objects as things, which are special AmbientTalk objects. An ambient reference represents a *variable* collection of things, e.g. the set of nearby books. This set is updated behind the scenes when books

move in and out of range. The example below shows an ambient reference to all books in the proximity, denoted by the `Book` type:

```
def books := ambient: Book;
```

Ambient references allow to specify various predicates to refine the set of things designated. This is shown in the example below where books are selected based on their category attribute:

```
def computerScienceBooks := ambient: Book where: {|b|
  b.category == "Computer Science";
};
```

The example below shows how we can address a single thing out of the group of nearby things encapsulated in the ambient reference. For example, if all books about computer science are placed in the same shelf in the library, it is sufficient to query any book about this topic in range for its shelf:

```
def shelfFuture := computerScienceBooks<-getShelf()@Any;
when: shelfFuture becomes: { |shelf|
  system.println("The book should be on shelf: " + shelf);
};
```

This happens by annotating the `getShelf` message with `@Any`.

We can also reach all things in range using one-to-many communication. The example below updates the shelf where computer science books should be located (e.g. because they have to be moved). The `Sustain` annotation causes the `setShelf` message to be perpetually sent to newly discovered computer science books.

```
computerScienceBooks<-setShelf("5D")@Sustain;
```

5.3.8 Putting It All Together

Finally, in this section we bring together the language constructs presented throughout this chapter to implement the motivating mobile RFID-enabled application introduced in section 5.1.1. First of all, while the user moves about in the library, the list of nearby books has to be updated. The following code snippet shows this:

```
1  deftype Book;
2  def books := ambient: Book;
3
4  whenEach: books<-getBookInfo()@Sustain becomes: {|infoAndRef|
5    GUI.addBookInfoAndReferenceToList(infoAndRef);
6  };
7
8  whenever: Book discovered: {|book|
9    whenever: book disconnected: {
10     GUI.removeBookFromList(book)
11   };
12 };
```

The first line declares the `Book` type and the second line creates an ambient reference that refers to all books in range. On line 4, the asynchronous message `getBookInfo` to the `books` ambient reference is annotated with `@Sustain`, which causes the ambient reference to perpetually send this message to newly appearing books. This returns a *multifuture* (see chapter 4 section 4.1.5, i.e. a special future object that can trigger the same callback block multiple times with a new value. This callback is registered on

the multifuture with a special `when`-construct (`whenEach:becomes:`). The code block is triggered each time the multifuture is resolved with a new return value from the message invocation on the ambient reference. The return value of this message is the info about the book (i.e. ISBN number, title and authors) and a reference to the book thing. These return values are bound to the `infoAndRef` parameter of the observer block, which is added to the list in the user interface object. This causes the user interface to show a new entry in the list of nearby books, and to associate a reference to the book entry in this list.

On line 7, for every book discovered, a `whenever:disconnected:` observer is installed that, when triggered because a book went out of range, removes the book from the list in the user interface by means of the `book` far reference. Notice that although the far reference points to an unreachable book, it can still be used to look up the book in the list and remove it. This is an example of the system being tailored towards scenarios where disconnections are the default rather than the exception.

As mentioned earlier, the references to the books are being associated with the list entries. This way, when a user double clicks on a list entry, a dialog box is shown in which the user can type a small review or some comments about the book. When accepting the input data of the dialog box, the application attempts to add the text the user just entered to the list of reviews associated on the book itself. As we showed earlier in section 5.3.5, invoking the `addReview` method on a book is a mutating operation (i.e. the method is tagged as a `Mutator`) which causes the book thing to be synchronized with its physical representation on the RFID tag. Notice that this write operation might not happen instantaneously because the RFID tag might be out of range for some time. The following code snippet shows the function that is called after the user wrote a comment in the dialog box we described above:

```
1 def addReviewToBook(book, text) {
2   when: book<-addReview(text)@Due(5.seconds) becomes: { |ack|
3     showOkDialog("Review added succesfully!");
4   } catch: TimeoutException using: { |exc|
5     showWarningDialog("Failed to add review!");
6   };
7 };
```

The dialog object passes the reference to the book and the user's text as arguments to the function shown above. This `addReviewToBook` function asynchronously sends the `addReview` message to the book via the far reference passed as an argument. The message is annotated with `@Due(5.seconds)` to indicate that if the message is not successfully processed after 5 seconds, a `TimeoutException` should be raised. The `when:becomes:catch:` observer installed on the future returned by the message send can trigger two blocks. The `becomes:` block is triggered when the message was successfully processed by the thing and *in addition* the mutated data was successfully written to the physical RFID tag (since the `addReview` method is a mutator). As mentioned earlier, within the 5 second timeout period, the RFID tag might have moved in and out of range for several times, but the underlying implementation of the language constructs keeps attempting to write the data until this timeout period has passed. If the timeout period passed without the review being successfully written on the tag, the `catch:` block of the observer is invoked. In response, the user can try again, maybe after repositioning himself closer to the book.

5.4 Implementation

In this section we detail the implementation of our ambient-oriented RFID framework in AmbientTalk. For our experiments, we used FEIG[2] ID ISC.PR101-USB 13.56 MHz proximity readers, high frequency desktop readers connected via USB. We used a variety of high frequency passive RFID tags, such as Philips I-Code1 and I-Code SLI tags (sticker-shaped), Philips MiFare Classic tags (contact cards) and Texas Instruments ISO tags (durable tags for harsh environments). Read range, the number of simultaneously detected tags and storage size vary greatly among these types of tags

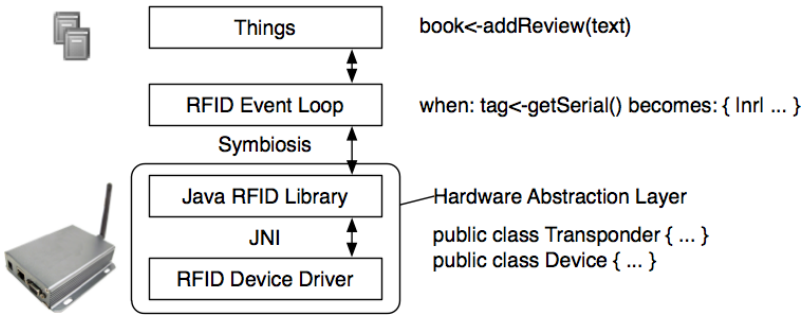

Figure 5.7: Implementation layers.

RFID Device Driver At the lowest layer shown in figure 5.7 we have a driver that allows to communicate with the RFID reader over a USB port. The driver provides only minimal functionality and provides operations to search available USB ports for the correct device, to open and close a given port and to read and write data (bytes) via the USB port.

Java RFID Library We implemented a Java library that implements vendor specifications for communicating with the RFID reader. The library interfaces with the driver using Java Native Interface (JNI). It provides classes such as `Transponder` and `Device` which represent the RFID tags and reader device. The methods of these classes implement classic RFID operations such as inventory all tags in range, read data from a tag or write data to a tag, and powering on and off the RF field of the reader. This is done by sending byte combinations via the driver to the RFID reader device. Note that the library (and driver) can be substituted by any custom built or vendor provided RFID library to support other RFID hardware, as long as the interface towards the RFID event loop is kept the same.

RFID Event Loop The RFID event loop keeps track of which tags are in range of the reader and keeps the data cached in the things it hosts up to date by continuously polling the RFID reader for an inventory. The RFID event loop does so by putting a request for an inventory in its own message queue by performing a message send to itself. The effect is that polling requests are interleaved with other operations the RFID reader needs to perform as messages from client applications enter the message queue. This way, race conditions in the execution of the event loop are prevented. Per new inventory the RFID event loop classifies the RFID tags based on their current and previous connectivity state. For

[2] `www.feig.de`

tags that for the first time appear, a new thing is created using the data on the tag (or from another data source as detailed in section 5.3.1). These new things are exported to other connected event loops. All remote references to tags that were previously connected (but are currently out of range) are disconnected, as detailed earlier in section 5.2.1. For tags that reappear (tags that were seen before and for which a thing already exists) the corresponding thing is updated and the remote references are reconnected as described in section 5.2.1. Per connected RFID reader one RFID event loop is needed.

The RFID event loop requires a minimal API from the Java library which is accessed using language symbiosis between AmbientTalk and Java. First, it relies on a class representing the RFID reader which should implement a method to perform an inventory of the tags in range. This class should also implement a method that allows the RFID event loop to instruct the RFID reader to keep its RF field permanently active (to obtain exclusive access to RFID tag memory during interaction) or switch it off after each new request, as explained earlier in section 5.3.6. Second, the presence of a class representing the RFID tags is assumed which has a read and write method to retrieve and store data associated with a tag. The implementation of this interface in the Java library layer determines how the data is actually stored, as described in 5.3.1. As mentioned earlier, we have experimented both with serializing AmbientTalk objects on tag memory itself and using an external database on which the read and write operation are performed.

Thing/Application Level The top level consists of applications such as our motivating library application. Their implementation is agnostic to the way various RFID event loops are implemented. The logic of these applications is solely expressed in terms of the appearance/disappearance of and messages to things.

5.4.1 Implementation of Things

In this section, we detail the implementation of things that read from and write to all their data (with which we mean their methods as well) onto the memory of RFID tags. Things reside only within the RFID event loop such that remote access can only happen through asynchronous message sends. The implementation of things heavily relies on the metaprogramming and reflection facilities of AmbientTalk, the which were discussed in the previous chapter in section 4.1.2. Figure 5.8 sketches the layered architecture of the example `Book` thing used throughout this chapter. The inner layer consists of the concrete object deserialized from the physical RFID tag's memory, inheriting from a tag object containing specific properties of the RFID tag (i.e. the serial number, the type of tag, its memory layout, a handle to the reader object that scanned it, the bare storage in bytes, and a number of other low-level properties).

It is wrapped by an object which has a custom mirror (i.e. a mirage). The main purpose of this mirror is to trap side effects to fields of the object it wraps and to trap methods invoked on the object it wraps tagged as mutator methods. These operations not only cause the wrapped object to change, but also schedule updates to the serialized object stored in the physical tag memory (which, of course, may fail).

The final wrapper is the thing itself to which far references from other event loops point. This layered architecture allows that the inner layer (containing the deserialized object) can be replaced with up-to-date versions when a new successful read of the memory of the tag associated with the thing occurs, and that additionally a new mirage

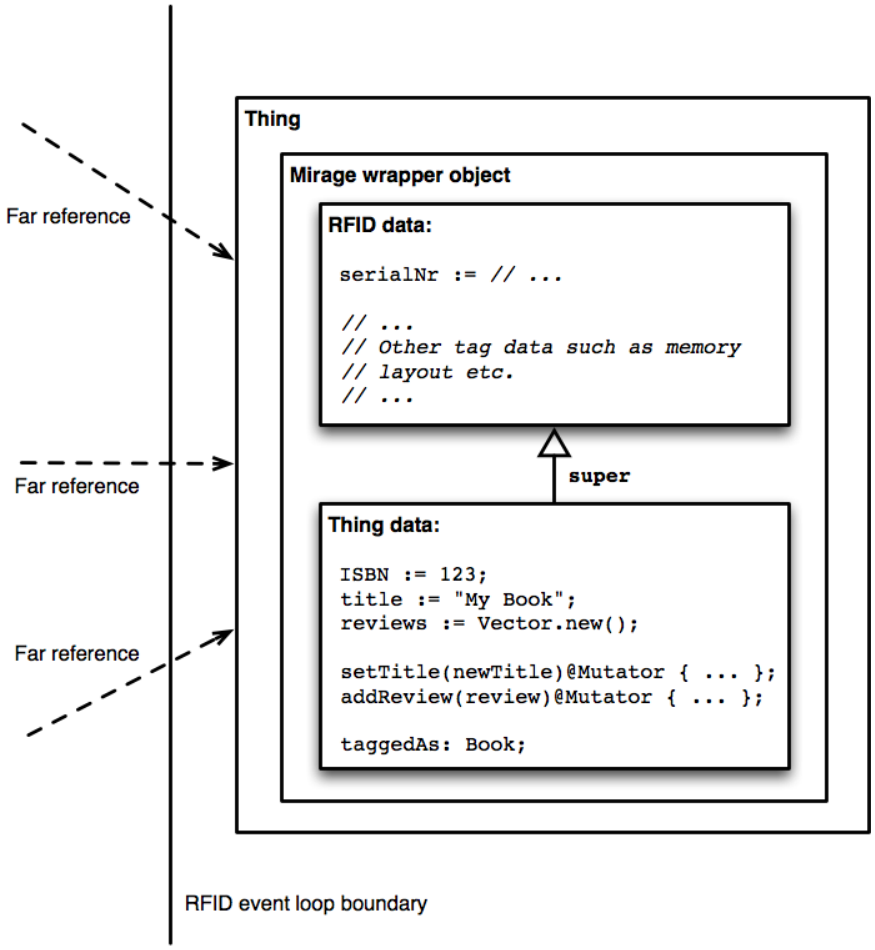

Figure 5.8: Layered architecture of the `Book` thing.

object with the custom mirror is created to wrap it (as mirrors on existing objects cannot be changed). All this can happen while keeping the surrounding thing wrapping the mirage the same. This is required to preserve the thing's identity in the face of subsequent reads and writes to the tag memory, that internally generate new objects.

Below, the implementation of the custom mirror used by the mirage object is shown.

```
1  deftype Mutator;
2  def IGNORE_INVOCATIONS := [`super, `==, `!=];
3
4  def ignore(invocation) {
5    IGNORE_INVOCATIONS.contains(invocation);
6  };
7
8  def mutatorsOf: someObject {
9    (reflect: someObject).listMethods.select: { |m|
10     is: m annotatedAs: Mutator;
11   };
12 };
13
14 def is: methodName mutatorOf: someObject {
15   try: {
16     (is: (reflect: someObject).grabMethod(methodName)
17      annotatedAs: Mutator).or: {
18       is: methodName mutatorOf: someObject.super; };
19   } catch: SelectorNotFound using: { |e|
20     try: {
21       is: methodName mutatorOf: someObject.super;
22     } catch: SelectorNotFound using: { |e|
23       false;
24     };
25   };
26 };
27
28 def makeMutatorInterceptor(interceptor) {
29   mirror: {
30     def invoke(slf, invocation) {
31       def returnValue := super^invoke(slf, invocation);
32       try: {
33         def methodName := invocation.selector;
34         if: (!ignore(invocation)) then: {
35           if: (is: methodName mutatorOf: slf) then: {
36             interceptor(self.base(), methodName);
37           } else: {
38             if: ((print: methodName) ~= ".*:=$") then: {
39               interceptor(self.base(), methodName);
40             };
41           };
42         };
43       } catch: SelectorNotFound using: { |e|
44         // do nothing
45       };
46       returnValue; // return original return value
47     };
48
49     def invokeField(slf, invocation) {
50       def returnValue := super^invokeField(slf, invocation);
51       try: {
52         def methodName := invocation;
53         if: (!ignore(invocation)) then: {
54           if: (is: methodName mutatorOf: slf) then: {
```

```
55              interceptor(self.base(), methodName);
56            };
57          };
58        } catch: SelectorNotFound using: { |e|
59          // do nothing
60        };
61        returnValue; // return original return value
62      };
63    };
64  };
```

First, it defines the `Mutator` type tag and a function that is used to filter out invocations that do not need to be trapped, namely `super`, `==`, and `!=`.

Subsequently, it defines two helper procedures to extract all methods tagged as `Mutator` from a given object (`mutatorsOf:`) and to test whether or not a method is a mutator for a certain object (`is:mutatorOf:`). This happens by using the metaprogramming interface of the mirror (obtained using the `reflect:` primitive) of the objects passed to these procedures.

Finally, a procedure `makeMutatorInterceptor` is defined that creates the actual mirror object. It takes a closure `interceptor` as argument, that is called whenever a mutator method is invoked. Remember from section 4.1.1 of the previous chapter that AmbientTalk unifies field accesses, side effects on fields and method invocations following the uniform access principle. Depending on how fields are accessed or set, or how zero-argument methods are called (e.g. the difference between `obj.msg` and `obj.msg()`), two different procedures of the metaprogramming interface of mirror objects are called. Hence, our custom mirror overrides both procedures: `invoke` and `invokeField` to trap field accesses, side effects on fields and method invocations. If such operations need to be trapped on the metalevel, the `interceptor` closure is applied, passing along the base object of the mirror and the method name.

As mentioned earlier, the RFID event loop wraps the objects deserialized from the tags' memory in a mirage object that uses the custom mirror of which the implementation is shown above. It uses the following closure as interceptor closure that will be applied when a side effect is performed on a field or when a mutator of the object is called:

```
1  def tagMutatorInterceptor := { |base, method|
2    if: ( method != `super:= ) then: {
3      try: {
4        saveToTag(base);
5      } catch: { |e|
6        theRFIDEventLoop<-saveToTag(base);
7      };
8    };
9  };
10
11 def makeTagMutatorInterceptorMirror() {
12   makeMutatorInterceptor(tagMutatorInterceptor);
13 };
```

It simply calls the `saveToTag` procedure with as argument the base object of the mirror wrapping the real object to be stored on the RFID tag. This procedure extracts the wrapped object and uses the underlying Java API to serialize the wrapped object and physically write it on the associated tag's memory using the `asCode:` primitive explained in section 5.2.3. The resulting string representation of the object is zipped before writing it on the tag's memory. In case this fails, the same method call is asyn-

chronously scheduled in the RFID event loop's message queue where it will – after having processed other pending requests – be re-executed. This procedure internally keeps track of how long ago it was executed for the first time. If this exceeds the time-out value used by the event loop, an asynchronous exception is raised such that client code is asynchronously notified that the write operation failed.

Read operations on a thing do not have to be trapped, they can simply use the data cached in the thing. The reason is that when an RFID-tagged object is in range of a reader device, the reader device has exclusive access to this tag and no concurrent writes can occur. This means that updating the cached data encapsulated by the thing can happen when a disconnection followed by a reconnection of the tagged object is observed. Only in this timespan where the tagged object was disconnected, other devices might have updated the object stored on the tag. This process of generating and maintaining things is discussed in the section below.

5.4.2 Generating and Maintaining Things

Now that we have explained the wrapped architecture of things, we discuss a final important snippet of the implementation of the RFID event loop. The `inventory` procedure shown below is responsible for periodically scanning the environment for RFID tags (performing an *inventory*) and synchronizing the state of the things encapsulated in the RFID event loop in response.

```
1  def inventory() {
2    def tags := Vector.new(device.inventory(true));
3    def previouslyConnected := serialToTag.clone();
4
5    tags.each: { |tag|
6      def serial := tag.getSerial().toString();
7      previouslyConnected.remove(serial);
8      if: (serialToTag.contains(serial)) then: {
9        // Tag previously discovered.
10       try: {
11         // Tag is reconnected, update thing object.
12         if: (disconnectedTags.contains(serial)) then: {
13           def disconnectedTag := disconnectedTags.get(serial);
14           disconnectedTag.reconnect();
15           disconnectedTags.remove(serial);
16         };
17         def newObject := load(tag);
18         // Safety check if reading goes wrong.
19         if: (nil != newObject) then: {
20           updateThingObject(tag, newObject);
21         };
22       } catch: { |e|
23         // Failed: tag disconnected again.
24         if: (!(disconnectedTags.contains(serial))) then: {
25           def disconnectedTag := disconnect: tagToObject.get(serial);
26           disconnectedTags.put(serial, disconnectedTag);
27         };
28       };
29     } else: {
30       // Tag discovered for the first time.
31       try: {
32         // Generate new thing object.
33         def tagObject := load(tag);
34         def s := serial;
35         if: (nil == tagObject) then: {
36           // Blank tag, export as such.
```

```
37            tagObject := GenericTag.new(tag);
38            def wrapper := wrapAndPlaceMutatorInterceptorMirror(tagObject);
39            createThingObject(tag, wrapper);
40            export: wrapper as: RFIDTag with: {
41              def serial := s;
42              def equivalent(otherRef) { self.serial == otherRef.serialNr };
43            };
44          } else: {
45            // Initialized tag.
46            def wrapper := wrapAndPlaceMutatorInterceptorMirror(tagObject);
47            createThingObject(tag, wrapper);
48            (reflect: wrapper).typetags.each: { |typetag|
49              export: wrapper as: typetag with: {
50                def serial := s;
51                def equivalent(otherRef) { self.serial == otherRef.serialNr };
52              };
53            };
54          };
55        } catch: { |e|
56          // Failed: tag disconnected again.
57          if: (!(disconnectedTags.contains(serial))) then: {
58            def disconnectedTag := disconnect: tagToObject.get(serial);
59            disconnectedTags.put(serial, disconnectedTag);
60          };
61        };
62      };
63    };
64
65    previouslyConnected.each: { |serial, tag|
66      if: (!(disconnectedTags.contains(serial))) then: {
67        def disconnectedTag := disconnect: tagToObject.get(serial);
68        disconnectedTags.put(serial, discoTag);
69      };
70    };
71
72    when: POLLING_INTERVAL elapsed: { self<-inventory() };
73  };
```

The inventory method first stores all scanned RFID tags in the `tags` vector. Because `true` is passed to the `inventory` method of the Java object representing the physical RFID device, tags are kept powered during the inventory process and grant exclusive access to the device. The drawback is that more power is used to keep the RF field active and the tags in range powered.

Given these tags in range, the procedure will recompute the tags that were previously in range and are now disconnected. They are stored in the `previouslyConnected` hash map, which is a clone of the `serialToTag` hash map that maps serial numbers to tag objects. Given the tags in range and the tags that became disconnected, it will generate new thing objects for newly discovered tags, update the thing objects for the tags that are still in range (using the mechanism described in the previous section), and signal disconnections for things of which the associated tags are not in range anymore after the previous inventory. This happens by iterating over all scanned tags in the inventory (lines 5-73). If a tag is in the list of tags returned by the inventory operation, it means it is currently in range and can be removed from the `previouslyConnected` map (line 7).

If it is in the `serialToTag` map, it means it was previously discovered (line 8). Disconnected tags are stored in the `disconnectedTags` hash map. In this case, the tag is back in range and can be removed from `disconnectedTags` and be reconnected

(which will change all connected far references' state to the connected state and signal the appropriate events) (lines 13-14). Additionally, since the tag is in range, an attempt is made to update the cached thing by loading the memory contents of the tag (line 17). If this raises an exception, the tag went out of range, it must be disconnected (signaling the appropriate events to all connected far references) again and be added to the `disconnectedTags` map again (lines 24-27).

If the the tag was not in the `serialToTag` map, it means it is discovered for the first time (line 29). After loading the object stored on the tag, two things can happen. Either the tag contained no data (line 35), or contained a previously stored serialized object (line 44). In the first case, a blank tag object is created and is exported as such. The `export:as:with` construct is part of the property references abstraction that was discussed in section 5.2.2. In the other case, a thing is generated using the object deserialized from the tag's memory, and exported under each of its type tags (which are obtained through the metalevel interface of the object) (line 47-53). All these operations may fail because the tag moved out of range while performing them. In that case, an exception is raised, which causes the thing to be disconnected again and put in the `disconnectedTags` map again (ready for the next inventory round) (line 55-61).

Subsequently, now that the currently connected tags are known, their thing objects are updated and the tags that were previously connected but now disconnected are known, we iterate over the latter and disconnect them all and put them in the `disconnectedTags` map (just like in the exception handlers of the code explained above) (line 65-70).

Finally, after a configurable polling time interval, the `inventory` procedure is scheduled again for execution by asynchronously sending the corresponding message to the RFID event loop (line 72). This will cause the message to be scheduled in its message queue and be re-executed when other pending requests are handled.

5.4.3 Performance Evaluation

In this section, we present the results of simulations that measure the computational overhead that comes with using our constructs for developing RFID-enabled applications. These simulations were ran on an Apple Macbook Pro laptop with an Intel Core i7 2.66 Ghz processor and 4 Gb of RAM running OSX version 10.6.7 and Java SE version 1.6.0. The version of AmbientTalk that we used is build 2.19.1. Our simulation is based on a simulated RFID reader device, i.e. a Java object able to generate "dummy" transponder objects. These dummy transponder objects allow reading and writing their single data field that represents the data stored on the simulated tag as a byte sequence.

We conducted two simulations: one for read operations and one for write operations, which we implemented by using both our constructs as by using a plain Java implementation without taking into account the requirements for ambient-oriented programming and mobile RFID-enabled applications.

For the read operations, we let the simulated RFID reader perform 100 inventory operations, producing a collection of tag objects and we read the data of every single tag object after each inventory operation. We vary the number of tags from 10 to 100 per inventory operation and measure how much time the complete simulation takes to finish.

For the write operations, we again start from a collection of tag objects, but this time we perform a write operation on every single tag and measure how long this simulation

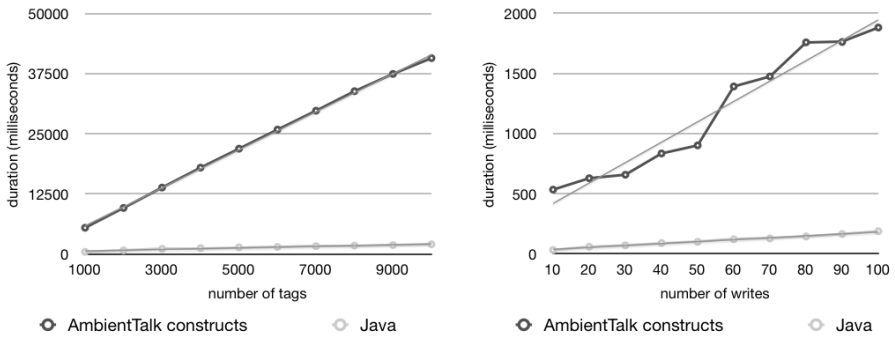

Figure 5.9: Simulation results of read and write operations.

takes. For both operations, we also varied the amount of data read from and written to the tag objects, but this had very little effect on the results, so we omit these results.

The results comparing the Java version and the version using our constructs are depicted by the two graphs shown in figure 5.9.

Without comparing both implementations, it is clear that both scale linearly when increasing the number of tags. When comparing both implementations, we can see that there is an overhead of a factor 10 to 20 depending on the operations performed. Although this seems excessively high, one must take into account that the version relying on our AmbientTalk constructs is generating, garbage collecting and updating the connectivity status of things, (de)serializing things, enqueueing and processing asynchronous message sends (sequentially) to provide fault tolerance, etc. The basic Java implementation assumes that not a single fault occurs in these simulation runs. We measured where the biggest overhead in our implementation occurs and found that the creation of things, their serialization and deserialization and the exporting and asynchronously discovering of references to these things contribute the most to the observed slowdowns.

It is more interesting to compare the the performance figures of our implementation with the read rates (in tags per second) that modern RFID hardware can provide. Large industrial ultra-high frequency RFID readers offer in ideal conditions a read rate of 200 tags per second if 100% reliable reads are necessary. If faulty reads are tolerable, about 450 tags per second can be read [LL10]. As can be be seen from the two graphs shown in figure 5.10 (the left graph shows read rates while the right graph shows write rates), read and write rates do not decrease while increasing the number of tags. Additionally, our implementation is able to read (see left graph) slightly more than 200 tags per second. This means that in practice, the performance gap between the two implementations will be barely noticed due to the delay caused by real RFID hardware.

We conclude from these benchmarks that our implementation can keep up with such a high-performance RFID reader and hence be applied to realistic systems, although we are targeting a different class of applications that consist of mobile, less performant devices. Finally, both the implementation of the AmbientTalk language as well as our RFID language constructs are unoptimized research artifacts that we expect to be further optimizable (e.g. by optimizing the serialization mechanism).

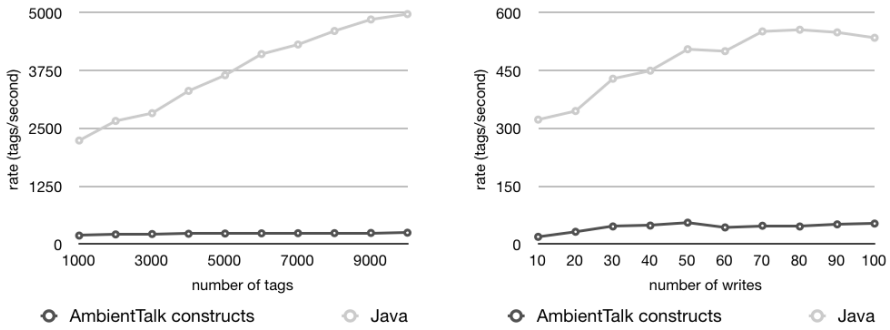

Figure 5.10: Read rates and write rates (tags per second).

5.4.4 Limitations

An issue that is not specific to our support for RFID, but to AmbientTalk programs in general, is that messages acknowledging the successful processing of asynchronous messages may time out while the corresponding asynchronous message was successfully executed. This means that in our case, client event loops will signal an exception, while the RFID event loop actually may have processed a request successfully. In case of writing data to an RFID tag, this leaves the application in an inconsistent state, as the client event loop incorrectly assumes that no changes were written onto the RFID tag's memory. However, it is important to note that this can of course only happen if the RFID event loop is hosted on another device. In the typical case, where the RFID event loop is used as an abstraction to interact with on-board RFID hardware, this problem does not occur. Mitigating this problem can be done by rolling back side effects when the RFID event loop is signaled an exception that the acknowledgement message timed out. Still, this is no water proof solution since the affected RFID tag may be out of range by that time or may have been read by other devices. Solving this issue requires transactional semantics over decentralized, unreliable interactions, which is a problem that is out of the scope of this dissertation.

Event loops consume their incoming messages sequentially. This means that no objects are shared between different event loops and race conditions cannot occur. A first limitation of this approach is that the fact that tags are processed sequentially restricts our implementation to systems where RFID readers do not have to process a massive amount of tags in real time. A second limitation is that, when we consider RFID tags as an ambient environmental memory, it may very well be that a set of RFID tags is in the range of multiple users at the same time. Currently, we offer a way to guarantee RFID tags are made invisible to other RFID devices when interacting with them (as discussed in section 5.3.6), but it requires making the set of tags that is interacting with a single RFID reader invisible in its entirety to other RFID readers. When this mechanism is switched off, tags are visible to different RFID devices during interaction. In this case, when users concurrently update the same tag from different devices, distributed race conditions on that tag may occur.

In this work, we have not considered security and privacy. Currently, RFID technology is considered problematic for applications where security and privacy is are important concerns as no workable solutions exist to offer complete security [RK09]. Similarly, we have only considered failures that are caused by RFID tags becoming

unreachable or unresponsive altogether. Problems that cause RFID hardware to malfunction in any other way were not considered.

A final limitation is the limited amount of writable memory on passive RFID tags. We have tested our implementation using RFID tags with up to 8 kbits of writable memory. This means that we can only store very small AmbientTalk objects on the tags. On the other hand, the technology is progressing and we can expect the storage on passive tags to steadily increase while the costs drop. One can now for example buy passive RFID tags with 32 Kb of writable memory [PS11]. This opens the door to use more standardized serialization formats as well, which we have not considered at the moment to cater to our prototype-based object model.

5.5 Conclusion

The abstractions presented in this chapter integrate closely with the object-oriented message passing paradigm and the ambient-oriented programming paradigm, thereby aligning physical objects tagged with writable RFID tags with true mutable software objects.

	Object	**Isolate**	**Thing**
Scope	Object and lexical	Object only	Object only
Distributed parameter passing	By far reference	By copy	By far reference
Remote messaging	Asynchronous	N/A	Asynchronous
Default identity	Reference-based	None	Tag-based
Side effects	On referenced object	On local copy	On referenced thing and RFID tag's memory (requires `Mutator` method annotation).

Table 5.1: AmbientTalk's extended object model for mobile RFID-enabled applications.

By implementing an example mobile RFID-enabled application, we have observed that the requirements that we set forward for programming mobile RFID-enabled applications are met in the following ways:

Addressing physical objects. The implementation of the application shows that mobile RFID-enabled applications can be written in an object-oriented fashion, where application-level "thing" objects uniquely represent physical objects in one's physical environment.

Storing application-specific data on RFID tags. The data needed to construct these things is stored on the RFID tags themselves.

Reactivity to appearing and disappearing things. Application logic is expressed in terms of reactions to changes in the physical environment by relying on a number of abstractions that are integrated into a communicating event loops framework.

Asynchronous communication. Interacting with physical objects is achieved by using the message passing metaphor on things, by means of asynchronous message passing and asynchronous signaling of return values.

Fault-tolerant communication. Communication failures are considered the rule rather than the exception. Failures that must be considered permanent are detected and raise the appropriate exceptions.

Data consistency. By allowing the mobile device equipped with an RFID reader to temporarily make its reachable tags invisible to other mobile devices (by keeping them powered in the RF field), exclusive access for that mobile device can be granted and data consistency guaranteed, in combination with the consistency guarantees of the event loop model among local applications for the device. For the moment, we have not considered security and this remains future work.

In short, in this chapter we have successfully mapped interactions with (passive) RFID technology onto the object-oriented programming model and event loop concurrency model of AmbientTalk. To support the requirements listed above, we have extended AmbientTalk's object model – consisting of normal objects and isolates – with things. How these three types of objects are treated in the language is compared in table 5.1.

Note that although multiple copies of the same thing may exist in an environment with multiple RFID-enabled devices that scan the same tags, the implementation makes it look like as if conceptually there is only one instance of the thing.

Still, by relying on the classic event handling constructs of AmbientTalk (see chapter 4 section 4.2), mobile RFID-enabled applications are prime examples of event-driven applications that without dedicated event handling constructs suffer from inversion of control. In the next chapter, we take care of the inversion of control problem by using dataflow programming techniques for dealing with the events that drive mobile RFID-enabled applications.

Chapter 6

Node-Centric Ambient-Oriented Dataflow Programming

In this chapter, we introduce a dataflow variant of AmbientTalk, called AmbientTalk/R. Recall from the conclusion of chapter 4 that AmbientTalk/2 applications must be structured as decentralized event-driven architectures, but suffer from the inversion of control problem inherent to classic event-driven programming because reacting to events happens by means of explicit callbacks. AmbientTalk/R's interpreter automatically tracks dataflow dependencies to support *reactive programming*, a variant of dataflow programming based on time-varying values discussed earlier in chapter 2 section 3.3.1. However, reactive programming is restricted to local AmbientTalk/R programs, which implies that the programmer must take care of distributing reactive code without dedicated support.

It is on this dataflow version of AmbientTalk that we build primitives for node-centric ambient-oriented dataflow programming. Additionally, we also introduce group-centric variants of these primitives. By relying on an underlying broadcast-based publish/subscribe architecture, it only depends on the subscription whether events from a single event producer or multiple event producers in the mobile ad hoc network are received. These event producers can vary over time, which must be reflected on the event consumer's side. Events and connectivity changes are transformed to value changes of AmbientTalk/R's time-varying values of which the dataflow dependencies are automatically tracked. This means that the programmer is now less concerned with coordinating control flow among different callbacks or event handlers, because this is being taken care of by the language runtime.

First, we start in section 6.1 by giving two motivating examples and point out their differences. The first example is an ambient-oriented application that will subsequently be used to explain AmbientTalk/R's dataflow language constructs in section 6.2. The second example is the mobile RFID-enabled application that we are using as a case study throughout this dissertation. In section 6.3 we continue our first motivating example in a distributed dataflow style thanks to a number of distributed dataflow and group-centric primitives. Section 6.4 shows the node-centric ambient oriented dataflow implementation of our mobile RFID-enabled application case study. Section 6.5 discusses the implementation of the distributed dataflow language constructs introduced

in this chapter and subsequently benchmarks it by comparing its performance to a handcrafted solution in standard AmbientTalk. After that, section 6.6 discusses their limitations. Finally, section 6.7 concludes this chapter.

6.1 Motivation

Before moving on to our mobile RFID-enabled application that we are using as a running example, in this section we first show that distributed dataflow programming is useful for general ambient-oriented applications as well. We use a simple ambient-oriented example application that is loosely inspired by the "ubiquitous flea market" application [EGH05] as a motivating example for adopting an ambient-oriented dataflow language. We identify the different types of events and how they must be handled in a traditional event-driven style. Although we restrict our discourse to mobile RFID-enabled applications, this shows that these constructs are useful for other sensor-driven ambient-oriented applications as well.

Subsequently, we do the same for the mobile RFID-enabled library application used as a case study. This allows us to compare both applications and, after implementing both applications in a node-centric ambient-oriented dataflow style, compare to which extent our programming model requirements listed in chapter 2 section 2.3.4 are met.

6.1.1 The Ticket Trader Application

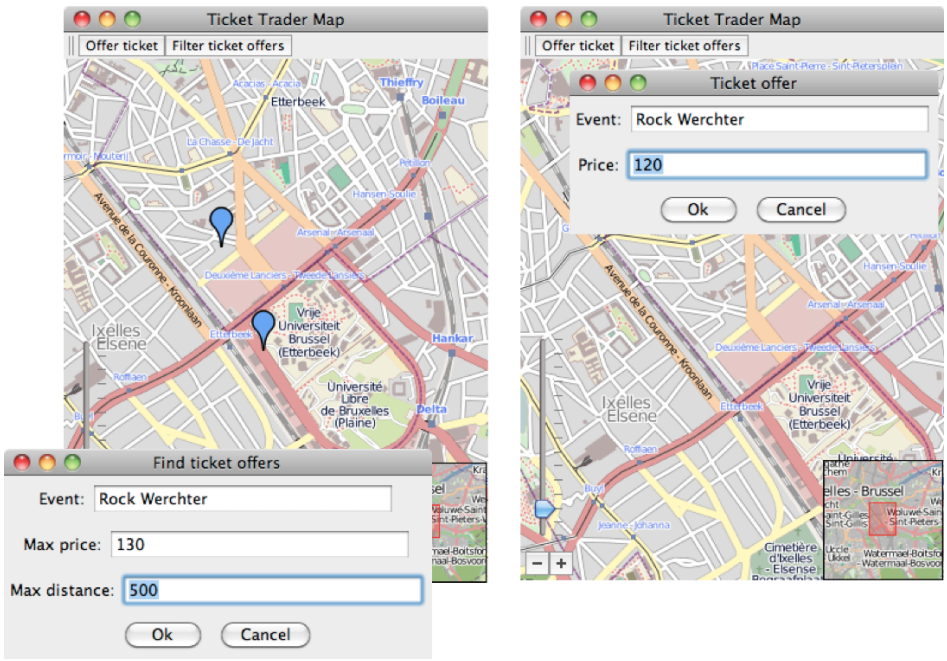

Figure 6.1: Two running instances of the ticket trader application.

In the ticket trader application, users are able to offer tickets for specific events for sale (e.g. close to the location of the venue). They can adjust the price at any time or

cancel the ticket for sale. Other users are able to search for tickets that they want to buy, taking into account both price limits and distance limits. All users have a maps application (based on OpenStreetMaps[1] in the current version) that shows tickets that are for sale that fit in their search criteria. When the users selling these tickets move about, their location is automatically updated on all potential buyers' maps. Figure 6.1 below shows two running instances of the Ticket Trader application.

Using the classic event-driven programming style of AmbientTalk, the notification messages and the callbacks shown that are needed to signal and react on these events are shown in table 6.1.

Event	**Event producer**	**Event consumer**
New ticket trader connected	Automatic notification by AmbientTalk.	`whenever:discovered:` callback
Ticket trader disconnected	Automatic notification by AmbientTalk.	`when:disconnected:` callback
Ticket trader reconnected	Automatic notification by AmbientTalk.	`when:reconnected:` callback
New ticket for sale	`notifyTicketForSale` asynchronous message send	`notifyTicketForSale` method invoked when receiving corresponding asynchronous message.
Price of ticket changed	`notifyTicketPriceChanged` asynchronous message send	`notifyTicketPriceChanged` method invoked when receiving corresponding asynchronous message.
Location of ticket trader selling interesting ticket changed	`notifyTicketTraderLocationChanged` asynchronous message send	`notifyTicketTraderLocationChanged` method invoked when receiving corresponding asynchronous message.
Own location changed	GPS abstraction invokes listener callback with new coordinates.	Callback registered on GPS abstraction.

Table 6.1: Traditional event handling in the Ticket Trader application.

6.1.2 The Book Recommender Application

The Book Recommender is an application that users can launch on their smartphone when they enter a library in order to help them find interesting books based on recommendations provided. This happens as follows. Books are tagged with RFID tags. These RFID-tagged books can be taken to a book scanner equipped with an RFID reader (for example integrated into an information desk). The reader will detect which books are carried around by the user, and will contact the Book Recommender application running on his smartphone to update the list of books that the user is carrying.

[1] http://www.openstreetmap.org/

Event	Event producer	Event consumer
Book scanner scanned new book thing	Automatic notification by RFID event loop.	`whenever:discovered:` callback.
Book thing out of book scanner's range	Automatic notification by RFID event loop.	`when:disconnected:` callback.
Book thing back into book scanner's range	Automatic notification by RFID event loop.	`when:reconnected:` callback.
Shelf scanned new book thing	Automatic notification by RFID event loop.	`whenever:discovered:` callback.
Book thing out of shelf's range	Automatic notification by RFID event loop.	`when:disconnected:` callback.
Book thing back into shelf's range	Automatic notification by RFID event loop.	`when:reconnected:` callback.

Table 6.2: Traditional event handling in the Book Recommender application: RFID connection status events.

Event	Event producer	Event consumer
New set of scanned books to show in Book Recommender GUI	Book scanner sends asynchronous `notifyNewScannedbooks` message.	`notifyNewScannedbooks` invoked on Book Recommender application GUI when corresponding message received.
New set of recommended books to show in Book Recommender GUI	e-book reader sends asynchronous `notifyNewRecommendedBooks` message.	`notifyNewRecommendedBooks` invoked on Book Recommender application GUI when corresponding message received.
New set of recommended books in stock in the library to show in Book Recommender GUI	Library shelf/shelves send(s) asynchronous `notifyNewRecommendedInStock` message.	`notifyNewRecommendedInStock` invoked on Book Recommender application GUI when corresponding message received.
New set of scanned books for e-book reader to process	Book scanner sends asynchronous `notifyNewScannedbooks` message.	`notifyNewScannedbooks` invoked on e-book reader when corresponding message received.
New set of recommended books to filter by shelf	e-book reader sends asynchronous `notifyNewRecommendedBooks` message.	`notifyNewRecommendedBooks` invoked on shelf when corresponding message received.

Table 6.3: Traditional event handling in the Book Recommender application: application-specific events.

Event	Event producer	Event consumer
Book Recommender GUI in range of book scanner	Automatic notification by AmbientTalk.	`whenever:discovered:` callback.
Book Recommender GUI disconnected with book scanner	Automatic notification by AmbientTalk.	`when:disconnected:` callback.
Book Recommender GUI reconnected with book scanner	Automatic notification by AmbientTalk.	`when:reconnected:` callback.
Book Recommender GUI in range of e-book reader	Automatic notification by AmbientTalk.	`whenever:discovered:` callback.
Book Recommender GUI disconnected with e-book reader	Automatic notification by AmbientTalk.	`when:disconnected:` callback.
Book Recommender GUI reconnected with e-book reader	Automatic notification by AmbientTalk.	`when:reconnected:` callback.
Book Recommender GUI in range of shelf	Automatic notification by AmbientTalk.	`whenever:discovered:` callback.
Book Recommender GUI disconnected with shelf	Automatic notification by AmbientTalk.	`when:disconnected:` callback.
Book Recommender GUI reconnected with shelf	Automatic notification by AmbientTalk.	`when:reconnected:` callback.
E-book reader in range of book scanner	Automatic notification by AmbientTalk.	`whenever:discovered:` callback.
E-book reader disconnected with book scanner	Automatic notification by AmbientTalk.	`when:disconnected:` callback.
E-book reader reconnected with book scanner	Automatic notification by AmbientTalk.	`when:reconnected:` callback.
E-book reader in range of shelf	Automatic notification by AmbientTalk.	`whenever:discovered:` callback.
E-book reader disconnected with shelf	Automatic notification by AmbientTalk.	`when:disconnected:` callback.
E-book reader reconnected with shelf	Automatic notification by AmbientTalk.	`when:reconnected:` callback.

Table 6.4: Traditional event handling in the Book Recommender application: connection status events.

Additionally, if the user is carrying a tablet PC, or e-book reader, or another similar device, the reader will send it the same list of books. In response, the device will contact the user's digital book store (e.g. Amazon, iTunes...) of choice and look up a list of similar books to the ones the user is carrying. Once this list is generated, it will broadcast this list to all reachable shelves in the book store. These shelves also have an internal RFID reader that by periodically scanning the entire shelf knows which books are present in the shelf and which ones are not. Using this information, the shelves filter out of the list of recommended books the ones that are present in the shelf, and send these lists to the user's Book Recommender application. In response, the application shows a list of recommended books that are present in the library, together with on which shelf they can be found.

At any time, users can take books out of the shelves and put books back in. Furthermore, book recommendations should only be sent to the relevant users in the shop.

Just like we did in the previous section for the Ticket Trader application, we list the events that must be handled in the application. We split them up into three tables: table 6.2 lists the RFID connection status events, table 6.3 the application-specific events, and finally table 6.4 the connection status events.

Note that for the RFID-tagged books represented as things we are employing a simplified scenario: we assume that book things do not change. If this would have been the case, they should be encapsulated in a separate RFID event loop, and instead of passing things around directly, far references to these things should be passed around. This would further increase the event-driven nature of the application as communication over far references can only happen asynchronously.

6.1.3 Conclusion

In this section, we have described two ambient-oriented application scenarios and identified the different events that have to be communicated and reacted on. In chapter 4 section 4.2, we concluded that such applications structured as traditional event-driven architectures in which events must be explicitly handled by callbacks imply inversion of control.

Comparing the two applications, the mobile RFID-enabled Book Recommender application is clearly more complicated. In contrast to the Ticket Trader application's homogenous peers, it consists of various heterogeneous distributed application components that each generate and react on distinct events.

Therefore, before moving on to the more complicated Book Recommender application used as a case study in this dissertation, we first use the more basic Ticket Trader application explained above to introduce our node-centric ambient-oriented dataflow constructs. Afterwards, we turn our attention back to our mobile RFID-enabled application and apply the same constructs to implement it in an ambient-oriented dataflow programming style in section 6.4. This allows us to evaluate if the requirements listed in chapter 2 section 2.3.4 are met.

6.2 Dataflow Programming in AmbientTalk/R

As a first introduction to dataflow programming in AmbientTalk/R, consider the following example: assume that the ticket trader application introduced in the previous section sports a user interface which includes a map that shows the position of ticket vendors. The user can use this map to seek out a vendor and purchase one of the tickets

being offered. To facilitate navigation, the map must be centered on the user's current position. Hence, whenever the position of the user changes, the user interface should be updated. The code excerpt below illustrates that using dataflow programming, such behavior can be achieved without registering event handlers or suffering from inversion of control.

```
1 gui.centerOn(GPS_Location.latitude, GPS_Location.longitude);
```

In the above code excerpt, it is assumed that `GPS_Location` is a reactive (or time-varying) value (in the rest of this chapter we will underline variables containing non-derived reactive values) that represents the user's current location. Later in this section, we will illustrate how to construct such a reactive value by means of a built-in GPS location sensor. Given the `GPS_Location`, dependent reactive values are created implicitly when accessing its `latitude` and `longitude` fields respectively. These reactive values are recomputed (i.e. the respective fields will be read anew) automatically whenever the `GPS_Location` is updated. In turn, the reactive values representing the user's current latitude and longitude are used as arguments to the invocation of the `centerOn` method. This method invocation is lifted by the interpreter, resulting in the construction of a reactive value which depends on both reactive arguments. Hence, when either one of the arguments changes, the method will be invoked anew with the updated arguments. The dataflow graph that corresponds to this code snippet is shown in figure 6.2.

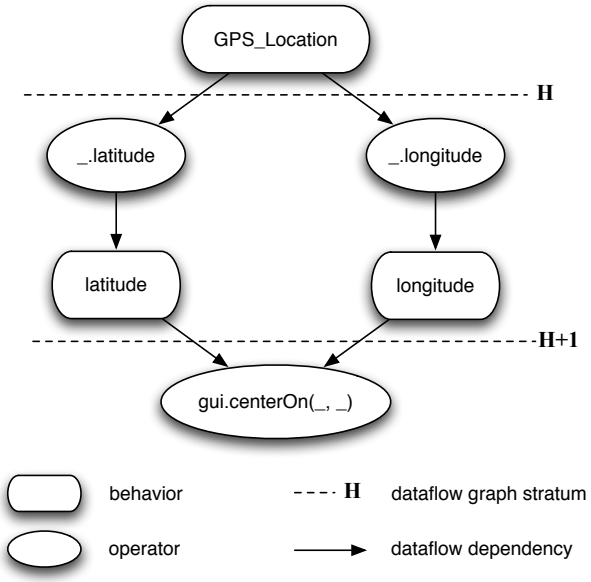

Figure 6.2: Dataflow graph for centering the map in the ticket trader application.

The graph shows the reactive value `GPS_Location` which acts as the *progenitor* for two dependent reactive values which represent the `latitude` and `longitude`. In turn, these reactive values are the progenitors for the reactive value that – as a side-effect – centers the user interface on the user's current location. Furthermore, figure 6.2 shows how the dataflow graph is partitioned into different layers or *strata* such that a reactive value only depends on reactive values situated in a lower stratum. This

stratification (first proposed in [CK06]) is used when propagating the updates to ensure that a reactive value is recomputed only when all of its progenitors have been updated. For instance, when the user's current position changes, the interpreter will first update both the `latitude` and the `longitude` reactive values before the `centerOn` method will be invoked anew.

Having explained how dependent reactive values are created implicitly by the interpreter in a reactive program, we now describe how to create new reactive values *ex nihilo*. For this, AmbientTalk/R introduces the `makeReactive` construct which creates a reactive value based on the object it is passed. In the code example given below, we define a `Coordinate` isolate object which represents a GPS location. In addition to its fields, `Coordinate` objects also define two methods, to wit `distanceTo` and `update`.

```
1  def Coordinate := isolate: {
2    def latitude := 0;
3    def longitude := 0;
4
5    def distanceTo(anotherCoordinate) {
6      /* Compute via the Haversine formula */
7    };
8
9    def update(newLatitude, newLongitude)@Mutator {
10     latitude  := newLatitude;
11     longitude := newLongitude;
12   };
13 };
14
15 def GPS_Location := makeReactive(Coordinate.new());
16 GPS.addLocationObserver: { |lat, lng| GPS_Location.update(lat, lng) };
```

In the example above, on line 15 a reactive value denoting a time-varying `Coordinate` object is created. When passing an object to the `makeReactive` construct, a clear distinction should be made between accessor methods which only read the state encapsulated by the object (e.g. `distanceTo`) and mutator methods which can change the object's internal state (e.g. `update`). All mutator methods must be explicitly tagged with an `@Mutator` annotation. This requirement stems from the fact that the semantics for invoking both kinds of methods on a reactive value differs significantly:

Accessor methods When invoking an accessor method (or reading a field) on a reactive value, *a dependent reactive value* is created and returned which depends both on the receiver and on any reactive values that were passed as arguments. Hence, if the reactive value is updated, the accessor method will be performed anew and the returned reactive value is updated when the new return value is different from the previous one. The same thing can happen in response to a change to one of the arguments of the method invocation if this argument is a reactive value. Reading fields of a reactive value behaves the same as invoking accessor methods (following the uniform access principle of AmbientTalk, as explained in chapter 4 section 4.1.1).

Mutator methods When invoking a mutator method (or writing a field), no dependency on the receiver is recorded (to prevent ending up in an endless reaction). In other words, if none of the arguments of the method invocation are reactive values, the method is simply performed once. If at least one reactive value was passed as an argument, a dependent reactive value is created which ensures that the mutator method is invoked anew whenever the reactive arguments change. However, changes to the receiver performed in the method body are disregarded,

again to prevent infinite reactions. Furthermore, the interpreter ensures that whenever a mutator method has been invoked, all dependents of the receiver are notified that their progenitor has been updated. Writing fields of a reactive value behaves the same as invoking mutator methods (following the uniform access principle of AmbientTalk, as explained in chapter 4 section 4.1.1).

This semantics is used in line 16 of the code example to register a location observer with the GPS device, which is automatically invoked whenever the user's position has to be updated. At this point in time, the mutator method `update` is invoked on the reactive object `GPS_Location`. This mutator method is invoked once (since its arguments are ordinary numeric values), updating the coordinates to reflect the most recent sensor values. Afterwards, all reactive values which implicitly depend on `GPS_Location` are notified that the location has changed. This may result for instance in an update of the user interface, such that the map is centered on the user's updated position. Note that mutator methods do not need to be atomic to guarantee correctness. The stratification of the dataflow graph constructed by the interpreter (explained earlier in this section) in combination with all updates that are scheduled in a single event loop according to this stratification prevent local concurrency control problems and ensure that the ordering of updates to reactive values mirrors the call graph of the program (which is critical when reactive updates trigger side-effects).

6.2.1 Reactive Object Semantics

In table 6.5, we give an overview of turning AmbientTalk/R's objects reactive. We incorporate things denoting RFID-tagged object from the previous chapter.

As can be seen from this table, reactive values that are parameter-passed to another event loop are turned into non-reactive values, more specifically: far references (to the original reactive value) and snapshot copies of isolates. There is a reason for this. Distributing dataflow graphs to track distributed dataflow dependencies among parameter-passed reactive values causes a tight coupling among distributed application components. However, by not tracking these dependencies across event loops, the programmer must manually poll for changes to remote reactive values, leading to a traditional event-driven style which suffers from inversion of control. Therefore, in the next section we introduce a number of constructs that allow ambient-oriented dataflow programming.

6.3 Ambient-Oriented Dataflow Programming in AmbientTalk/R

The dataflow programming system described in the previous section only deals with events in a single, local event loop. In many cases, distributed application components are interested in events coming from other devices (and thus event loops) in the mobile ad hoc network. In this section, we introduce a language construct called *ambient behaviors* that allows the loosely-coupled propagation of events to reactive values hosted on different event loops by means of publish/subscribe.

The transition from local reactive values to ambient behaviors needs some special consideration in order to uphold the ambient-oriented programming characteristics listed in chapter 2 section 2.3.4. Because of the dynamic nature of mobile ad hoc

	Object	**Isolate**	**Thing**
Scope	Object and lexical	Object only	Object only
Distributed parameter passing	By far reference	By copy	By far reference
Remote messaging	Asynchronous	N/A	Asynchronous
Default identity	Reference-based	None	Tag-based
Side effects	On referenced object	On local copy	On referenced thing and RFID tag's memory (requires `Mutator` method annotation).
makeReactive	Returns reactive object that changes when methods annotated with `Mutator` are invoked.	Returns reactive isolate that changes when methods annotated with `Mutator` are invoked.	Returns reactive thing that changes when methods annotated with `Mutator` are invoked, and additionally the changes are written onto the RFID tag.
Distributed parameter passing of reactive value	Far reference to reactive value (must be manually polled).	Non-reactive copy (snapshot).	Far reference to reactive value (must be manually polled).

Table 6.5: AmbientTalk/R's reactive objects semantics.

networks, one cannot assume a stable dataflow graph as is constructed on the local interpreter level, such as explained in the previous section. Instead, the dataflow dependencies between different distributed computations should be established in a loosely coupled way.

When we map behaviors onto event producers and dependent computations onto event consumers, there should be a very loose coupling between event producers and event consumers. In this section, we describe a publish/subscribe system where event producers and consumers, denoting reactive application components, find each other in the mobile ad hoc network by means of *intensional descriptions* that are broadcasted using UDP to allow *decentralized and spontaneous discovery*. The difference with an extensional approach (e.g. a list of registered subscribers) is that one merely states the conditions that the properties of a producer or consumer must satisfy to establish a loosely-coupled binding between the two.

6.3.1 Ambient Behaviors

Ambient behaviors are our main abstraction to allow ambient-oriented dataflow programming. Ambient behaviors are like normal behaviors, i.e. reactive values that change over time. Just like normal behaviors, when they are used in AmbientTalk/R code, the dataflow dependencies of the expressions in which they are used are automatically tracked. The difference with normal behaviors is that they do not use local events to update themselves, but change in response to remote events on which they are intensionally subscribed using publish/subscribe.

We will continue the ticket trading example introduced in section 6.1.1. Recall that ticket vendors have a behavior that denotes their current location by means of GPS coordinates. What we actually want to achieve is to discover ambient behaviors made available by other devices that signal the events in which we are interested. In our example a behavior that represents the GPS coordinates of the location of the ticket vendor. Publishing such a behavior happens as follows, on the ticket vendor's device:

```
1  deftype TicketVendorLocation;
2
3  exportBehavior: GPS_Location as: TicketVendorLocation
4   to: { |buyer| buyer.interestedIn == "Rock Werchter" };
```

The `exportBehavior:as:to` construct is used on the publisher's side to publish a reactive value. Each time it changes, it will propagate an event to all reachable subscribers, and buffer it for the temporarily unreachable ones. The first argument is the reactive value, the second argument an AmbientTalk type tag (acting as a topic or event type). The third argument is an optional content-based event matching closure that is executed locally on the publisher's side, by applying it on potential subscribers' matching descriptions. Once publishers discarded subscribers of which the descriptions do not match with the closure, these subscribers can be safely disregarded and no communication resources are wasted on propagating events towards them (which would be the case if they would perform the event filtering on their side). The return value of the above construct is a "publication object" that implements a `cancel` method that when invoked causes the language runtime to cease advertising the publication in the network and to notify subscribers of change events.

Applications running on other devices can subscribe themselves on the events that are signaled by this behavior. This happens as follows:

```
1  deftype TicketVendorLocation;
2
3  def vendorLocation := ambientBehavior: TicketVendorLocation
4    where: { def interestedIn := "Rock Werchter" } @Any;
```

The `ambientBehavior:` construct is used to create an ambient behavior, which is nothing more than a local reactive value which is bound to one or more behaviors exported by other (possibly remote) event loops. In the example given above, the `@Any` annotation is used to indicate that `vendorLocation` should denote the location of a single ticket vendor, rather than a collection of vendor locations (the group-centric case is discussed in the next section). The first argument denotes the subscription type tag and the second argument is an optional description object that will be passed by-copy to potential event producers. Any definition in this description can be used by an event producer to match on, as is done above with the event that a ticket buyer is interested in.

Once an exported behavior can be found that matches the intensional descriptions given by the programmer (which can be either topic-based using only the type tag or content-based as above), the exported behavior will transparently start propagating update events to `vendorLocation`. Note that multiple applications can be subscribed to the `TicketVendorLocation` topic at the same time. The group communication required to notify all these subscribers is internally handled by the M2MI framework [KB02]. A sequence diagram showing an ambient behavior interacting with two exported behaviors is depicted in figure 6.3. It shows a `vendorLocation` ambient behavior subscribed to exported `GPS_Location` behaviors. It illustrates that when a disconnection with the exported behavior to which the `vendorLocation` currently is subscribed is detected, a replacement exported `GPS_Location` can be discovered and used to update the `vendorLocation` ambient behavior.

These update events trigger an update in the reactive value which may result in further reactive computation in its own event loop. For instance, the `vendorLocation` could be used to update the location of the ticket vendor on the map in the graphical user interface:

```
1  GUI.showLocationOnMap(vendorLocation);
```

The point here is that while the ticket vendor roams the environment and his GPS device signals updates to all subscribed behaviors, the maps on the user interfaces of the (reachable) interested parties are transparently updated with the new locations without resorting to callbacks. Furthermore, if an ambient behavior is disconnected from the exported behavior it was bound to, the ambient behavior will attempt to match with any other matching exported behavior in the ad hoc network. Finally, since ambient behaviors are treated as regular behaviors by the interpreter, they can be used in local reactive code as if they were behaviors that depend solely on local changes. On the other hand, applications that export the `GPS_Location` have no idea to which event consumers they propagate events, nor do they keep an explicit list of event consumers. This loose coupling between event producers and consumers is necessary to reflect the dynamic nature of mobile ad hoc networks and to support transparent reconfiguration when devices are roaming.

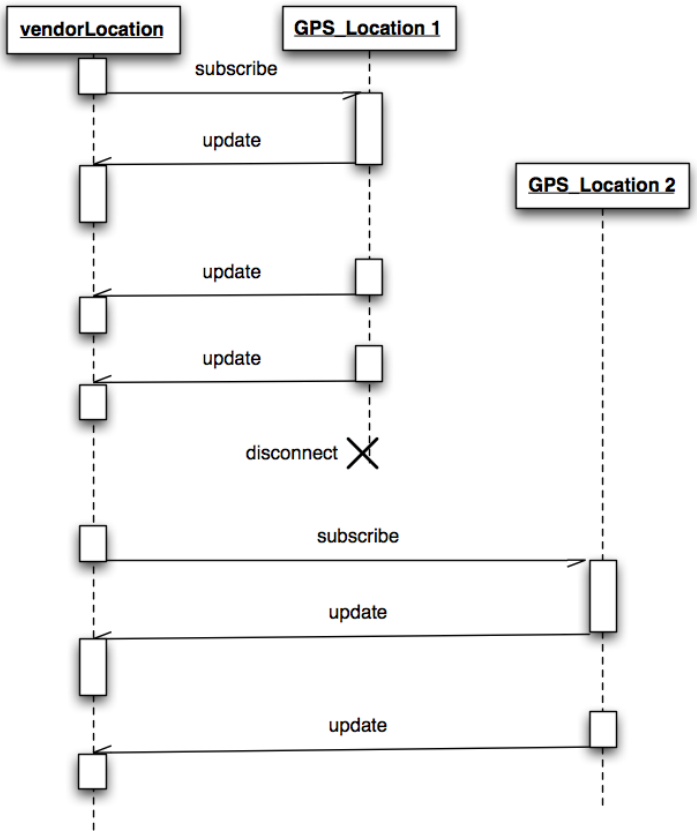

Figure 6.3: Example sequence diagram of an ambient behavior.

6.3.2 Group-centric Ambient Behaviors

In many cases, one is not only interested in receiving events from a *single* event producer, but to receive events from *all* reachable event producers. To cater for this case, an ambient behavior can be created that denotes a varying *set of* reactive values. Consider the following code:

```
1 def vendorLocations := ambientBehavior: TicketVendorLocation
2   where: { def interestedIn := "Rock Werchter" } @All(3.seconds);
```

Here, `vendorLocations` denotes a time-varying set of the locations of *all* nearby vendors of a ticket for Rock Werchter. Such group-centric ambient behaviors are created by annotating the construct with `@All`.

Notice that there is a time period passed to the `@All` annotation. The reason for this is that all update events are delivered asynchronously. This means that they can arrive at any order, possibly with large time-spans in between. Although we model the reactive `vendorLocations` as a continuous reactive value, in reality event processing still happens in a discrete fashion. This means that we have to *time-sample* the arriving events to determine to which update batch to the `vendorLocations` reactive set they belong. This happens as follows:

1. The `vendorLocations` ambient behavior listens for update events for three seconds and adds each value of the event coming from a different event producer as a new element to a new slot in the set. From event producers that already had published an event that was not invalidated yet by a new event, the event value is also received. If an event producer propagated a new update event before the three second timeout, the value in its dedicated slot of the reactive set is overwritten with the new value.

2. After three seconds, the `vendorLocations` reactive value itself signals a change. Its new value is the set containing the accumulated values from all event producers that propagated an event in the three second time span. The local part of the distributed dataflow graph is now signaled an update with the new value of the `vendorLocations` set.

3. When new event producers appear, their latest propagated event value is added to the next value of the `vendorLocations` set (to be signaled after three seconds).

4. Behind the scenes, the three seconds time span is not only used for time-sampling incoming events, but also as a timeout to heartbeat messages sent to event producers that they should acknowledge in the specified time span. If an event producer that previously added an event value to the `vendorLocations` fails to acknowledge a heartbeat message after three seconds, its propagated value is removed from the set, and the next value of the set will be one without that propagated value. If the event producer reappears in the network, it is rediscovered and can add again its latest value to `vendorLocations`.

This process is depicted in a sequence diagram in figure 6.4. It shows the `vendorLocations` group-centric ambient behavior (denoting a reactive set) subscribed to all reachable exported `GPS_Location` behaviors. Update event messages cause the reactive set to be updated with new locations. When a heartbeat message sent to a `GPS_Location` times out, its value is taken out of the set. If a new `GPS_Location` is discovered, its value is added to the set.

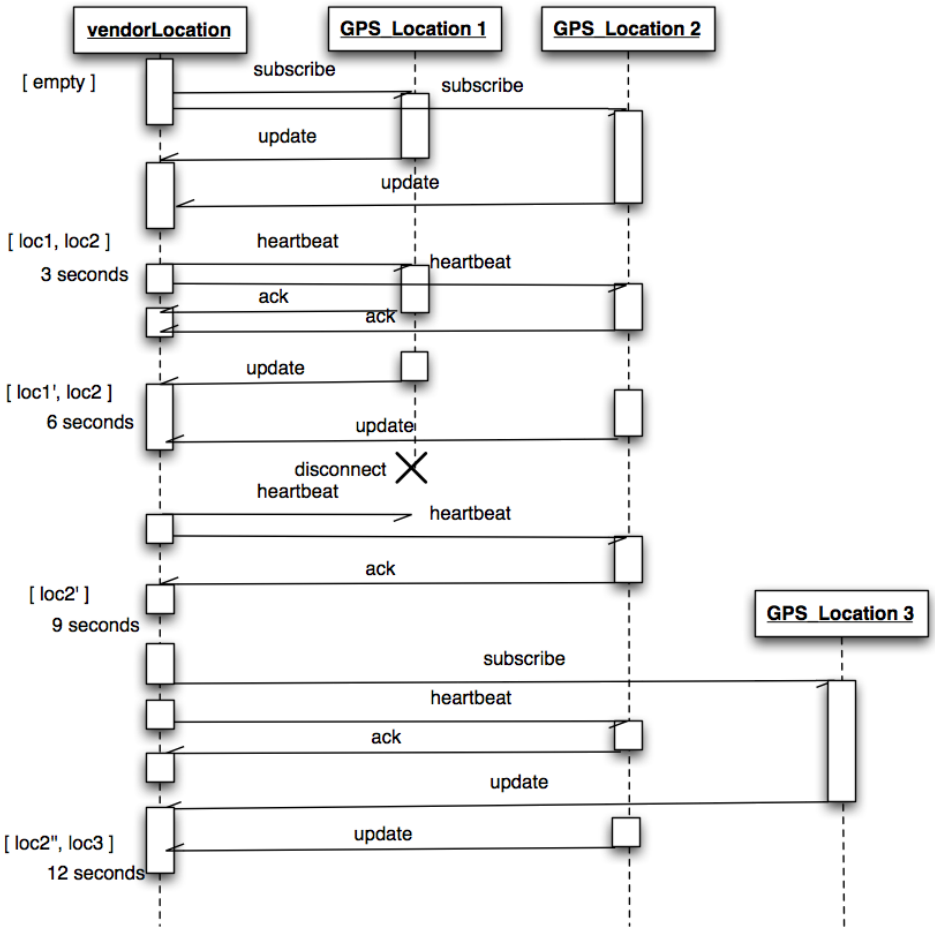

Figure 6.4: Example sequence diagram of a group-centric ambient behavior.

Incoming events are again handled as regular AmbientTalk messages by the event loop such that no race conditions can occur. Group-centric ambient behaviors abstract both over the events of changing values and the events of appearing and disappearing event producers. In response, the reactive set that is returned varies over time by updating its contents, growing or shrinking.

Table 6.6 below gives the operations that are supported by group-centric ambient behaviors.

`+(otherSet)`	Concatenation with another set (which can be reactive). Returns a new reactive set.
`contains(aValue)`	Checks if the set contains `aValue` (can be reactive). Returns a reactive boolean value.
`each:(aClosure)`	Applies `aClosure` to every element of the set, and will be reexecuted each time the reactive set on which it is invoked changes. Returns `nil`.
`filter:(aClosure)`	Returns a new set with only those elements that returned `true` when `aClosure` was applied to them. Returns a new reactive set.
`find:(aClosure)`	Returns the index of the first element that returned `true` when `aClosure` was applied to it (`nil` if no such element was found). Returns a reactive set.
`inject:into:` `(initialValue, aClosure)`	Collects all elements of the set by combining them using the given closure. Returns a reactive set.
`isEmpty()`	Checks if the set is empty. Returns a reactive boolean value.
`length()`	Returns the number of elements in the set. Returns a reactive value.
`map:(aClosure)`	Returns a new set by applying `aClosure` to each element. Returns a new reactive set.

Table 6.6: Supported operations by goup-centric ambient behaviors.

These operators make it possible to for example update the locations of interesting ticket vendors on the user's screen as follows:

```
1 vendorLocations.each: { |newLocation|
2   GUI.showLocationOnMap(newLocation);
3 };
```

Note that almost all operations return reactive values in their turn that can be used in the rest of the dataflow code. This degree of functional composition is not present when programming with explicit callbacks.

6.3.3 Reactive Queries

Reactive queries can be regarded as the dual language construct of ambient behaviors, offering pull-based instead of push-based communication. This duality is shown in figure 6.5. Ambient behaviors as described above can only be used if there is a node that publishes its service as an ambient behavior. Otherwise, the subscriber has to obtain ambient behaviors by querying the network for relevant information itself. For

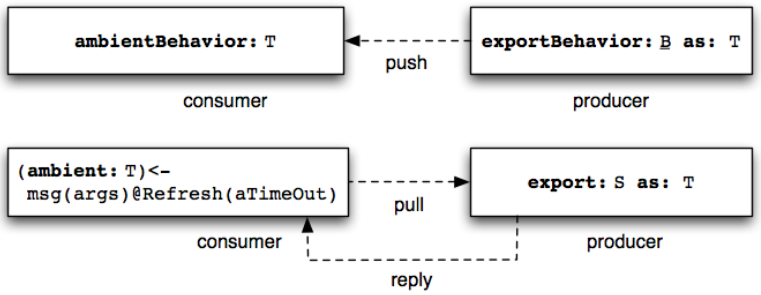

Figure 6.5: Duality between ambient behaviors and reactive queries.

this the programmer is provided with an abstraction that allows creating a behavior that autonomously queries the network to update itself. This abstraction is an integration of ambient references (which allow querying the network by sending messages) with the reactive programming language facilities of AmbientTalk/R (which allow reacting to and processing events without inversion of control). The example below shows the creation of a behavior by querying the network using a reactive ambient reference. The ambient reference is created on line 4 and the reactive query is launched on line 6, returning a reactive value.

```
1  deftype TicketVendor;
2
3  def werchterVendors :=
4    ambient: TicketVendor where: { |tv| tv.event == "Rock Werchter" };
5
6  def locations := werchterVendors<-getLocation()@Refresh(5.seconds);
```

The `getLocation()` message is annotated with `@Refresh`, which implies that the results of the message are accumulated in a reactive value. Hence, the `locations` variable contains a reactive value which initially denotes an empty set. The `@Refresh` annotation implies that the annotated `getLocation` message is sent every 5 seconds to all nearby ticket vendors offering a ticket for Rock Werchter[2]. The resulting `locations` behavior is updated every five seconds and contains a set of all responses from the ticket vendors in range.

An example scenario is depicted in the sequence diagram in figure 6.6. It shows the `locations` behavior that queries every 5 seconds all `TicketVendor`s that match the description of the ambient reference for their location through an ambient reference. Disconnected `TicketVendor`s are discarded by the ambient reference and newly discovered `TicketVendor`s are queried for their location as well. Replies to queries and observed disconnections cause the `locations` reactive set to change.

Since `locations` is a reactive set, it can be passed on to other functions or methods as any other value, as done below to update the map in the user interface of the user with *all* locations:

```
1  locations.each: { |coordinates| GUI.showLocationOnMap(coordinates) };
```

[2]In addition to the `@Refresh` annotation, one can add annotations to the message that specify the message sending semantics. By varying these annotations, one can decide to send the message to all objects in range like in the example (`@All`), which will result in a changing array of results, or send the message to just one of the objects in range, resulting in a behavior containing a single value (`@Any`).

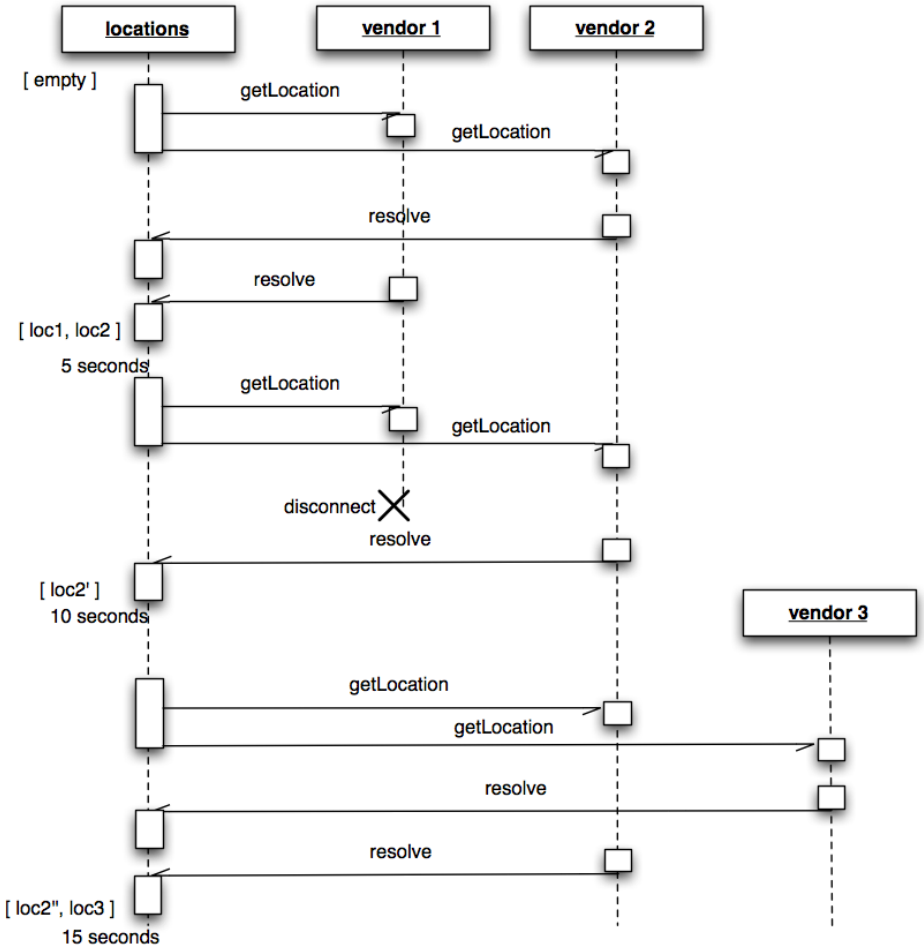

Figure 6.6: Example sequence diagram of a reactive query.

Note that by making use of a reactive query, the programmer no longer has to explicitly poll the environment in a loop.

To conclude, integrating ambient references with reactive programming allows the results of queries over the network to be collected into a behavior that is automatically synchronized with the environment. Ambient references provide an abstraction over the events of appearance and disappearance of services in the network, while the reactive programming system provides an abstraction over the events generated by the reception of results of asynchronous queries.

6.3.4 Summary

In this section, we summarize the language constructs introduced earlier in this chapter that allow to implement ambient-oriented applications as node-centric dataflow programs.

`@Mutator` Methods of reactive objects annotated with this annotation trigger an event when invoked.

`makeReactive(anObject)` Returns a reactive version of `anObject`[3].

`exportBehavior:as:(anObject, aTypetag)` Publishes a reactive object `anObject` under `aTypetag` (topic-based publication). Returns a publication object p such that `p.cancel()` cancels the publication.

`exportBehavior:as:to:(anObject, aTypetag, aClosure)` Publishes a reactive object `anObject` under `aTypetag` and uses the `aClosure` predicate to filter out subscribers (content-based publication). The predicate closure is applied locally on description objects sent by potential subscribers. Returns a publication object p such that `p.cancel()` cancels the publication.

`ambientBehavior:(aTypetag)` Subscribes to all reactive objects published under `aTypetag` (topic-based subscription). Returns two objects: the behavior itself and a subscription object s such that `s.cancel()` cancels the subscription.

`ambientBehavior:where:(aTypetag, aDescription)` Subscribes to all reactive objects published under `aTypetag` using the `aDescription` object to describe the content-based subscription. Returns two objects: the behavior itself and a subscription object s such that `s.cancel()` cancels the subscription.

`@Any` An ambient behavior subscription annotated with this annotation returns a single reactive object that reflects the latest value of a published object (using best effort semantics with respect to network failure handling). If a disconnection is observed, the subscription mechanism is restarted to find a replacement published reactive object.

`@All(aTimePeriod)` An ambient behavior subscription annotated with this annotation returns a reactive set of reactive objects that reflect the latest value of all received published objects. Results from unresponsive publishers are removed from the collection and results from newly appearing publishers are added to the collection. `aTimePeriod` is used for both time-sampling incoming results as well as for a timeout value to discard unresponsive publishers.

[3] More precisely, it establishes a dataflow dependency between a reactive *variable* and the original object to which it refers.

`@Refresh(aTimePeriod)` A message broadcasted via an ambient reference annotated with this annotation returns a reactive set of results to the broadcasted message. The message is broadcast every `aTimePeriod` and the reactive set signals every `aTimePeriod` a new event with its latest results (results from peers that already replied are replaced with fresh results).

As can be seen from the last construct, reactive queries are implemented as an extension of ambient references [Van08]. Although both rely on AmbientTalk/R's reactive interpreter, reactive queries and ambient behaviors do not depend on each other.

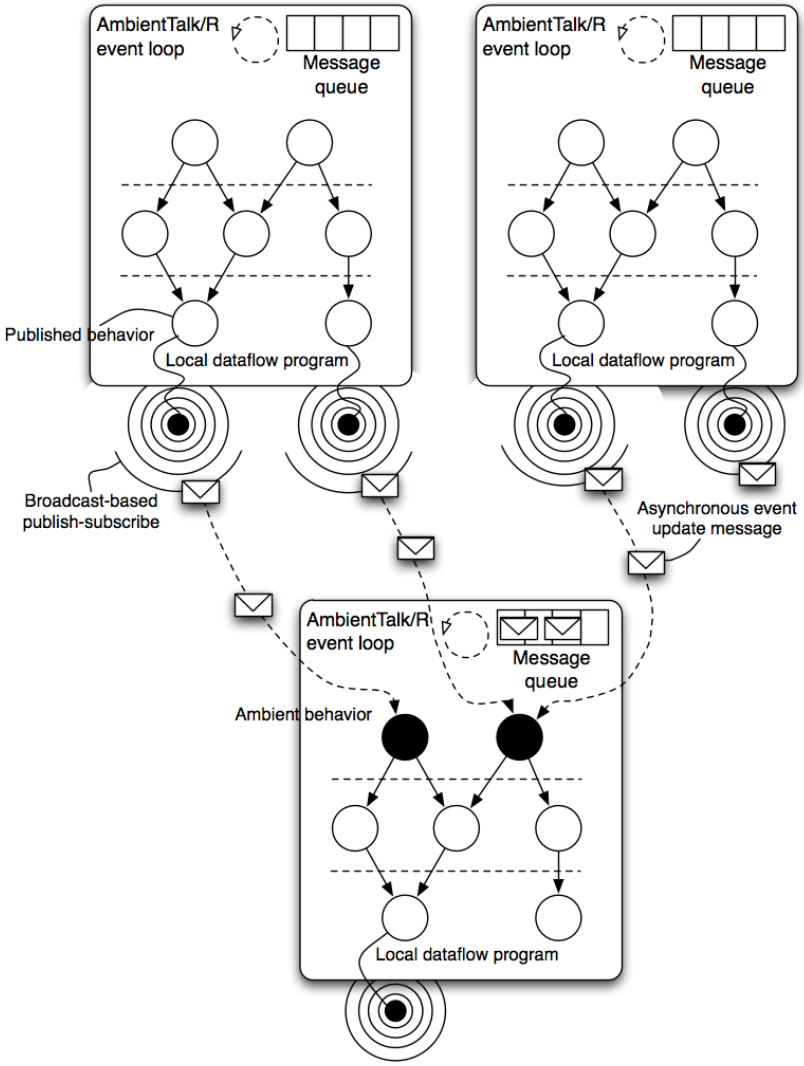

Figure 6.7: Schematic overview of an AmbientTalk/R dataflow program distributed using ambient behaviors.

These language constructs allow to distribute dataflow programs over different event loops using space, time and arity-decoupled publish/subscribe primitives. Fig-

ure 6.7 shows a schematic overview of an AmbientTalk/R dataflow program distributed using ambient behaviors. It shows three AmbientTalk/R event loops hosting local dataflow programs. These local dataflow programs constitute a larger distributed dataflow program. The distributed dataflow dependencies behave as publishers and subscribers in a publish/subscribe system. They take care of transforming dataflow events into asynchronous AmbientTalk messages that can be sent across event loops and scheduled the in message queues of subscribed event loops. When processed, these event messages update dependent ambient behaviors.

The loosely coupled interaction offered by the underlying publish/subscribe primitives allows that event loops are dynamically matched at runtime to satisfy dataflow dependencies. This means that the distributed dataflow graph can be reconfigured at runtime because of changes in the network topology of the mobile had hoc network. This also means that the set of exported behaviors that are aggregated into a single group-centric ambient behavior is in constant flux as the network topology changes.

In the next section, we use the aforementioned ambient-oriented dataflow abstractions to complete the implementation of the Ticket Trader example application.

6.3.5 Implementing The Ticket Trader Application with Ambient Behaviors

In previous sections, we have used the dynamic discovery of ticket vendors and their location as a running example to explain the various features of AmbientTalk/R. This section presents a slightly more elaborate version of the application, which matches ticket vendors with prospective clients. In publish/subscribe terminology, ticket vendors *publish* the offers for tickets they are willing to sell, while their prospective clients *subscribe* to events concerning tickets being offered in their vicinity. Clients are able to identify which ticket offers are relevant to them by specifying which events they want to attend, the price they are willing to pay for the ticket and the maximal allowed distance between themselves and the ticket vendor. The latter filter requires that both vendors and clients have access to a GPS device, such that their GPS coordinates can be used to compute the distance.

Note that different ticket vendors can offer tickets for the same event (possibly for a different price) and that different clients can be interested in the same ticket. Furthermore, both vendors and clients roam the environment, can cancel their offers, change the price of their offers, and announce new offers. The different instances of the application on the different devices should all be able to deal with these changes.

Before turning our attention to the implementation of both parties in the application, we show the implementation of a simple isolate object representing a ticket offer:

```
1  def TicketOffer := isolate: {
2    def eventName := nil;
3    def price     := 0;
4    def location  := nil;
5
6    // Constructor
7    def init(anEventName, aPrice) {
8      eventName := anEventName;
9      price     := aPrice;
10   };
11 };
```

The object contains three slots: the event the ticket provides access to, the price at which it is currently being offered and the vendor's current location. During the lifetime

of the application, the latter two values may change: the vendor may roam and decide to adjust the price at which the ticket is being offered. Typically, the price can be reduced if interest is low or if the event is about to start.

To determine the vendor's current position, we reuse the `GPS_location` abstraction, which was defined previously as follows:

```
1 def GPS_Location := makeReactive(Coordinate.new());
2 GPS.addLocationObserver: { |lat, lng| GPS_Location.update(lat, lng) };
```

The following code excerpt defines an AmbientTalk type tag that is used as the topic under which ticket offers are published.

```
1 deftype TicketOfferT;
```

Having described the necessary abstractions, we can now describe how ticket offers are published by the vendor:

```
1  def TicketVendor := object: {
2    def offeredTickets := HashMap.new();
3
4    // Offer a new ticket.
5    def offerTicket(eventName, price) {
6      def ticketOffer := makeReactive(TicketOffer.new(eventName, price));
7      ticketOffer.location := GPS_Location;
8      offeredTickets.put(eventName, ticketOffer);
9      exportBehavior: ticketOffer as: TicketOfferT;
10   };
11
12   // Change the price of a ticket on offer.
13   def setTicketPrice(eventName, newPrice) {
14     (offeredTickets.get(eventName)).price := newPrice;
15   };
16 };
```

Tickets are offered to all nearby prospective clients by invoking the `offerTicket` method. In it, a reactive `TicketOffer` object is created (line 6). Because the object is reactive, the vendor is guaranteed that whenever the offer changes, these changes are automatically propagated to all prospective clients. One way in which the offer might change is if the location of the vendor changes. Note that in line 7, the location associated with the offer is set to the vendor's `GPS_Location`. Due to the fact that `GPS_Location` is a reactive value and because of the semantics of mutating reactive values (see section 6.2), the `location` field of the `ticketOffer` will be set anew whenever the `GPS_Location` is updated. In turn, this update will be propagated to reactive values which depend on `ticketOffer`. In this particular case, this includes all prospective clients that are currently in range. The fact that these prospective clients can detect the offer, stems from the fact that it is published using the `exportBehavior:as:` construct in line 9.

Additionally, vendors keep track of the various events for which they offer tickets in the `offeredTickets` map, which associated the names of the events with the tickets offered for these events. This map is used to update the price at which tickets are being offered. The `setTicketPrice` method uses the mapping to find a particular reactive ticket offer in order to update its price. Because of the semantics of mutating reactive values, setting the price causes an update to be propagated to all prospective clients in reach.

The following code excerpt shows the `findOffers` function, which permits prospective clients to detect ticket offers that have been exported by nearby vendors by means

of the `ambientBehavior:` construct.

```
1  def findOffers(event, maximumPrice, maximumDistance) {
2    // Subscribe to TicketOffers
3    def allNearbyOffers := ambientBehavior: TicketOfferT @All(_timespan_);
4
5    // Filter out interesting TicketOffers
6    allNearbyOffers.filter: { |offer|
7      (offer.eventName == event).and: {
8        (offer.price <= maximumPrice).and: {
9          GPS.Location.distanceTo(offer.location) <= maximumDistance }}};
10 };
11
12 def werchterTicketVendors := findOffers("Rock Werchter", 200, 500);
13 gui.updateWithOffers(werchterTicketVendors);
```

The ambient behavior `allNearbyOffers` will be bound to a collection that contains all ticket offers made by nearby vendors. This is because the `@All` annotation is used, rather than the `@Any` annotation used before. In other words, `allNearbyOffers` is a reactive value denoting a time-varying set of exported ticket offers. The size of this set evolves as ticket vendors move in and out of range.

The reactive set `allNearbyOffers` is subsequently filtered to produce a selection of ticket offers that are relevant to the client (lines 6-9). An offer is deemed relevant if it provides access to the correct event, its price does not exceed the maximum set by the client and if the distance to the client's current location does not exceed a given maximum. Since `allNearbyOffers` is an ambient behavior, the invocation of its `filter:` method (an accessor method) creates a dependent reactive value, which is the return value of the function. This reactive value is updated when vendors go in and out of range, but also if one of the previously detected offers change (i.e. the vendor has moved or the price has been updated). Furthermore, it is important to note that the condition to determine whether an offer is relevant also depends on the location of the prospective client. In line 9, the vendor's location is compared to the current location of the client. This implies that an offer can suddenly become relevant as the client is roaming.

In the code snippet, the `findOffers` function is called to find ticket offers to attend Rock Werchter, which cost less than 200 euro and whose vendor is less than 500 meters away. The resulting collection is passed as an argument to the `updateWithOffers` method of the user interface. This method expects a set of ticket offers (which all contain their last location) and draws them on the map. Since the `werchterTicketVendors` collection is a reactive set, this method will be invoked anew whenever the collection is updated.

Discussion

Notice that ticket vendors and their prospective clients are loosely coupled with regard to one another. They discover one another by means of a topic-based publish/subscribe architecture (which uses an ordinary AmbientTalk type tag to denote the type of events that are exchanged). Parties that are disconnected from each other (e.g. by network partitioning of the mobile ad hoc network) are automatically discarded while newly connected parties dynamically discover each other and start exchanging events.

Furthermore, the publication of new events is integrated closely with the imperative object-oriented programming style of the host language. Provided that mutator methods are properly identified, any object can be used to create a reactive value. Once

such a reactive value has been published, it suffices to write one of the object's fields or invoke one of its mutator methods to implicitly emit events notifying all reachable subscribers of the change. Recall from the previous chapter that our programming model for mobile RFID-enabled applications relies on side-effects to trigger writes on the memory on RFID-tagged things. An entirely functional reactive programming style would not integrate well with this model.

Finally, the subscriber can trivially indicate which events it is interested in receiving (by means of a topic-based subscription) and can handle incoming events without resorting to a complex network of event handlers. In the ticket trader example, two sources of events are considered, to wit `allNearbyOffers` and `GPS_Location`. Changes to the former are the result of the appearance and disappearance of vendors (which result in the addition and removal of certain offers) as well as updates to the offers themselves (i.e. price and/or location updates). Changes to the latter are the result of roaming clients, and affect the number of offers reported to the user as they affect the distance between the vendor and the prospective client. The interplay between these different event sources are handled implicitly by the AmbientTalk/R interpreter.

6.4 Case Study: The RFID-Enabled Library

In this section, we continue the example mobile RFID-enabled Book Recommender application introduced in section 6.1.2. Important to note here is that we are going to integrate the programming model for mobile RFID-enabled applications introduced in chapter 5 with ambient-oriented dataflow programming. This way, the explicit callbacks necessary to react to different kinds of RFID events are replaced by reactive values denoting changing sets of reachable RFID proxy objects. The same holds for parts of AmbientTalk's event-driven communication constructs. The consequence is that part of the control flow of the application is not explicitly managed by the programmer by registering callbacks, but implicitly by the interpreter of the language that tracks dependent reactive expressions.

6.4.1 Connecting a Reactive Value to a Sensor

In many cases, reactive values are driven by sensor devices and must be causally connected to them in some way. *Event sources* are AmbientTalk/R time-varying values that – in contrast to behaviors – do not trigger reevaluation of dataflow-dependent expressions. They require the programmer to manually query them for their latest value. On the other hand, they allow the programmer to create a time-varying value without resorting to the `makeReactive` construct in combination with `Mutator` methods. This is useful when porting legacy objects to reactive applications. Typical examples are collection objects such as lists, sets, etc., GUI elements.... Similarly, in almost all cases, sensor device APIs, will not be conceived taking into account reactive programming abstractions. This is the case as well for our RFID abstractions introduced in the previous chapter. Hence, we rely on an event source to causally connect a collection object to an RFID reader.

Consider the code below that can be used to construct a reactive set of all scanned RFID-tagged objects of a certain type.

```
1  // Initialize the RFID event loop
2  /.at.briges.RFID.RFIDServiceRunner();
3
4  // Scan for tags of a certain type
5  def scan(aTypetag) {
6    def theScannedThings := [ ];
7    def theEventSource := makeEventSource();
8
9    def theBehavior := hold: theEventSource with: (object: {
10     def updateWithValue(aValue) { aValue };
11   });
12
13   theEventSource<-updateWithValue(theScannedThings);
14
15   whenever: aTypetag discovered: { |obj|
16     theScannedThings := theScannedThings + [ obj ];
17     theEventSource<-updateWithValue(theScannedThings);
18     whenever: obj disconnected: {
19       theScannedThings.remove(obj);
20       theEventSource<-updateWithValue(theScannedThings);
21     };
22     whenever: obj reconnected: {
23       theScannedThings := theScannedThings + [ obj ];
24       theEventSource<-updateWithValue(theScannedThings);
25     };
26   };
27
28   theBehavior;
29 };
```

The `scan` procedure takes a type tag as argument and returns a reactive set of things tagged with that same type tag. The reactive set grows and shrinks as objects leave the reading range of the RFID reader used by the RFID event loop [4].

First, it encapsulates a private set `theScannedThings` storing the scanned things in range. Subsequently, it also holds a private event source `theEventSource`. This event source is used to initialize a reactive value using the `hold:with:` primitive. This primitive takes an event source as first argument and an updater object as second argument. When this updater object's methods are invoked, the event source signals a new event and the reactive value that was derived from it gets a new value. The value is the return value of the method. In this case, we implement simply the identity function. The updater object's methods can be invoked by sending the event source that is associated with it an asynchronous message, passing it the value used for updating.

The initial value of the event source (and hence the reactive value as well) is set to an empty set. Subsequently, a number of event handlers are registered that are triggered when things tagged with the passed type tag are discovered, disconnected or reconnected again. In these event handlers, the event source is updated by sending it the message corresponding to the associated updater object's update method. In this message, we simply pass the new set of RFID-tagged objects in range (with matching type tag).

The result of calling the `scan` procedure is a reactive set that contains the last set of scanned RFID-tagged objects tagged with the type tag passed to the procedure.

[4] For the sake of brevity, we represent RFID-tagged objects here as isolates that are stored on the tags' memory and that are passed by copy. In a setting where the data on the tags may frequently change, it is better to not just rely on the discovery primitives, but use the full infrastructure introduced in the previous chapter where things are encapsulated in an RFID event loop and addressed by far reference.

6.4.2 Node-Centric Dataflow Primitives at Work

Once we have a reactive value denoting the currently scanned RFID-tagged books, we can build the complete reactive distributed dataflow application on top. The distributed application application consists of five objects:

- The `book` thing from the previous chapter (see section 5.3.1),
- The `BookRecommender` object that implements the part of the application running on the user's smartphone,
- The `BookScanner` object representing an RFID-enabled book scanning station in the library,
- The `EBookReader` object implementing the book recommendation part of the application running on the user's e-book reader (which could possibly be hosted on the user's smartphone as well),
- The `Shelf` object representing an RFID-enabled shelf in the library.

The source code of the Book Recommender application is given below. Lines 2 to 11 show some private definitions and two methods to update the two lists that are shown in the user interface: a list with current books the user is carrying around, and based on that list another list with recommended books that can be found in the library, together with the shelf on which they are located.

```
1  def BookRecommender := object: {
2    def user := //...
3    def gui :=  jlobby.at.demo.ShopAssistantGUI.new();
4
5    def showBooks(books) {
6      gui.updateBookList(books);
7    };
8
9    def showRecommendationsInStock(books, location) {
10     gui.updateRecommendedInStock(books, location);
11   };
12
13   deftype ScannedBooks;
14   deftype RecommendedInStock;
15
16   def scannedBooks := ambientBehavior: ScannedBooks
17     where: { def customer := user } @Any;
18
19   def recommendedBooks := ambientBehavior: RecommendedInStock
20     where: { def customer := user } @Any;
21
22   deftype Customer;
23   export: user as: Customer;
24
25   showBooks(scannedBooks);
26   showRecommendationsInStock(recommendedBooks[1], recommendedBooks[2]);
27 };
```

To update these lists, two ambient behaviors are subscribed. First, the `scannedBooks` ambient behavior listens for events type-tagged as `ScannedBooks` (lines 16-17). Additionally, it attaches a description object to the subscription that stores a reference to the user of the application, such that publishers of these events can direct events only to

the interested user. The ambient behavior is annotated with `@Any` because we assume that the user is in range of a single RFID reader.

Similarly, there is also an ambient behavior subscribed to events denoting recommended books that are in stock together with their location (lines 19-20). In this scenario, these events are published by the shelves in the library that have an integrated RFID reader. This ambient behavior is also annotated with `@Any`. It would also be possible to receive *all* recommendations from *all* shelves in communication range in the library by annotating it with `@All`.

To allow remote parties interacting with the application to detect that there is a customer nearby and obtain a reference to this customer, the application exports the object representing the user as a `Customer` (lines 22-23).

To update the user interface, these two ambient behaviors are simply passed to the two user interface methods updating the two lists (lines 25-26). The dataflow evaluation model of the AmbientTalk/R interpreter automatically ensures that these methods are reinvoked when these two behaviors change (e.g. because the user changed the books that he is carrying or because he moved closer to another shelf).

As already mentioned, the RFID reader scanning books will notify book recommender applications of the list of scanned books that the user is carrying around. Such devices equipped with an RFID reader host the `BookScanner` object shown below.

```
1  def BookScanner := object: {
2    deftype Customer;
3    deftype ScannedBooks;
4    deftype Book;
5    def booksBehavior := /.bridges.rfid.ReactiveRFIDService.scan(Book);
6
7    whenever: Customer discovered: { |customer|
8      def publication := exportBehavior: booksBehavior as: ScannedBooks
9        to: { |desc| desc.customer == customer };
10     whenever: customer reconnected: {
11       publication := exportBehavior: booksBehavior as: ScannedBooks
12         to: { |desc| desc.customer == customer };
13     };
14     whenever: customer disconnected: {
15       publication.cancel();
16     };
17   };
18 };
```

It creates the necessary type tags and (line 5) creates a reactive set denoting all currently scanned books by the RFID reader, using our abstraction introduced earlier in section 6.4.1. Subsequently, it registers listeners to detect customers. This is needed to use the detected customer in the listener in the subscription matching process to send the recommendations to the correct customers. When a customer is detected or reconnected, the `booksBehavior` is published to subscribers of which the customer matches. When a customer disconnects, the publication of the `booksBehavior` for that customer can be cancelled (line 14-16).

A second part of the distributed dataflow application that the user is carrying around, is the part that runs on his e-book reader.

```
1  def EBookReader := object: {
2    def user := //...
3
4    deftype ScannedBooks;
5    deftype RecommendedBooks;
6
```

```
7     def books := ambientBehavior: ScannedBooks
8       where: { def customer := user } @Any;
9
10    def recommendedBooks :=
11      flatten(books.map: { |book| generateSimilarBooks(book) });
12
13    exportBehavior: [ user, recommendedBooks ] as: RecommendedBooks;
14  };
```

It subscribes an ambient behavior `books` that listens for the books scanned by the RFID reader. The subscription uses a predicate to filter out books that were not intended for the user. The resulting reactive set is subsequently used to map over a closure that for each book generates a collection of similar books based on the electronic book store information on the device (such as Amazon or iTunes). After flattening this set, this results in another reactive set with recommended books. This reactive value is published as a two-tuple consisting of the user together with his recommended books. The software running on the shelves in the store receiving this tuple can then use the reference to the customer in it to publish the filtered recommended books to the correct user.

The object hosted by the shelves in the store is shown below. Every shelf is associated with a location identifier corresponding to their location in the library (an idea for a more advanced version of the application could be to highlight the shelf on a floor plan of the library shown in the user interface of the application), such that user can easily find the books that are being recommended.

```
1   def Shelf := object: {
2     def location := //...
3
4     deftype Book;
5     deftype ScannedBooks;
6     deftype RecommendedBooks;
7     deftype RecommendedInStock;
8
9     def booksInStockBehavior :=
10      /.bridges.rfid.ReactiveRFIDService.scan(Book);
11
12    def userPublicationMap := HashMap.new();
13
14    def recommendedForUser :=
15      ambientBehavior: RecommendedBooks @All(3.seconds);
16
17     recommendedForUser.each: { |userAndRecommendedBooks|
18      def [ customer, recommendedBooks ] := userAndRecommendedBooks;
19      def recommended := recommendedBooks.filter:
20        { |book| booksInStockBehavior.contains(book) };
21
22      def oldPublication := userPublicationMap.get(customer);
23      if: (nil != oldPublication) then: {
24        oldPublication.cancel();
25      };
26
27      def publication :=
28        exportBehavior: [ recommended, location ]
29          as:  RecommendedInStock
30          to: { |desc| desc.customer == customer };
31
32      userPublicationMap.put(customer, publication);
33    };
34  };
```

Shelves are equipped with an RFID reader, which allows them to be aware of which books they contain. Again, we use our reactive abstraction used earlier to generate a reactive value of books in reading range. These denote the current books in the shelf.

On line 12, a map is created to associate the publication objects of book recommendations published for a certain customer with the corresponding customer. We explain later why.

The `recommendedForUser` ambient behavior registered next listens for *all* user - books pairs broadcasted by `EBookReader`s using the `@All` annotation. Hence, it denotes a reactive set of such pairs. It is used to iterate over (lines 17-33) all of these pairs. In this iteration, a new reactive value `recommended` is created that depends on two other reactive values: the list of recommended books of the current tuple in the iteration and the `booksInStockBehavior` that denotes the current books in the shelf. The resulting reactive set contains the filtered recommended books that are present in the shelf.

Subsequently, after the old publication for that customer out of the tuple is cancelled, the new reactive list of recommended books that are in the shelf is published together in a two-tuple with their location under the type tag `RecommendedInStock` and by matching on the customer for which the recommendations are intended. The resulting publication object is put in the `userPublicationMap`.

6.4.3 Evaluation

Let us evaluate the implementation of the Book Recommender application given above with respect to the programming model requirements for mobile RFID-enabled applications put forward in chapter 2 section 2.3.4. Here, we focus on the distribution and event handling part of these requirements. The requirements for interacting with RFID-tagged objects were covered in the previous chapter.

Space Decoupling

The various distributed application components in this application spontaneously interact when they are in proximity by relying on the publish/subscribe interaction style of ambient behaviors. For the `BookRecommender` mobile application, it suffices to subscribe to the appropriate exported behaviors, as shown in the code snippet taken from the `BookRecommender` object below:

```
1  // ...
2  deftype ScannedBooks;
3  deftype RecommendedInStock;
4
5  def scannedBooks := ambientBehavior: ScannedBooks
6    where: { def customer := user } @Any;
7
8  def recommendedBooks := ambientBehavior: RecommendedInStock
9    where: { def customer := user } @Any;
10 // ...
```

The only knowledge the `BookRecommender` needs for this are the type tags under which ambient behaviors are exported. Because both subscriptions are annotated with `@Any`, in case of a disconnection with an event producer, subscribing to a replacement (or the same in case it reconnected) event producer is attempted automatically.

Arity Decoupling

In the code snippet shown above, `BookRecommender` objects are subscribed to singular ambient behaviors (because the subscriptions are annotated with `@Any`). This means the Book Recommender application will receive recommended books present in the library from a single nearby shelf. Adapting this behavior to receiving recommended books from *all* nearby shelves happens by simply changing the `@Any` annotation to `@All`, as shown in the code snippet below:

```
1 // ...
2 def recommendedBooks := ambientBehavior: RecommendedInStock
3   where: { def customer := user } @All;
4 // ...
```

It only depends on the subscription whether a group-centric or a singular ambient behavior is created: event producers do not have to take this into account.

Time Decoupling and Fault Tolerant Communication

Ambient behaviors rely on standard asynchronous AmbientTalk messages to communicate events. These messages buffered by the sender in case of a disconnected and resent in case of a reconnection by default. To deal with permanent failures, the programmer can specify a timeout an ambient behavior subscriptions that determines how long events intended for a specific subscriber are buffered. If the timeout expires, these event messages are discarded.

Rich Representation of Events

To be able to rely on the thing abstraction introduced in the previous chapter for representing RFID-tagged books, events must be represented as full-blown book things. Ambient behaviors transform events to asynchronous AmbientTalk messages, which can be passed any kind of AmbientTalk object (including things), keeping into account AmbientTalk's parameter passing semantics.

No Inversion of Control

It is clear from the behavior denoting the currently detected books that polling for books should not happen manually anymore and calling and registering callbacks when books are detected is eliminated. This allows to declaratively reason over the detected set of books, as done in the implementation of the `Shelf` object. Consider the following code snippet, taken from the implementation of `Shelf`:

```
1 recommendedForUser.each: { |userAndRecommendedBooks|
2   def [ customer, recommendedBooks ] := userAndRecommendedBooks;
3   def recommended := recommendedBooks.filter:
4     { |book| booksInStockBehavior.contains(book) };
5
6   // ...
7 };
```

It depends on two reactive values: `booksInStockBehavior` denoting the current set of books in the shelf and `recommendedForUser` denoting a reactive set of tuples associating customers with their personalized book recommendations. This latter reactive value is an ambient behavior to which the `Shelf` objects are subscribed. The expression shown above registers a dataflow dependency on this reactive value by calling the

`each:` method on it (which will apply the closure passed as an argument to each element of the set). Within that closure, the list of recommended books for a particular user is retrieved (second element of the pair), and a second dataflow dependency is registered by using the reactive value `booksInStockBehavior` in the closure passed to the `filter:` method called on this particular set of recommended books. The result is that when the set of books in the shelf changes, the dependent `recommended` reactive value signals an event and causes the rest of the code in which it is used to be reevaluated. Similarly, when a remote `EBookReader` object signals a new event for the ambient behavior published under the topic `RecommendedBooks`, the `recommendedForUser` set of customer-books tuples will change and cause the surrounding call to the `each:` method to be reinvoked. The AmbientTalk/R interpreter with its automatic stratification of dataflow dependencies makes sure the reevaluation happens in the expected order and hence no glitches occur.

Global and Automatic Control Flow Management

In this implementation, the control flow is automatically managed on the node level. Thanks to the composability of (ambient) behaviors, the control flow of the application is now clearer than in an implementation that is entirely based on callbacks. Still, because of the automatic reevaluation of dataflow-dependent expressions, it is not entirely reflected in the textual representation of the program code.

On the network level, the programmer must keep a clear mental overview of what application component is subscribing to what and how it affects its execution. This issue is clearly more prominent in this application – where multiple heterogenous parties interact – than in the ticket trader application introduced earlier where all parties are homogenous and there is less variety of events to be handled.

Other Shortcomings

The astute reader will have noticed that the `BookScanner` object still registers a number of event handlers to associate one by one the lists of books generated by the RFID event loop with a single customer, as shown below:

```
1  whenever: Customer discovered: { |customer|
2    def publication := exportBehavior: booksBehavior as: ScannedBooks
3      to: { |desc| desc.customer == customer };
4    whenever: customer reconnected: {
5      publication := exportBehavior: booksBehavior as: ScannedBooks
6        to: { |desc| desc.customer == customer };
7    };
8    whenever: customer disconnected: {
9      publication.cancel();
10   };
11 };
```

This is needed to allow the other remote application components that are subscribed to `ScannedBooks` to generate personalized recommendations. This check crosscuts the entire distributed application and is clearly awkward. It stems from the fact that the broadcast-based publish/subscribe event delivery is undirected by default and requires subscribing application components to manually filter out events that were not intended for them.

To deal with these open issues, in the next chapter we present a *network-centric* ambient-oriented dataflow language that allows to *visually* wire distributed reactive

components together using dataflow dependencies, giving the programmer a *network-centric* view on the entire dataflow program. Different instances of the visual dataflow program can run in the same network without interfering with each other, unless explicitly specified by the programmer.

6.5 Implementation

In this section, we detail the implementation of the constructs that allow the publishing of and subscribing to ambient behaviors. They are implemented entirely in AmbientTalk/R by making use of the ambient references [Van08] extensions to the language.

6.5.1 Publishing Ambient Behaviors

Publishing reactive values as ambient behaviors happens using the `exportBehavior:as:` or `exportBehavior:as:to:` constructs. For the sake of brevity, we only show the implementation of `exportBehavior:as:to:`. The implementation of `exportBehavior:as:` is identical except that it passes `nil` as the subscription closure that is normally passed. The implementation is shown below.

```
1  def exportBehavior: aBehavior as: type to: contentCheckClosure @Lifted {
2    def behaviorMirror := reflect: aBehavior;
3    findAndUpdateListeningBehaviors(
4      aBehavior,
5      type,
6      contentCheckClosure,
7      behaviorMirror);
8  };
```

The procedure is annotated with `@Lifted`. This instructs the AmbientTalk/R interpreter to interpret the `aBehavior` formal parameter as a non-changing value and hence not create a dataflow dependency between this argument and the invocation of the procedure. This also causes `aBehavior` to denote a first-class time-varying value in the body of the procedure, such that the interpreter does create a dataflow dependency when used in the procedure's body.

By using AmbientTalk's reflection API, the mirror of the behavior object is retrieved. Every reactive value has a unique mirror object, which allows to uniquely identify the reactive value. This mirror, together with the formal parameters of the procedure are passed to the `findAndUpdateListeningBehaviors` procedure, which is shown below:

```
1  def findAndUpdateListeningBehaviors(
2    value,
3    type,
4    contentCheckClosure := { |x| true },  // by default: always match
5    behaviorMirror) {
6
7    def handleAndAr := behaviorMirrorsHandlesMap.get(behaviorMirror);
8    def ar := nil;
9    if: (nil != handleAndAr) then: {
10     ar := handleAndAr[2];
11     handleAndAr[1].cancel();
12   } else: {
13     ar := ambient: type where: contentCheckClosure;
14   };
15
```

```
16    def newHandle := ar<-setValue(value, self)@All;
17    behaviorMirrorsHandlesMap.put(behaviorMirror, [ newHandle, ar ]);
18
19    object: {
20      def cancel() { newHandle.cancel(); };
21    };
22  };
```

The implementation of publish/subscribe interaction of ambient behaviors is entirely based on ambient references [Van08]. Here, ambient references denote a time-varying group of subscribers. Ambient references are used to broadcast event update messages to all subscribed ambient behaviors: because of the push-driven evaluation strategy of this implementation, it is the publisher that sends update messages to the subscribers. When broadcasting a message over an ambient reference, a handle is returned. Such a handle can be used to cancel the sustained message sends of the ambient reference. In our case, we are first checking if a certain reactive value was already previously published. This is the case if there is a value in the `behaviorMirrorsHandlesMap` hash map for the mirror of the reactive value (which uniquely identifies it). If this is the case, the existing ambient reference is retrieved again from the `handleAndAr` pair and the handle of the old event message broadcasted over the ambient reference is cancelled. In the other case, a new ambient reference is created whose job it will be to refer to all reachable subscribed ambient behaviors. If no `contentCheckClosure` is supplied, the ambient reference just listens for subscribers subscribed to the topic denoted by the type tag `type`. If a `contentCheckClosure` is passed, the ambient reference only denotes those subscribers of which passing their subscription object to the `contentCheckClosure` returns `true`.

Now that we are sure that we have an active ambient reference, we create a new handle for the new event update message by sending the `setValue` message to all subscribed ambient behaviors (by annotating the message send with `@All`). This message, when executed by ambient behaviors, will cause these ambient behaviors to update their internal value with the new value passed to the message, and signal an event (causing an update of its dataflow dependents). This new handle is again put in the `behaviorMirrorsHandlesMap` together with the ambient reference. This procedure will be triggered each time the reactive value passed to it changes, but `value` denotes the non-changing value itself.

The return value of this procedure is a publication object of which the only method cancels the publication of the behavior by canceling the event message broadcasting over the ambient reference using its handle.

6.5.2 Subscribing to Ambient Behaviors

The `ambientBehavior:` and `ambientBehavior:with:` constructs dispatch to two different concrete implementations depending on how they are annotated (i.e. with `@Any` or `@All`). For the sake of brevity, we omit the dispatching process based on these annotations because it requires explaining the deep internals of the AmbientTalk reflection API. Instead, we immediately show the implementation of the functions to which they dispatch.

Singular Ambient Behavior Subscriptions

In the case the ambient behaviors subscription construct is annotated with `@Any`, it dispatches to `ambientBehavior:with:`, which is privately defined in the AmbientTalk

Ambient Behaviors module. Its implementation is shown below:

```
1  def ambientBehavior: type with: subscriptionProperties {
2    def [ theBehavior, subscriptionObject ] :=
3      createBehaviorAndSubscriptionObject();
4
5    AmbientRefModule.export: subscriptionObject
6      as: type
7      with: subscriptionProperties;
8
9    [ theBehavior, subscriptionObject ];
10 };
```

First, the `createBehaviorAndSubscriptionObject` procedure is called which returns two objects: the actual reactive value and a subscription object. Such a subscription object implements a `cancel` method that can be used to cancel the subscription. Both objects are assigned in one go to two variables on line 2. At the end of the procedure, this resulting pair is returned[5]

Before returning from the method, the subscription object is exported using the `export:` construct of the ambient references implementation[6] along with the type tag acting as the subscription topic and the `subscriptionProperties` object passed to the construct. This object allows the programmer to define arbitrary properties about the subscription, which will be sent over the network to potential matching publishers. These publishers can apply a predicate on this object that decides wether the subscription matches (the `contentCheckClosure` of the previous code snippet).

Subscriptions to which no such object is passed (using the plain `ambientBehavior:` construct) always match if the publication topic matches the subscription topic. The only difference is the following line:

```
1    AmbientRefModule.export: subscriptionObject as: type;
```

What remains to be explained is the `createBehaviorAndSubscriptionObject` procedure, shown below.

```
1  def createBehaviorAndSubscriptionObject() {
2    def theEventSource := makeEventSource();
3
4    def theBehavior := hold: theEventSource with: (object: {
5      def updateWithValue(aValue, sender) {
6        aValue;
7      };
8    });
9
10   def subscriptionObject := object: {
11     def subscription := nil;
12     def cancel() {
13       subscription.unexport();
14     };
15     def setValue(aValue, sender) {
16       theEventSource<-updateWithValue(aValue, sender);
17     };
18   };
19
20   [ theBehavior, subscriptionObject ]; };
```

[5] In code shown before we have assumed for simplicity's sake that the construct simply returns the reactive value instead of such a two-tuple.

[6] This `export:` construct works differently than the regular `export:` primitive of AmbientTalk to allow the publish/subscribe interaction style of ambient references.

This procedure creates a reactive value derived from an event source and makes sure that it is updated using the same technique as shown before in section 6.4.1, where events signaled by the RFID event loop update an event source and its derived reactive value. In this case however, the event source and derived reactive value is updated by remote publishers sending it update messages.

First, the event source is created. Subsequently, a behavior depending on that event source is created (using the `hold:with:` AmbientTalk/R primitive explained in section 6.4.1). The update method registered with the event source simply returns the new value that the behavior should have.

Finally, the subscription object that can be used to cancel the subscription is created. This subscription object serves a double purpose: apart from canceling the subscription, publishers that match the subscription obtain a remote reference to this subscription object (using an ambient reference as discussed above). Publishers can then propagate an event update message to this subscription object, which will invoke the `setValue` method, which in its turn will send an asynchronous update message to the event source, causing the dependent reactive value to be updated as well. The update message is sent asynchronously such that it is scheduled in the hosting event loop's message queue and will be executed as the event loop has finished other previously scheduled tasks. Additionally, the message can be sent synchronously by the programmer to for example immediately set the initial value of the ambient behavior to something meaningful. If this initial value is not set, the initial value is `nil` by default and requires dependent reactive code to allow such a `nil` value. Both the resulting reactive value and the subscription object are returned in a two-tuple.

Group-Centric Ambient Behavior Subscriptions

In the case the ambient behaviors subscription construct is annotated with `@All`, the implementation dispatches to `ambientBehaviorCollection:with:timeout:`, which is privately defined in the AmbientTalk Ambient Behaviors module and shown below:

```
1  def ambientBehaviorCollection: type
2    with: subscriptionProperties
3    timeout: theTimeout {
4    def [ theBehavior, subscriptionObject ] :=
5      createCollectionBehaviorAndSubscriptionObject(theTimeout);
6
7    AmbientRefModule.export: subscriptionObject
8      as: type
9      with: subscriptionProperties;
10
11   [ theBehavior, subscriptionObject ];
12 };
```

It works exactly the same as the `ambientBehavior:with:` procedure discussed in the previous section. The only difference is that it calls `createCollectionBehaviorAndSubscriptionObject` instead of `createBehaviorAndSubscriptionObject`. The implementation of that procedure is shown below.

```
1  def createCollectionBehaviorAndExportedObject(refresh) {
2    def theEventSource := makeEventSource();
3    def senderAndValueMap := HashMap.new();
4    def senderAndPingMap := HashMap.new();
5
6    def timerSub := whenever: millisec(refresh) elapsed: {
```

```
7       senderAndPingMap.keySet().toArray().each: { |sender|
8         when: sender<-ping()@Due(millisec(refresh)) becomes: { |ack|
9           senderAndPingMap.put(sender, now());
10        } catch: TimeoutException using: { |e|
11          senderAndPingMap.remove(sender);
12          senderAndValueMap.remove(sender);
13          theEventSource<-updateWithValue(nil, `REMOVED);
14        };
15      };
16    };
17
18    def theBehavior := hold: theEventSource with: (object: {
19      def updateWithValue(aValue, sender) {
20        if: (sender != `REMOVED) then: {
21          senderAndPingMap.put(sender, now());
22          senderAndValueMap.put(sender, aValue);
23        };
24        senderAndValueMap.values().toArray();
25      };
26      def setInitialValue(aValue) {
27        aValue;
28      };
29    });
30
31    def subscriptionObject := object: {
32      def subscription := nil;
33      def cancel() {
34        timerSub.cancel();
35        subscription.unexport();
36      };
37      def setValue(aValue, sender) {
38        theEventSource<-updateWithValue(aValue, sender);
39      };
40    };
41
42    theEventSource.setInitialValue([]);
43    [ theBehavior, subscriptionObject ];
44  };
```

Similarly to the implementation of singular ambient behaviors explained in the previous section, it first creates an event source that is used to update a behavior derived from it. However, this implementation is responsible for sequencing events that are asynchronously received from a group of publishers. The event source will now signal events that do not contain a single value, but a *set* of values received from a group of publishers that all match the subscription. It also associates references to publishers with their latest propagated value and their latest acknowledgment times in two hash maps.

Next, it schedules a task in the event loop using the `whenever:elapsed:` construct, which periodically schedules a parameterless closure passed as a second argument. The periodicity is determined by the time interval passed as a first argument. This is also the timeout period that is used to determine when publishers can be disregarded because they are unresponsive, disconnected, or have experienced other faults.

The parameterless closure iterates over all discovered publishers and sends them an acknowledgement message to which they must reply within the specified timeout period by annotating the message with `@Due`. If no reply to the acknowledgement message is received within the timeout period, an exception is raised and the reference to the publisher is removed from both hash maps: it is assumed the publisher is unreachable or has disappeared. The event source is updated with a special `REMOVED` symbol

that signals that a publisher was removed from the set of connected publishers. When a reply to the acknowledgment message is received within the timeout period, the latest acknowledgement time of that particular publisher is updated with the current time.

The next step consists of creating the behavior derived from the event source, just like in the previous section. However, here the `updateWithValue` method used to update the event source from which it is derived has a different implementation. If the event contains the `REMOVED` symbol as the publisher, it assumes the publisher is disconnected and just returns the associated propagated values as a table (Java arrays are automatically coerced to AmbientTalk tables) from all connected publishers (of which the disconnected publishers are removed). If the publisher that is passed is a valid remote reference to a publisher, the reception of the update message is considered as an acknowledgement and the latest acknowledgement time is updated in the `senderAndPingMap`. Subsequently, the new value that the publisher propagated is associated with it in the `senderAndValueMap`. When the values in the map are returned, they now contain the latest values. There is also a method provided to set the initial value of the behavior to the empty table, which is the most reasonable default initial value.

After that, the subscription object is created, almost identical to the one of the previous section. However, this time the task that periodically pings publishers and listens for their acknowledgments should be cancelled as well.

The final steps of the procedure consist of setting the behavior to the empty set as initial value and a pair containing this reactive set and the corresponding subscription object are returned. As soon as this has happened, the resulting ambient behavior will start to match with discovered publishers, if they match update itself, and if they do not respond remove their propagated latest values from its result set. If a new publisher matches and signals an event, its value is added to the set. The set changes periodically determined by the timeout value that is passed by the programmer (unless if there is no change, then nothing is signaled).

6.5.3 Networking Technology Used by Ambient Behaviors

Our implementation relies on the built-in primitives of AmbientTalk/R and on the Ambient References library [Van08].

AmbientTalk/R borrows its communication primitives from its non-reactive counterpart AmbientTalk. This means that in our implementation of ambient behaviors, messages sent across different event loops to update distributed dataflow dependencies are simple asynchronous messages that transport the new values of dependent ambient behaviors. This can happen over a number of communication technologies, such as Wifi, Bluetooth, etc. These messages are sent over *ambient references* to all subscribed (and reachable) ambient behaviors.

This means that usually event updates are represented as AmbientTalk *isolate objects* (which have no surrounding lexical scope to prevent having to transitively copy it over the network). However, regular AmbientTalk (and hence also Java) objects can also be used as event updates. These will cause the dependent ambient behaviors to receive *far references* to these objects (i.e. they follow the default distributed parameter passing semantics of AmbientTalk that does not allow synchronous access to remote non-isolate objects). This could be useful in some scenarios, but one should keep in mind that sending messages over far references can only happen asynchronously and return values must be captured using a callback (registered on the future returned by the message send).

Ambient references are implemented as a library on top of AmbientTalk and their implementation relies on the M2MI (Many-To-Many Invocations) [KB02] Java framework. Ambient references are used by our implementation to create a time-varying group of subscribers to which update messages can be sent using M2MI. M2MI broadcasts messages to all objects implementing a certain interface, in this case represented by AmbientTalk type tags. Broadcasting happens on the radio level and does not require more power than directed communication. This broadcast-based communication maps well onto our desired broadcast-based publish/subscribe architecture.

Similarly, the subscribers broadcast their subscription to all reachable entities using this framework. Matching subscribers filter out these subscriptions each time they are notified of a newly joined subscriber or a subscriber that is disconnected or unresponsive. The filtered out set of subscribers is the set encapsulated by the ambient reference that is used to multicast event update messages. An alternative approach could be to broadcast publication information to all subscribers. Both approaches may have their benefits and disadvantages in different scenarios, but in general there is no particular reason to prefer one above the other.

Following the formal terminology introduced by Baldoni et al. [BCPV03], our implementation is *minimal*, i.e. the same event is signaled at most once to the same subscriber, but not *complete*, i.e. there are no guarantees that an event will be eventually delivered to a subscriber.

6.5.4 Performance Evaluation

In this section, we discuss the results of a performance comparison between standard AmbientTalk that uses standard asynchronous messages and discovery primitives for event communication, and AmbientTalk/R together with ambient behaviors used for discovery and communication. We used the following synthetic test setup:

- A number n AmbientTalk event loops are spawned.
- Each event loop signals a number m events to each other event loop.
- The value that is being measured is the total number of messages k processed by a single event loop (k is the same for every event loop). Multiplying this value by n yields the total number of messages sent across event loops in the experiment in order to communicate all events.

This gives us an idea of the overhead that our abstractions induce in terms of the amount of extra AmbientTalk messages that have to be sent over the network to communicate the same amount of events. It is measured by using a special AmbientTalk event loop of which the procedure that is called when scheduling a message in its event queue is overridden to increase a counter. At the end of each ran experiment, this counter contains the value of k.

The standard AmbientTalk version consists of n event loops that each host a single object that responds to a `notify` message, implementing a callback for an incoming asynchronous message. Every event loop exports this object under the same type tag. Additionally, these objects register a `whenever:discovered:` callback listening for that type tag. The effect is that every object will obtain a far reference to every other object hosted on the other event loops. Subsequently, every event loop sends m asynchronous `notify` messages containing a single dummy event over every far reference obtained in the previous step. This means that each remotely hosted object is notified of m

events. Of course, we expect the number of messages `k` received per event loop to be only slightly more than to the number of events generated by the other event loops ($(n-1).m$).

The version implemented using ambient behaviors consists of n event loops that each host a single reactive value, implementing a single mutator method. This reactive value is exported by every event loop using the `exportBehavior:as:` construct. Similarly, every event loop subscribes a single ambient behavior using the `ambientBehavior:` construct using the same type tag. Finally, every event loop invokes m times the mutator method of their reactive value, resulting in m events broadcasted to all other event loops, which in response update their subscribed ambient behaviors.

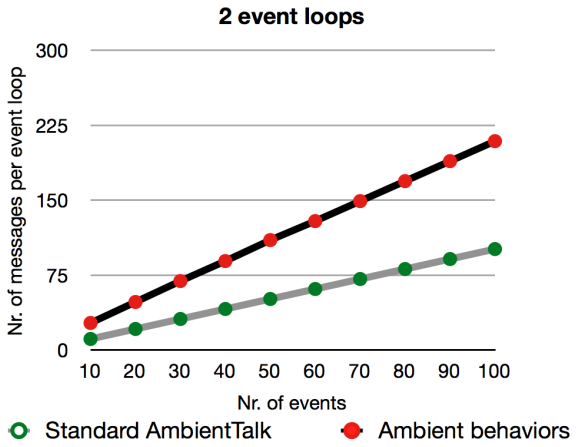

Figure 6.8: Number of messages processed per event loop with two communicating event loops ($n = 2$ and m is varied from 10 to 100).

In short, the AmbientTalk version invokes all remote listener objects' callback methods using asynchronous messages representing event notifications, while the ambient-oriented dataflow version subscribes an ambient behavior that is used to handle the same m events.

Given this setup, k is plotted on the following three graphs each time for increasing m. Figure 6.8 shows the results for two event loops, figure 6.9 for four event loops, and figure 6.10 for six event loops. The horizontal axis shows an increasing m (i.e. the number of signaled events per event loop), while the vertical axis shows the resulting values for k (i.e. the number of processed messages per event loop).

Note that in this experiment all event loops were hosted on a single JVM and no disconnection can occur. Hence, no messages are lost and the lower k, the better (less messages are sent).

We conducted all experiments both for singular ambient behaviors as for group-centric ambient behaviors. The results differed a constant t from each other, where t is the number of ping and acknowledge messages (of which an equal amount is generated since we are not emulating disconnections) the group-centric ambient behaviors exchange while the experiment is running. Taking a very large timeout for these group-centric behaviors resulted in the same k as for the experiment with singular ambient behaviors. This is the expected result since even the singular ambient behaviors re-

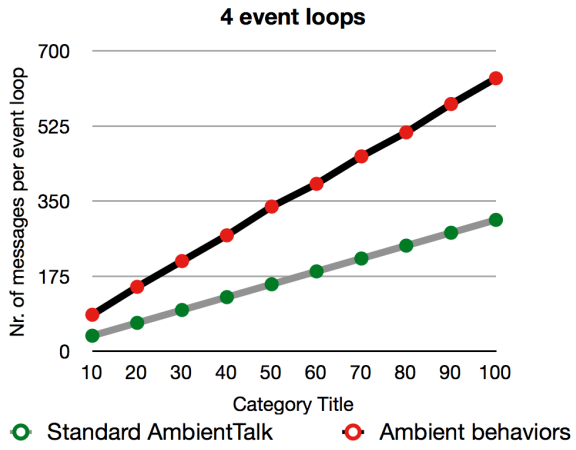

Figure 6.9: Number of messages processed per event loop with four communicating event loops ($n = 4$ and m is varied from 10 to 100).

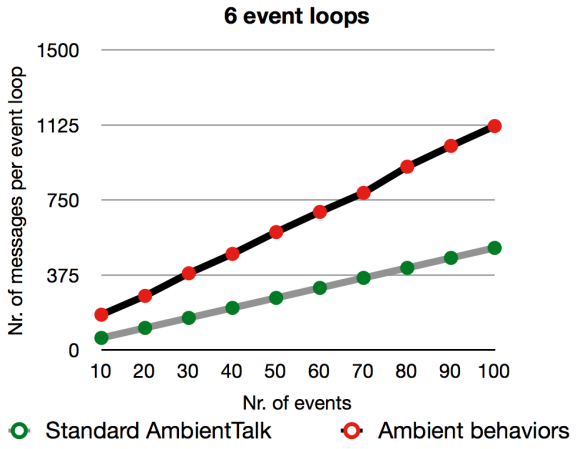

Figure 6.10: Number of messages processed per event loop with six communicating event loops ($n = 6$ and m is varied from 10 to 100).

ceive events from all other event loops. For this reason, we have omitted the results for group-centric ambient behaviors.

The other conclusions we can draw from these experiments are the following:

- Both the AmbientTalk experiment and the experiment using ambient behaviors scale linearly with respect to both the number of signaled events per event loop (m) and the number of event loops (n) broadcasting and reacting to events. Other values for n yielded similar results as the ones shown here. This concretely means that a higher number of interacting parties or a higher number of events being signaled do not generate an excessive amount of messages in both approaches.

- The version using ambient behaviors needs about twice the amount of AmbientTalk asynchronous messages to communicate the same amount of events to the same amount of event loops. These extra messages are acknowledgement messages generated by the ambient references library [Van08] (used by our implementation) to check the availability of other peers.

6.6 Limitations

As a first issue, it is clear that the reactive programming system of AmbientTalk/R comes with an overhead in terms of computational resources compared to the plain AmbientTalk interpreter. More (dataflow) events are scheduled in the various event loops of an application and more memory is being consumed to keep track of the different dataflow dependencies in a local application. We also observe that, with respect to pure processing power, AmbientTalk and AmbientTalk/R are targeted towards highly networked applications where the network forms the performance bottleneck instead of the performance of the interpreter.

Some of the limitations stem directly from the inherent properties of distributed systems. Because of both the unreliability of the connections between the devices and the unpredictable delays on the arrival of messages at remote parties, our system cannot provide real-time guarantees on the processing and reacting to events. Furthermore, when a message is sent from one device before a different message is sent from another device, the underlying AmbientTalk virtual machines do not guarantee that these messages will arrive at their destination in that same order. Providing such guarantees would involve keeping a global clock over all the distributed virtual machines (an assumption that is for example made in the GEM event monitoring language [MsS97]) to time-stamp events, which is impractical in this setting. The programmer has to take this into consideration if causality between events has to be inferred (i.e. using ambient behaviors, distributed glitches cannot be prevented by the runtime). However, our system focuses on applications that work with human time scales (e.g. seconds, minutes), so slightly drifting distributed clocks are tolerable.

Currently, there is no way for an event consumer to tell its event producer to limit the events that it wants to receive: an ambient behavior publication or subscription can only be cancelled. Afterwards, the subscription can be re-established. We have to investigate whether this can lead to network congestion or performance issues on the device that acts as event consumer.

Finally, the naming and discovery of services currently happens via Java interfaces represented as AmbientTalk type tags. We make the underlying assumption that the

name of such Java interfaces represents a unique service and is known by all participating services. This discovery mechanism also does not take versioning into account explicitly. For example, if the `TicketVendor` from the example in section 4.1 is updated, older clients may discover the updated service, and clients that want to use only the updated service may still discover older versions. Clients and services are thus themselves responsible to check versioning constraints.

6.7 Conclusion

We have presented a number of language constructs that reconcile the loose coupling of a distributed publish/subscribe architecture and the elegant event processing of a reactive programming language, AmbientTalk/R. Concretely, the dataflow graphs constructed by the AmbientTalk/R interpreter to keep track of dataflow dependencies can now be seamlessly distributed by means of ambient behaviors, which is a new language construct added to the language. The distributed dataflow dependencies are implemented on top of a decentralized publish/subscribe architecture to achieve a very loose coupling between the dependents (event consumers) and their progenitors (event producers). Hence, event producers can be dynamically replaced at run-time when they become unreachable due to network partitions, or the events signaled by multiple event producers can be aggregated into a single reactive value, that can subsequently be processed using dedicated group-centric operators. By adopting the reactive programming paradigm, the reception of events can be represented as (external) updates to a reactive value. Such updates are propagated implicitly to all relevant parts of the application. Hence, it is possible to react trivially to external events without the inversion of control that would result from having to resort to the use of explicit callbacks.

Furthermore, we have demonstrated that our dataflow constructs can be seamlessly combined with programming model for mobile RFID-enabled applications described in chapter 5, such that these applications can be effectively programmed without suffering from inversion of control.

In section 6.4.3, we observed that the purely broadcast-based communication primitives that we proposed in this chapter often require subscribers to manually filter out events that were not intended for them. Additionally, although the problem of inversion control is for the most part eliminated, the control flow of a distributed reactive program can still be hard to grasp due to the combination of asynchronous event delivery, publisher-subscriber bindings that change at runtime, and reactive code that is triggered because of events. To alleviate this problem, in the next chapter we introduce an abstraction that offers a network-centric view on the networked application, by representing the global ambient-oriented dataflow program visually, and offering dedicated primitives with a visual notation to specify how publishers and subscribers are bound and how events are propagated over dataflow dependencies. This network-centric visual language is compiled into a dataflow network where the distributed dependencies are represented as ambient behaviors.

Chapter 7

Network-Centric Visual Dataflow Programming

In this chapter, we introduce AmbientTalk/R^V: a network-centric visual ambient-oriented dataflow language. It relies on the node-centric and group-centric ambient-oriented dataflow abstractions introduced in the previous chapter. This way, a network-centric graphical view on the mobile RFID-enabled application is reconciled with the loosely-coupled nature of the underlying broadcast-based publish/subscribe architecture.

We first motivate our work and give a brief overview of the concept of coordination via coarse-grained dataflow in section 7.1. Subsequently, in section 7.2 we give an overview of the different concepts in AmbientTalk/R^V together with their visual representation using the RFID-enabled Book Recommender application as an example. In section 7.3, we discuss a slightly different flavor of AmbientTalk/R^V that we implemented with minimal changes to the original concepts to allow dataflow operators to publish multiple events. After that, in section 7.4 we detail our prototype visual programming environment which allows the graphical specification of AmbientTalk/R^V dataflow graphs. The implementation of both the language and the programming environment is discussed in section 7.5 after which the limitations of our approach are discussed in section 7.6. Finally, section 7.7 concludes this chapter.

7.1 Motivation

In the previous chapter, we shifted the execution model of a mobile RFID-enabled application from stack-based sequential programs communicating via asynchronous messages, to a distributed dataflow program where distributed dataflow components are dynamically bound using a broadcast-based publish/subscribe architecture. This causes dataflow components to be loosely-coupled and prevents reactive code to be structured around explicit callbacks.

Recall from section 6.4.3 from the previous chapter that our approach failed to satisfy one of the programming model requirements for mobile RFID-enabled applications listed in chapter 2 section 2.3.4: **global control flow management**. We additionally observed that the broadcast-based publish/subscribe mechanism that we employed requires the various peers of a distributed application consisting of multiple heteroge-

nous peers (such as the Book Recommender application) to filter out event messages that were not intended for them, but for other peers pertaining to a different *instance* of a dataflow program. In short, the drawbacks of the approach discussed in the previous chapter are that there is no network-centric view on the global mobile reactive application and that the structure of the program maps poorly on a sequential, textual representation.

In this chapter we aim to alleviate these last two shortcomings while keeping the other programming model requirements satisfied. For this, we draw inspiration from the programming systems discussed in chapter 2 section 3.3.2 and 3.4.3 to represent mobile RFID-enabled applications on a network-centric perspective by offering a visual language in which the programmer specifies the (distributed) dataflow graph representing the application *directly*. This way, the distributed control flow of the application is made explicit visually such that it can be changed without having to adapt the fine-grained dataflow components. As we will show, the effect of a single change in the visual network-centric dataflow program would require multiple changes in a node-centric textual version of that same program.

The choice for a visual language is motivated by the fact that a unified data flow and control flow lend themselves very well for a visual representation, as observed amongst others in the business process modeling and workflow community [PA05, MPMJPPS05]. Whitley conducted a study summarizing the at that time available results of comparisons between textually and visually represented programs [Whi97], concluding the following:

> "People in design and problem-solving situations perform better when information is presented in a consistent and organized manner. Furthermore, the more efficient representations tend to be the ones that make information explicit. These two guidelines apply to all notations, including textual ones. The studies [...] indicate that, compared to textual notations, visual notations can provide better organization and can make information explicit."

Similarly, Baroth and Hartsough conducted a study [BH95] where the visual dataflow language LabVIEW [Kal95] was compared to C in the context of a specific but real-world problem domain (lab data acquisition). They found that visual dataflow programming reduced development time and improved communication among developers and from developers to consumers and vice versa.

7.1.1 Visual Dataflow as a Coordination Paradigm

Gelernter and Carriero [GC92] argue that a complete programming model consists of both a *computation model* and a *coordination model*. The computation model allows programmers to build a single computational activity (e.g. a process, a thread, an actor in an actor language). The coordination model is the glue that binds separate activities into a concurrent application.

An ordinary computation language embodies a computation model. Several concurrent languages in addition provide a coordination model that ranges over different levels of abstraction, from manual thread creation and locking to event-based communication abstractions. An example of the latter can be found in the AmbientTalk programming language used throughout this dissertation.

A coordination language embodies a coordination model; it provides operations to *create* computational activities and to support *communication* and *synchronization*

among them. In this dissertation, we require the coordination model to be applicable in mobile ad hoc networks, meaning that the model should be resilient to network partitioning and reactive to network topology changes. In short, it should support the ambient-oriented programming paradigm.

The Dataflow Coordination Model

Before the Von Neumann architectures took over the parallel programming world (after becoming the de facto standard computing architecture in the sequential world), dataflow languages were popular for programming massively parallel systems to be built on top of a dataflow hardware architecture [JHM04]. By making data dependencies explicit, these languages and hardware architectures allowed a high degree of parallelization while preventing race conditions and other problems arising when parallelizing programs intended for Von Neumann architectures. Moreover, the resulting dataflow graphs are easy to visualize, allowing a visual representation of both the data dependencies as well as the coordination they imply on the different parallel components of the application. The dataflow coordination model can be informally described as follows:

- Dataflow programs consist of *dataflow operators* that take a number of input values and return a single output value. These dataflow operators are best compared to functions or procedures in functional or imperative programming languages that always run in parallel.

- Dataflow operators communicate with each other over *dataflow edges*. These edges represent data dependencies and always flow from the output of a dataflow operator (corresponding to its output value) to one of the inputs of a dataflow operator (corresponding to one if its input values). Functional reactive programming languages (such as the ones discussed in chapter 2 section 3.3.1) and AmbientTalk/R (discussed in the previous chapter) track data dependencies *implicitly*. In languages where dataflow is used for coordination, data dependencies are made *explicit* by representing them as dataflow edges in the dataflow graph.

- When exactly a dataflow operator is fired depends on the concrete coordination model used. Some languages only fire dataflow operators once as soon as *all* their input values have received a value. We call this model *synchronous dataflow*. These languages are usually *synchronous* languages that assume a global clock. Other languages repeatedly fire the dataflow operator as soon as *one* of its input values received a *new* value. We call this model *asynchronous dataflow*. Example languages are functional reactive programming languages. Both categories are discussed in chapter 2 sections 3.3.1 and 3.3.2.

Such a coordination model allows various dataflow operators in the dataflow graph to execute in parallel as long as their data dependencies are satisfied. For example a number of operators in a pipeline execute in parallel when the first operator is fed a stream of data. In such a pipeline the first operator is being applied to new data from the stream while operators later in the sequence are being applied to data already processed by earlier dataflow operators in the pipeline.

Currently, the dataflow paradigm is mostly used in the form of the *coarse-grained* dataflow model, as can be seen from some of the systems discussed in chapter 2 section 3.3.2. In this model, the dataflow paradigm is used to orchestrate the control flow

between different modules (possibly running in parallel) that can be of an arbitrary level of abstraction, usually implemented in a conventional programming language.

When looking at the characteristics and requirements of mobile RFID-enabled applications, we have observed that the dataflow model may provide a very suitable coordination model for this kind of applications. These applications consist of different distributed components running in parallel that in many cases have to be invoked whenever some external data is fed to them (event-driven architectures). Hence, the driving force for program execution in such applications is not the control flow, which is explicitized by the order of statements in an imperative textual program, but the data flow, which is implicit in an imperative textual program.

Coordination based on dataflow allows to explicitly program the data flow of an application, which is unified with an implicitly handled control flow. Therefore, this chapter proposes ambient-oriented dataflow as a coordination paradigm to satisfy our final requirement for programming mobile RFID-enabled applications: **global and automatic control flow management**.

7.2 Visual Dataflow Programming and AmbientTalk/R^V

Figure 7.1 shows the general idea behind our visual dataflow language, which we call AmbientTalk/R^V. AmbientTalk/R^V uses the boxes-and-arrows notation to denote dataflow operators and dataflow dependencies between them respectively. The

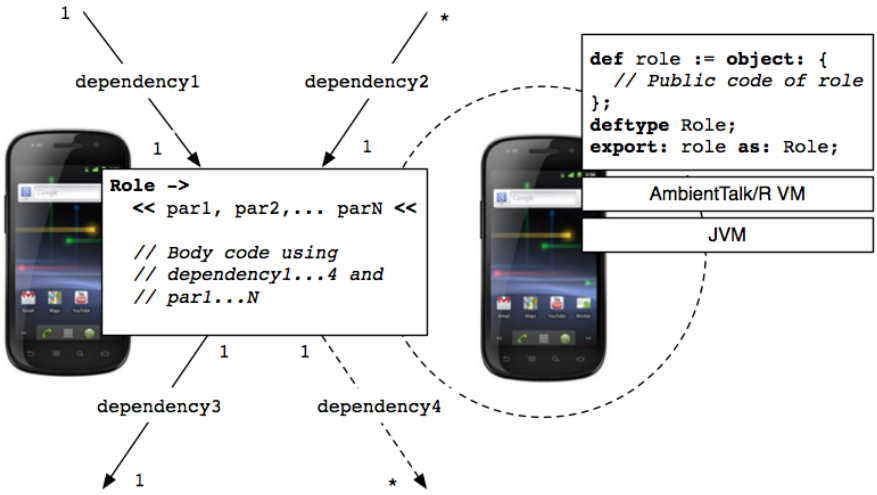

Figure 7.1: Represention of a dataflow operator in AmbientTalk/R^V.

identifier before the -> symbol denotes the *role* of the operator (for now, it suffices to think of a role as a procedure name). After the role name, its list of state variables are declared between the << delimiters. The code after this list can be any sequence of AmbientTalk expressions (hence our visual dataflow language could be considered a *hybrid* language [JHM04]) and comprises the body of the dataflow operator, which serves as its implementation. This code is automatically parametrized by "pseudo"-variables that are bound to the dataflow input values (`dependency1` to `dependency4`) of the dataflow operator. We will name these variables *dataflow parameters*. This is

achieved by naming the edges, such that these names can be used as the names of the dataflow parameters in the dataflow operator implementation. The dataflow operator firing rule in our visual languages is based on the asynchronous dataflow model: *any new value propagated along an incoming dataflow edge results in reapplying the dataflow operator with the new value of the dataflow variable*.

Optionally, one can declare a number of local *state variables* (`par1` to `parN`) that can be used in the body of the dataflow operator. They initially have the `nil` value, but can be assigned in the body of the dataflow operator. In any *subsequent* execution of the operator (i.e. when a new event is signaled over its incoming dataflow edges), these parameters will have the values assigned to them in the *previous* execution of the operator. This is used to allow state to be used across dataflow operator executions and is further explained in section 7.2.6.

Executing a dataflow program happens by distributing the dataflow operator code to devices that match the roles designated to the operators in the graph and installing communication channels that represent the dependency edges between them. Dependency edges can be either *fixed* (uninterrupted lines) or *rebinding* (dashed lines). The service discovery needed for this is further explained in section 7.2.2. To take part in such a network-centric AmbientTalk/R^V application, these devices should have an AmbientTalk/R virtual machine running on top of a Java virtual machine, and additionally host the necessary library code to execute their role code. The code associated with a role is mobile AmbientTalk/R code that can call any AmbientTalk/R or Java library code that is made visible to it by the device on which the dataflow operator code is executed.

This is depicted in figure 7.1. On the left-hand side, it shows a device hosting an AmbientTalk/R^V dataflow program. On the right-hand side, another device is shown that hosts an AmientTalk/R interpreter (on top of a Java Virtual Machine). The interpreter in its turn hosts a special *operator host* object that exposes public code to be used within the role code of the dataflow operator. At deployment time, the device hosting the dataflow program will attempt to contact such a device that matches one of the roles. If successful, the role code will be sent to this device and be executed on that device by relying on the infrastructure offered by the operator host object. This is further explained in section 7.2.3.

Recall that one of our programming model requirements for mobile RFID-enabled applications was arity-decoupled communication. Finally, to represent different communication styles, we extended the basic dataflow coordination model with dependency arities that allow dataflow dependencies to be one-to-one, one-to-many, many-to-one, many-to-many. This is indicated by the programmer by annotations at the start point or end point of the graph edges and is further explained in section 7.2.5.

AmbientTalk/R^V allows to visually specify a network-centric dataflow program of which the dataflow nodes are parametrized by reactive values that denote dataflow parameters which are dependent on the output of remote dataflow nodes. This model allows for a straight-forward compilation to distributed node-level code that consists of distributed reactive code snippets acting as roles in the dataflow program and that are driven by a number of ambient behaviors (explained in the previous chapter) which represent dataflow parameters. These ambient behaviors can be correctly subscribed to other distributed dataflow operators by relying on the information supplied by the programmer in the AmbientTalk/R^V dataflow program. The result is that the programming model requirements that were met by ambient behaviors in the previous chapter are still satisfied, but in addition AmbientTalk/R^V offers a network-centric view with global control flow.

7.2.1 The Book Recommender Application in AmbientTalk/R^V

The AmbientTalk/R^V program shown in figure 7.2 is the mobile RFID-enabled Book Recommender scenario that we introduced earlier in section 6.1.2 of the previous chapter implemented in AmbientTalk/R^V. In this example, the Java package paths to the methods that are invoked in the role code are omitted for the sake of brevity, but other than that the program shown here is entirely functional. Dataflow operators are represented as boxes while the directed edges connecting these boxes represent data dependencies.

Similarly to its node-centric textual counterpart, it is a distributed reactive application in which various distributed components are subscribed to remotely signaled events. The `BookScanner` dataflow node signals the set of scanned books over two dataflow edges, which are named `books` and `scannedBooks`. These one-to-one fixed edges propagate these sets of books to the `BookRecommender` and `EBookReader` nodes respectively. In response to changes in the set of books, they reexecute their role. For the `BookRecommender` node, this causes the set of scanned books to be updated in the user interface. For the `EBookReader` node, this causes the set of recommended books to be recomputed and to be propagated over its two outgoing edges: `recommended` and `similarBooks`. The `recommended` fixed one-to-one edge causes the set of recommended books to be propagated to always the same `BookRecommender` node, which in its turn will show the new set of recommended books in its user interface. The `similarBooks` rebinding one-to-many edge causes the set of recommended books to be broadcasted to all `Shelf` nodes in range. In response, the `Shelf` nodes will filter out the set of books they contain out of the set of recommended books and propagate this filtered set over the `recommendedInStock` fixed one-to-one edge to always the same `BookRecommender` node. The result is that the `BookRecommender` node will receive personalized recommendations present in the library for its user from a single shelf. However, because of the `similarBooks` rebinding one-to-many edge, this may be a different shelf of the library depending on in which shelf's range the `BookRecommender` node is roaming.

In the remainder of this chapter, we will use this application as a running example to explain the concepts and their semantics in AmbientTalk/R^V in more detail.

7.2.2 Discovering Operator Nodes

AmbientTalk/R^V serves the purpose of coordinating event-driven application components running in parallel and distributed over a mobile ad hoc network. Because the devices hosting these application components can move out of range and back into range of each other at any point in time, our dataflow engine has to discover these application components at runtime without relying on naming servers or other fixed infrastructure, as explained in chapter 2 section 2.1.1. In AmbientTalk/R^V, mobile application components that play a role in a dataflow program are discovered based on their *roles*. These roles actually have the same use (and in fact are implemented this way) as the type tags that are used by AmbientTalk/R to discover remote objects and hence act as the topics that are being used by the ad hoc publish/subscribe architecture. The system uses the built-in service discovery mechanisms of AmbientTalk/R to both advertise dataflow nodes and as well devices willing to execute one or more of the roles in the dataflow program. This is an entirely decentralized approach that does not assume any other infrastructure but the mobile devices themselves.

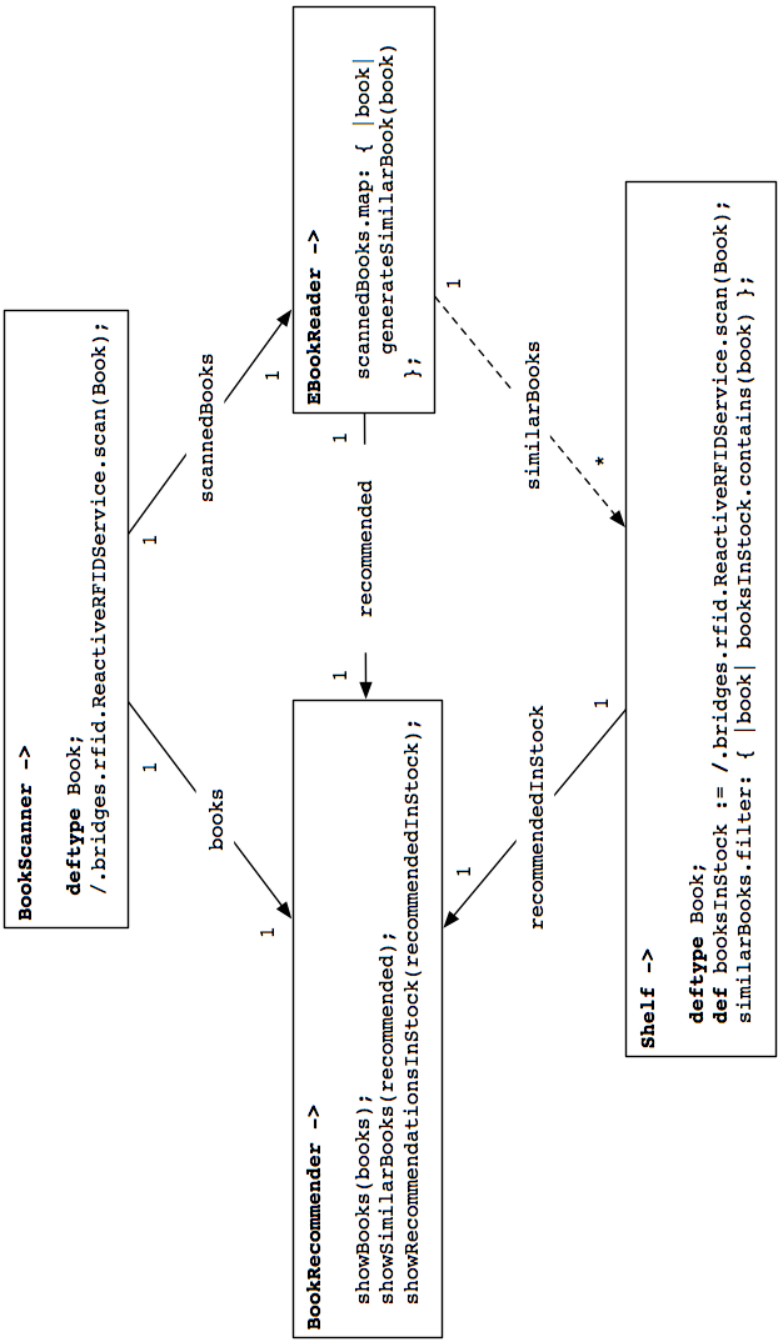

Figure 7.2: AmbientTalk/R^V implementation of the RFID-enabled Book Recommender application.

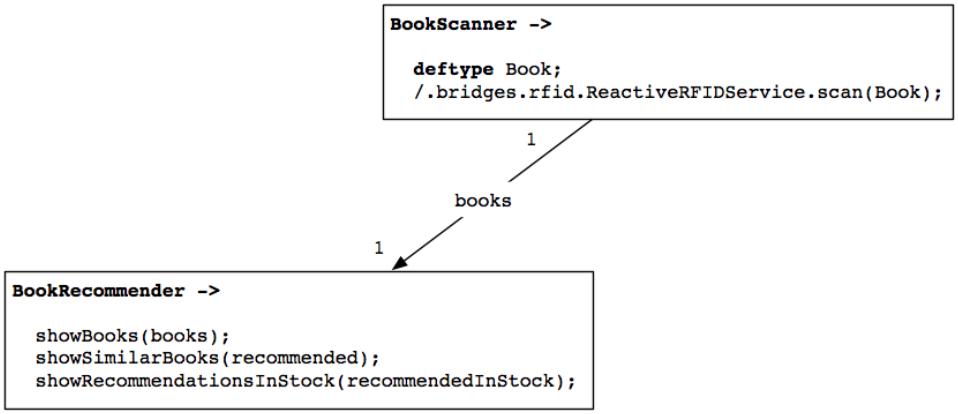

Figure 7.3: Fixed dataflow edge from `BookScanner` to `BookRecommender`.

Fixed Edges

Depending on the kind of dataflow dependency edge drawn between two operators, the discovery mechanism works differently. In the case of *fixed* dataflow edges, such as between the `BookScanner` node and the `BookRecommender` node shown in figure 7.3, once the `BookRecommender` node is discovered, the same instance of the dataflow program expects always the same instance of a `BookRecommender` node. This means concretely that when the connection is lost with the `BookRecommender` node, the `BookScanner` node will wait until the connection is restored with the original `BookRecommender` and not look for a replacement node. This makes sense if there is a stateful relationship between both distributed nodes in the dataflow program, e.g. if users should receive personalized recommendations and not recommended products from different users. We observed such a stateful relation in the node-centric implementation of the application in chapter 6 section 6.4. Our node-centric implementation required clumsy event filtering and connection tracking, which are now circumvented by the automatic management of dependencies on the network level by AmbientTalk/R^V. To detect a permanent disconnection the dataflow dependency edge can be annotated with a timeout period. Event messages are buffered until the timeout is signaled[1]. In this case, a `TimeoutException` is raised that can be caught from the role code in which the exception was raised.

Rebinding Edges

In some cases a different operator node encoding the same role can be used as a substitute. This is typically the case when there is no state associated with the operator's execution. This is catered for by *rebinding* dataflow edges, which are represented by a dashed line, such as the one between the `EBookReader` node and the `Shelf` node as shown in figure 7.4. In this scenario, the shop may consist of different shelves which are all represented as `Shelf` nodes. Concretely, a rebinding dataflow edge allows rebinding the dataflow dependency to another subscribed operator at runtime, because the network topology has changed for example. Again, a timeout can be specified that determines how long event messages are buffered. In this example, the user might have

[1] Buffer overflows can happen in theory for very large timeout periods and will raise a Java exception.

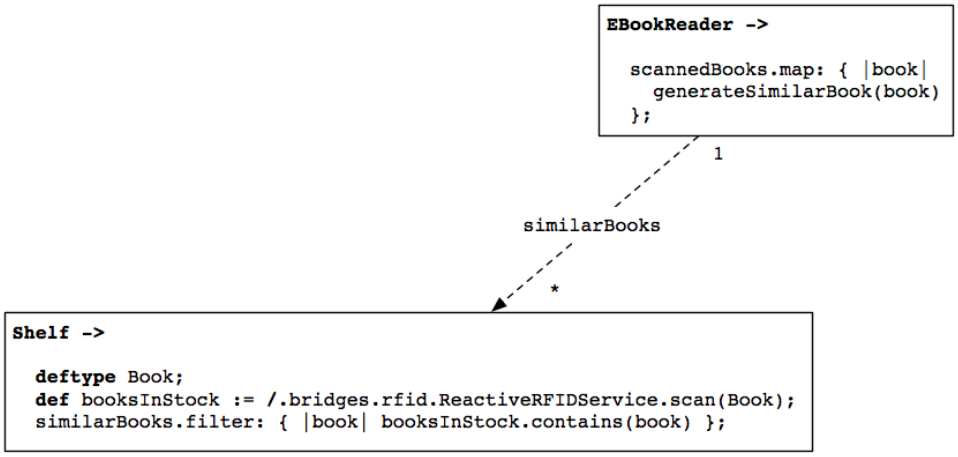

Figure 7.4: Rebinding dataflow edge from `EBookReader` to `Shelf`.

moved out of range of a shelf in the shop and moved into communication range of a different shelf. How the group communication to the different shelves is handled is discussed later in section 7.2.5.

Note that by differentiating between *fixed edges* and *rebinding edges* (representing two different kinds of dataflow dependencies), the awkward event filtering and connection tracking mechanisms in the node-centric implementation of the Book Recommender application (shown in chapter 6 section 6.4) are now replaced by dedicated constructs with clear semantics.

7.2.3 Executing Mobile AmbientTalk/R Code

Dataflow operators operate on event streams. In most cases, it makes sense to process the stream on the device that hosts the operator through which the stream flows, to reduce communication overhead and to avoid a performance bottleneck when all processing happens on a single device. In such scenarios, it is thus cheaper to move the processing code towards the data than the data stream itself towards where the processing code is hosted. Furthermore, in the face of intermittent network connections, long-lasting computations can continue while the network connections with other nodes is temporarily broken, and flush buffered return values when the network connection is restored. This is why when deploying a dataflow operator, its role code is sent as mobile AmbientTalk/R code to the host executing the operator.

To be able to execute roles, hosts need to provide some existing infrastructure in the form of some pre-implemented AmbientTalk/R or Java methods that are published in a service object under a topic matching their role name. This means that services playing a role in an AmbientTalk/R^V program are usually implemented as objects that provide an interface that can be called by the mobile code. This is shown in our example (figure 7.2) where the public `generateSimilarBook` method is assumed to be implemented by the `EBookReader` nodes, the `BookScanner` and `Shelf` nodes are assumed to have public RFID libraries available, and the `BookRecommender` nodes are assumed to have public GUI methods to display books, as shown in figure 7.5[2].

[2]One can designate a dedicated namespace that is visible to the mobile code and organize the library code

```
BookRecommender ->

  showBooks(books);
  showSimilarBooks(recommended);
  showRecommendationsInStock(recommendedInStock);
```

Figure 7.5: `BookRecommender` role using public code of its host.

To create such service objects which contain public code to be called from an AmbientTalk/R^V role, we provide a basic `OperatorHostInterface` object. Custom implementations specific to the device hosting the service can be built (e.g. to impose restrictions on the received mobile code to enforce security policies) by overriding specific methods on the `OperatorHostInterface`. Usually, this object will be extended with the definitions visible to AmbientTalk/R^V roles it is intended to execute.

The code snippet below shows how a host can advertise itself as a `BookRecommender` by simply exporting a service object implementing the `BookRecommender` role (hence, this is local code present on the customer's machine), matching the `BookRecommender` role shown in figure 7.5.

```
1  def bookRecommender := extend: OperatorHostInterface with:  {
2    // ... Fields ...
3
4    def showBooks(books) {
5      // Update GUI with the new books the customer is carrying.
6    };
7
8    def showSimilarBooks(books) {
9      // Update GUI with similar books to the ones being carried.
10   };
11
12   def showRecommendationsInStock(books) {
13     // Update GUI with recommended books present in the library.
14   };
15 };
16
17 deftype BookRecommender;
18 export: bookRecommender as: BookRecommender;
```

The `bookRecommender` object extends from the `OperatorHostInterface` discussed above and implements the public methods to fulfill its role in the application. These public methods are used in the `BookRecommender` role shown in figure 7.5. The last two lines declare an AmbientTalk/R type tag that will be used as the publication's topic and publish the service, such that it can be discovered by the device deploying the network-centric AmbientTalk/R^V program. The type tag under which the service is exported should match the role name of the AmbientTalk/R^V role that is sent when the device that invoked the AmbientTalk/R^V program detects the service. Once the dataflow operator is deployed on the host, the connection with the device that initiated the AmbientTalk/R^V program should only be made by other services (or possibly the same service in case of a disconnection) to either fill in new roles in the graph or contribute to a group of equivalent roles (see section 7.2.5 for groups of roles).

Note that the same device hosting a number of application components can play a role in different dataflow programs. However, race conditions cannot occur because

to be called by the mobile code in Java packages.

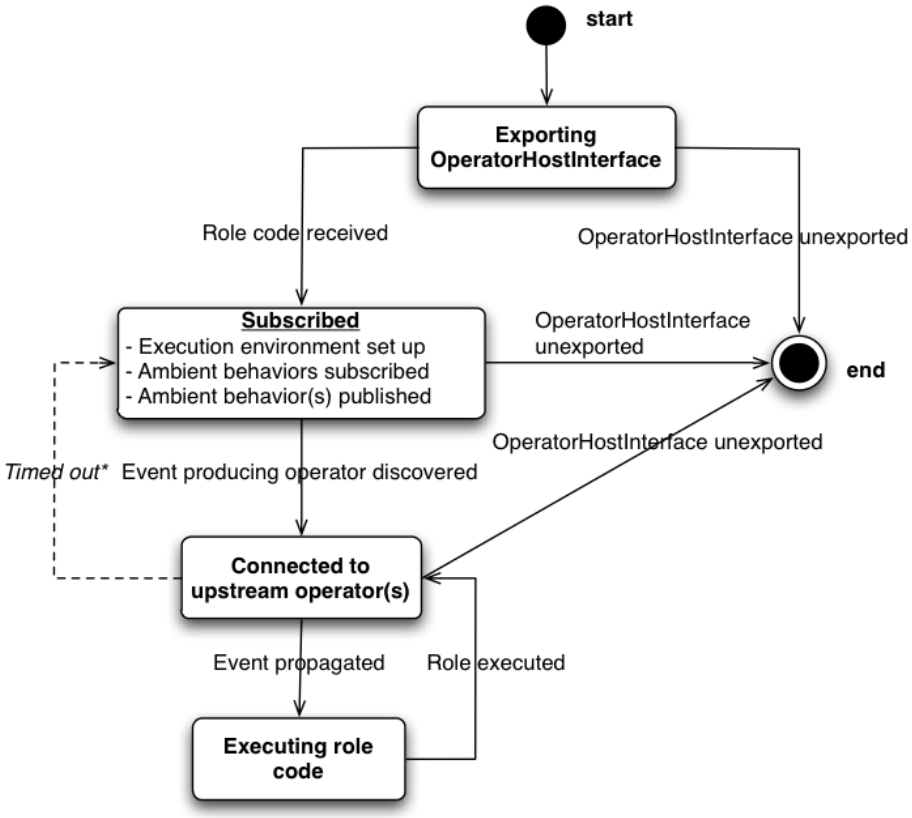

Figure 7.6: Lifecycle of a dataflow operator host.

the communication between all dataflow operator nodes happens by means of asynchronous AmbientTalk/R messages that are scheduled in the event loop of each host, and are sequentially executed by a single thread (causing the sequential execution of the operator code as well).

Lifecycle of a Dataflow Operator Host

In this section, we summarize the lifecycle of a dataflow operator host. It is presented schematically in the state diagram depicted in figure 7.6.

First, (a child object of) the `OperatorHostInterface` object mentioned above must be exported. It contains the necessary infrastructure to execute a role in the AmbientTalk/R^V dataflow program and has access to public code that is invoked by the role. At any moment in time, the device hosting the `OperatorHostInterface` object can unexport the object, which means that the device can no longer participate in the dataflow program.

In case the `OperatorHostInterface` object is exported, it can be discovered by the device hosting the implementation of the dataflow graph. If the `OperatorHostInterface` object in question matches a role, this device will send it the role (isolate) object containing the role code. In case of group-centric subscriptions, all matching `OperatorHostInterface` objects are sent their role object. In case of a singular sub-

scription, this will only happen when the role is not (yet) assigned to any node.

When the `OperatorHostInterface` object receives the role object, it sets up its execution environment and in this environment subscribes the necessary ambient behaviors that represent dataflow parameters in the role code. Additionally, it also publishes the ambient behavior that signals events over the outgoing edges (in case of the variation on the language described in section 7.3, it will not be one ambient behavior, but multiple ones).

As soon as an upstream event producing operator is discovered (i.e. because it published matching ambient behaviors as well), it can signal events to the subscribed ambient behaviors. When this happens, the dependent role code will be (re)executed. If this caused a value change, the affected published ambient behaviors will signal an event to their subscribed downstream operators.

In case a dataflow edge is unable to propagate events because of a disconnection, events are buffered on the propagating node. They are buffered as long as the edge timeout period does not elapse. When the timeout period elapses, a `TimedOutException` is raised in the execution environment of the role[3]. How the exception is handled is determined by a programmer-supplied exception handler. Therefore, this is not depicted in the state diagram. If there is no timeout specified for the disconnected edge, the dataflow operator simply goes back to the "subscribed" state and waits for new matching dataflow operators to be discovered.

Before the timeout period elapsed, there are two possibilities. If the timed out edge is a fixed dataflow edge, the dataflow operator will not try to discover matching replacement operators and wait out the timeout period to reestablish a connection with the same operator. If the timed out edge is a rebinding dataflow edge, the dataflow operator will attempt to discover a matching replacement dataflow operator and – if this successfully happens before the timeout period elapses – continue execution.

In the following sections, we explain how this semantics is represented in AmbientTalk/R^V in greater detail by means of our Book Recommender example program.

7.2.4 Propagating Events and Reacting to Events

Until this point, we have not elaborated yet on how the actual AmbientTalk/R^V program is executed by our dataflow engine. This is based on the AmbientTalk/R interpreter and the node-centric and group-centric dataflow abstractions that we introduced in the previous chapter. Concretely, the dataflow parameters (e.g. `books`, `recommended` and `recommendedInStock` in the `BookRecommender` role) in the role code denote ambient behaviors (see chapter 6 section 6.3.1). These dataflow parameters are bound to ambient behaviors that are subscribed to a topic with the same name as the dataflow variable and the corresponding dataflow edge (representing a dataflow dependency). Hence, dataflow parameters denote reactive values. These reactive values are updated each time their respective input value changes (by new data objects flowing over the dataflow edges corresponding to the input values).

Analogous to reactive programming and following the asynchronous dataflow model described in section 7.1.1, a dataflow operator is re-executed as soon as one of the reactive values it depends on changes. Dataflow updates are signaled simply by executing a dataflow operator, which results in a new return value for the executed dataflow op-

[3]Timing out for group-centric subscriptions happens when the number of matching operators reaches zero and does not increase for the timeout period.

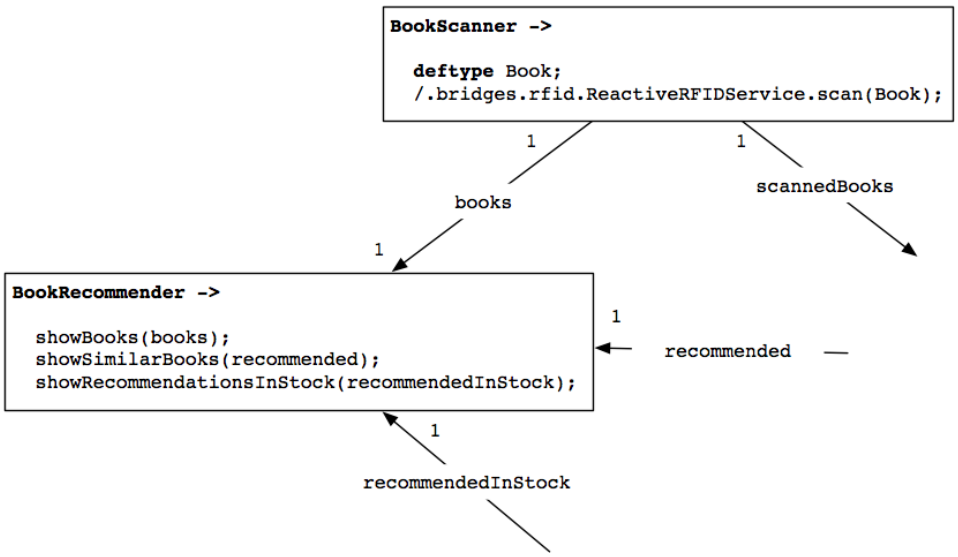

Figure 7.7: `BookScanner` role publishing events under topic `books` and `BookRecommender` role subscribing `books`, `recommended`, and `recommendedInStock` dataflow variables.

erator This value is in its turn propagated over all the outgoing dataflow edges. For example, in the `BookScanner` role shown in figure 7.7, the scanned books are periodically updated by the RFID reader by periodically scanning its surroundings for tags and updating the corresponding reactive value. It is propagated along the `books` and `scannedBooks` edges, causing the invocation of the rest of the dataflow graph.

In the other direction, the `BookRecommender` node will observe updates of its `books`, `recommended` and `recommendedInStock` dataflow parameters, which will result in the re-execution of the dataflow operator, leading to the necessary updates to the user interface.

As we previously mentioned, the names identifying the dataflow dependency edges are – on the nodes to which these edges point – mapped to ambient behaviors (see chapter 6 section 6.3.1) subscribed to topics with the same name. This way, the reactive values used in the dataflow operator code are updated. Important to note is that the propagation of dataflow events happens by means of the underlying reliable asynchronous messages of AmbientTalk, on which the ambient behaviors abstraction is based. This means that intermittent connections between dataflow operators do not cause errors. Instead, the event messages are buffered and resent when the same dataflow operator host comes back in range or a replacement host is found. Per dataflow dependency, a timeout can be specified that determines how long these messages are buffered. In case of a timeout, an exception is raised on both disconnected application components. It should be caught by putting the role code in a standard AmbientTalk exception handler, which can trigger cleanup actions in response. In the case both components are connected via rebinding edges, a replacement node will (if available) be automatically bound to one of the roles.

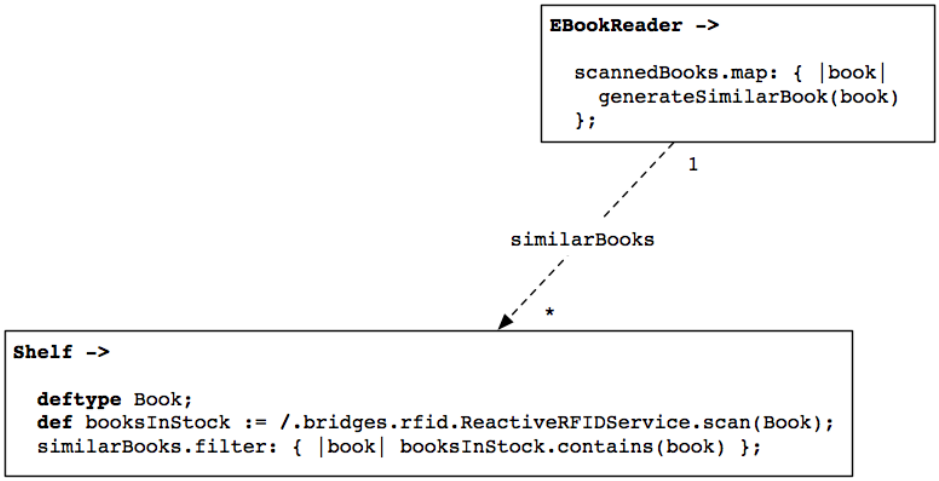

Figure 7.8: One-to-many dataflow dependecy between `EBookReader` node and `Shelf` nodes.

7.2.5 Dependency Arities

In many cases it is necessary to gather information from and/or propagate information to a multitude of peers (which all can move out of range and back into range at any point in time). For example, a mobile application may want to query other mobile applications about their GPS coordinates in order to show the location of those applications on a map. On the other hand, an application may also want to periodically broadcast the new GPS coordinates of its host device to nearby peers. To cater for this kind of arity-decoupled remote communication with some prescribed of equivalent peers, we have extended the original dataflow coordination model with *dependency arities*.

The mechanism is directly based on the distinction between ambient behaviors (see chapter 6 section 6.3.1) that subscribe to be updated with a single value and *group-centric* ambient behaviors (see chapter 6 section 6.3.2) that subscribe to be updated with sets of values. These dependency arities can be one-to-one, one-to-many, many-to-one or many-to-many. This is depicted graphically in our visual dataflow language on the end points of edges (i.e. `1---1`, `1---*`, `*---1` and `*---*`, respectively).

In the example given above and as highlighted in figure 7.8, there is a one-to-many dataflow dependency between the `EBookReader` and `Shelf` nodes. This will cause the dataflow engine not just to look for a single `Shelf` operator host (representing a shelf in our shop scenario), but to all hosts able to fulfill this role in the dataflow program.

They will all receive the mobile operator code and will all receive the events propagated along the `similarBooks` edge. Now it is up to the dataflow programmer to decide what will happen with all the different return values from the replicated `Shelf` nodes running in parallel.

One could either choose to receive the events propagated by a single `Shelf` (although other ones can be running in parallel) by specifying a one-to-one dependency between the `Shelf` and the `BookRecommender` node, as shown in figure 7.9. In this case, the first shelf with which a successful connection can be established is selected. As long as a connection can be maintained with the shelf, it keeps on propagating events. But, since it is a rebinding edge, a new shelf can be selected in case of a disconnection

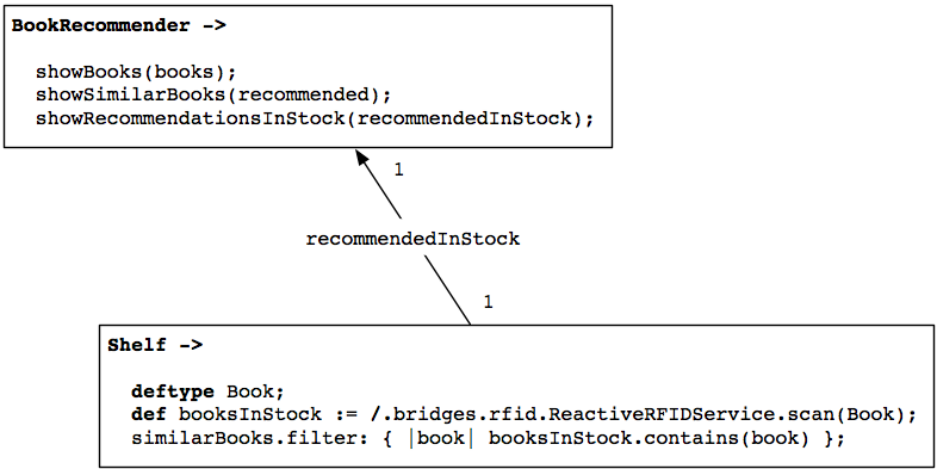

Figure 7.9: One-to-one dataflow dependency between `Shelf` node and `BookRecommender` node.

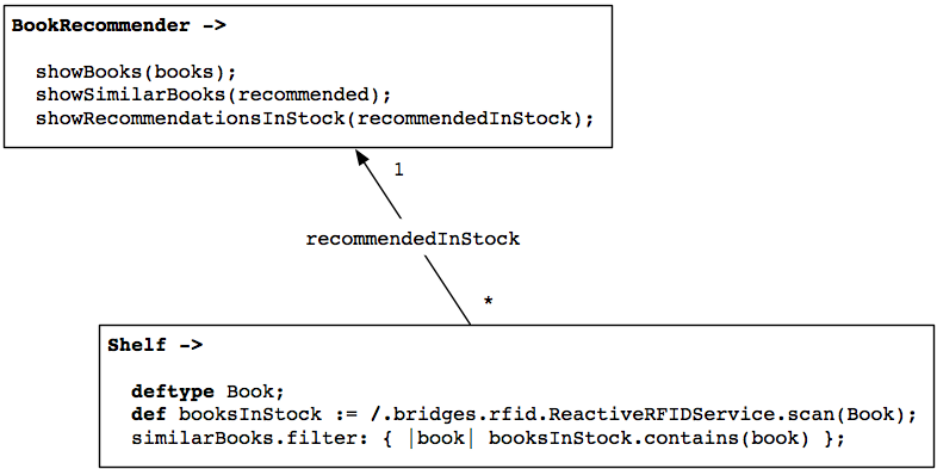

Figure 7.10: Many-to-one dataflow dependency between `Shelf` nodes and `BookRecommender` node.

to replace the disconnected shelf. In our scenario this means that the user only receives recommended products from a single shelf, although multiple ones are filtering similar book lists based on their contents.

The alternative would be to install a many-to-one dependency between these components, as shown in figure 7.10. In that case, the `BookRecommender` node will receive *all* the events of *all* the replicated `Shelf` nodes that are in communication range. The result here is that the user receives recommended products from all shelves in communication range, i.e. a reactive set of sets of books. The programmer can specify a timeout to determine how long values received from non-responding nodes should be kept in the reactive set. Hence, the reactive set changes not only when dataflow parameters change, but also when values are added (because new nodes were discovered) or removed (when nodes time out). The processing code that operates on this reactive set should of course take into account that the dataflow parameter represents such a changing set.

Declaring a dataflow dependency one-to-many or many-to-many will automatically convert the respective dataflow edge into a rebinding edge (see section 7.2.2). The reason is that, when broadcasting events to all nodes playing the same role, a fixed dataflow dependency simply makes no sense: events are propagated to a dynamically changing collection of subscribers as the network topology changes and as devices roam. This is not always desirable: the dataflow dependency between the `Shelf` node and `BookRecommender` node should clearly be fixed: one instance of the dataflow program should always send personalized recommendations to the same user.

Table 7.1 summarizes the semantics of the different combinations of dataflow nodes and dependency arities connecting them.

	Incoming `1`	**Incoming** `*`
Outgoing `1`	Signal one value to a single (rebound or fixed) node	Signal one value to all reachable nodes of same role
Outgoing `*`	Signal set of values to a single (rebound or fixed) node	Signal set of values to all reachable nodes of same role

Table 7.1: Dependency arity semantics.

As mentioned above, incoming edges that are one-to-many or many-to-many are mapped to a dataflow variable containing a reactive set of reactive values. The pace at which this reactive set is updated (and signals events) and hence how long these values remain in the set after their event source disconnected is determined by the timeout period associated with these incoming edges (see section 7.2.2). These reactive sets are nothing more than the group-centric ambient behaviors introduced in the previous chapter that need such a time period to apply time-sampling on asynchronously received values that arrive at irregular time intervals.

7.2.6 Stateful Reactions

In many cases, it is not enough to be able to react to a number of events, but in addition to generate, use and adapt state throughout different reactions as well (e.g. computing an average over previously received values). Up until now, we have assumed that dataflow operator code is re-executed in response to an event in a fresh scope. This prevents state from being used throughout different executions of the operator.

```
BookRecommender ->
  << previousBooks <<

  showBooks(books);
  showSimilarBooks(recommended);
  showRecommendationsInStock(recommendedInStock);

  def booksPutBack :=
    if: ((nil != previousBooks).and: { nil != books }) then: {
      previousBooks.filter: { |book| !books.contains(book); };
    } else: {
      [];
    };

  showBooksPutBack(booksPutBack);
  previousBooks := books;
```

Figure 7.11: `BookRecommender` role that additionally displays the books that are put back by the customer.

Consider a variant of the Book Recommender application where the books that the customer just put back (e.g. because they turned out not to be interesting enough to take home) are shown in the application GUI as well. This could be useful to display a history of consulted books. In figure 7.11 shown above, the `BookRecommender` role is extended to implement this behavior. AmbientTalk/R^V allows dedicated *state variables* to be specified to preserve state over subsequent executions of the dataflow operator. This list of variables is specified between the `<<` delimiters. These variables, `previousBooks` in this example, are kept in the scope of the dataflow operator, can be assigned, and remember their value over multiple executions of the operator. In the example, the `previousBooks` state variable is assigned at the end of the operator code, such that in the next execution the books that are put away by the customer can be computed by filtering out all books that the customer is currently carrying out of the `previousBooks` the customer was carrying[4].

Naturally, in the case of rebinding dataflow dependencies, that are represented by rebinding edges that allow the role to be rebound to another host in case the device hosting the dataflow operator times out, the state held by these state variables is not transferred to the new host. Hence, in most cases, maintaining state only makes sense in the case of fixed dataflow dependecies.

7.2.7 Evaluation

In this section, we have presented AmbientTalk/R^V, a visual dataflow language that reifies the coarse-grained control flow of a mobile event-driven application and unifies it with the data flow that steers the application. It is specifically geared towards mobile RFID-enabled applications in the following ways:

Dataflow coordination model based on AmbientTalk/R's object-oriented reactive programming model allowing to react to events **without resorting to explicit call-**

[4]The code added to the implementation of the `BookRecommender` role could in case of frequent usage be abstracted away in the public interface of the host. The language leaves it up to the programmer on which level such behavior should be implemented.

backs that invert the control flow of the application.

Fine-grained reactive node-centric code operating on time-varying dataflow parameters that is dynamically sent to remote services that host the necessary interfaces to implement the desired behavior. Dataflow parameters can contain any AmbientTalk/R reactive value. Hence, the language supports a **rich representation of events**.

Coarse-grained network-centric dataflow view on the entire program where dependencies between remote components are visualized and can be adapted without changing the fine-grained node-level code (if semantically correct).

Dataflow dependency arities allowing to subscribe dataflow operators to either a single event producer or a group of event producers publishing events under the same topic, or to publish events to only a single consumer or a group of consumers subscribed to a specific topic. Hence, the language supports **arity decoupling**.

Fixed edges and rebinding edges to discriminate in the visual node-centric program between dataflow operators that are allowed to be dynamically bound to replacement nodes, or that have to interact with always the same node within one instance of the dataflow program.

Support for stateful dataflow operators to allow state to be used over different executions of a dataflow operator (hosted on the same physical node).

Fault-tolerant, asynchronous event messaging by relying on the distributed event-loop concurrency model of AmbientTalk/R. Hence, the language supports **time decoupling**.

Decentralized, broadcast-based publish/subscribe of dataflow events by associating dataflow edge names with publications and subscriptions of the underlying node-centric dataflow and publish/subscribe system based on ambient behaviors (see previous chapter) and by associating dataflow role names with correspondingly tagged AmbientTalk/R services. Hence, the language supports **space decoupling**.

Through its unified data flow and control flow that have to be explicitly and visually specified for network-centric dataflow programs, it fulfills our remaining requirement: **global and automatic control flow management**.

7.3 A Variation: Producing Multiple Results

The dataflow coordination model that we have employed so far assumes that dataflow operators behave as regular reactive procedures and return a single value that is propagated to all dependent dataflow operators. This model is very close to the reactive programming model employed by the underlying reactive AmbientTalk/R interpreter. In this section, we introduce a variation on our visual dataflow language that we implemented. Its coordination model is moved more towards a publish/subscribe interaction style by allowing dataflow operators to signal multiple output events instead of returning a single reactive value.

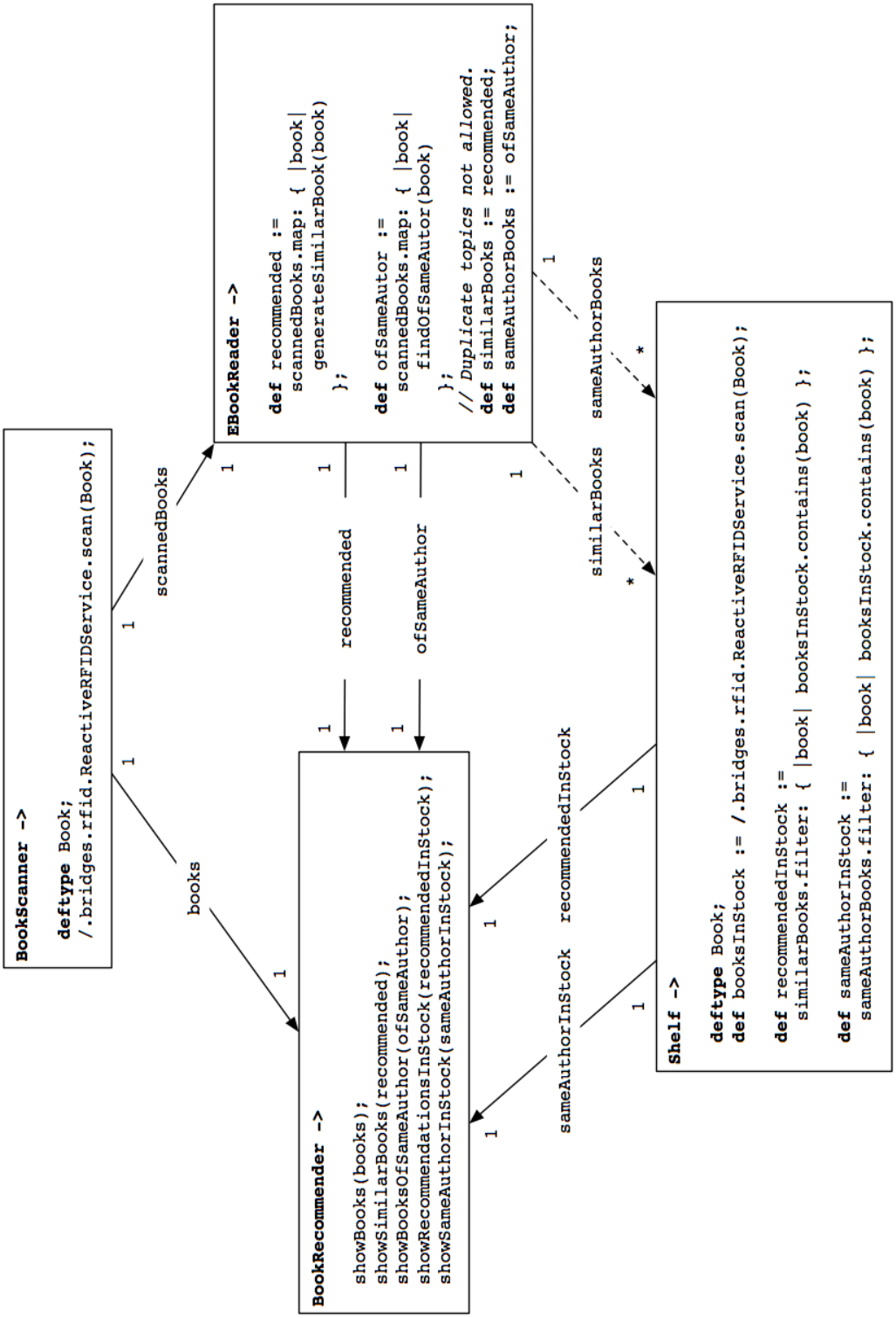

Figure 7.12: Visual dataflow implementation of the RFID-enabled Book Recommender application where multiple outgoing events are used to display books of the same author in the GUI as well.

Consider the following adaptation to the Book Recommender example application employed this far. The `EBookReader` role previously generating similar books from an existing catalogue, could as well generate a list of books by the same author. However, using the coordination model described up until now, the `EBookReader` role is only able to return a single result after reacting to an incoming event. This requires the programmer to either signal the list of similar books and the list of books by the same author manually wrapped in a pair, or either to split up the role in two roles, cluttering the dataflow graph with more roles that only marginally differ from one another.

A better solution would be to allow the `EBookReader` role to signal both generated lists of books and have them propagate over their dedicated set of dataflow edges. This requires two changes to the dataflow coordination model. First, dataflow operators should be able to signal more than one event after executing. Second, it should be able to associate multiple outgoing dataflow edges with the correct events that are signaled by the operator.

In this variant of AmbientTalk/R^V, we adopted the following strategy. Remember that dataflow operators are allowed to define new variables in their own dedicated scope. Now however, outgoing edges *with the same name* as these variables are used to propagate events when these variables change. This happens by, after each execution of the dataflow operator, looking up in the scope of the operator all variables that have the same name as outgoing edges and – if they have changed – propagate their new values over the edges. The drawback is that the dataflow programmer must take care of correctly defining these variables in the operator's scope according to edge names. The tricky part is that multiple edges with the same name are not allowed since they are represented as publish/subscribe topics, which must be distinct if they represent different events. The rest of the coordination model remains the same as previously explained.

For an illustrative example, figure 7.12 shows the adapted implementation of the Book Recommender dataflow application making use of different outgoing events and dataflow edges.

The `EBookReader` role now generates, besides a set of similar books, a set of books written by the same author. The `BookRecommender` role is made dependent to both reactive sets by the `recommended` and `ofSameAuthor` dependency edges that map to the variables with the same name defined in the `EBookReader` role. Subsequently in the same role, to prevent name clashes in the underlying topic-based publish/subscribe system, two variables holding the same values are defined: `similarBooks` and `sameAuthorBooks`. When they change, their new value is broadcast to all `Shelf` nodes via the edges with the same name. In response, the `Shelf` nodes filter out the set of books out of these two collections that are present in the shelf and propagate the resulting filtered time-varying collections over the `recommendedInStock` and `sameAuthorInStock` edges to the single `BookRecommender` node interacting with the current instance of the dataflow program.

Although this adapted coordination model makes things slightly more complicated, in many cases its additional flexibility allows to introduce more edges in the graph to reduce the amount of nodes, as shown above. Of course, it relies on the looser coupling of application components that interact using publish/subscribe instead of for instance direct messaging.

7.4 An AmbientTalk/R^V Programming Environment

The concrete implementation of AmbientTalk/R^V presented in this chapter comes with a visual programming environment that allows to specify and edit the dataflow graphs – although the focus of the implementation was the language runtime and not the visual tool. We give a brief overview of our prototype. It supports the basic operations required to specify, edit, save, load and run AmbientTalk/R^V programs and it notifies the user about errors in the dataflow program that can be easily statically verified. It does not include any further debugging support. Similarly, it does not offer mechanisms for reducing screen clutter such as grouping nodes or representing them as icons. This is an interesting topic of future work. Other than for editing purposes, the programming environment serves as a deployment tool for AmbientTalk/R^V programs.

7.4.1 Basic Operations

Figure 7.13 shows the AmbientTalk/R^V implementation of the original version of the RFID-enabled Book Recommender application first shown in section 7.2.1. The colors of the nodes are only for illustrative purposes and have no special meaning. Figure 7.14 gives an overview of the basic operations that can be found in the menu of the prototype tool. The tool allows the user to draw the dataflow graph by adding nodes representing roles, editing their source code in the tool itself, and wiring them together by means of dependency edges. The names and arities of dependecy edges can be edited directly in the tool as well. If error checking is enabled, the tool will notify the user about duplicate dependency names or role names and of undefined dataflow parameters in the role code (e.g. when a node has no corresponding incoming dataflow edge). Each edge has an associated timeout value that indictates how long event messages remain buffered before an exception is raised (0 means infinite).

Deploying the dataflow graph means that the AmbientTalk/R^V deployment tool will generate node-centric AmbientTalk/R code to look up nodes in the network matching the roles in the dataflow program and attempt to deploy the program by contacting them, sending them the role implementations and install the necessary node-level subscriptions and publications on these nodes. It is only during this step that individual nodes must be connected with the node that invokes the program (unless nodes are hosted on the same device of course).

7.4.2 Persistence and Importing Partial Graphs

Up until now we have not elaborated on how AmbientTalk/R^V interprets dataflow graphs. The visual dataflow programming environment presented above saves dataflow graphs edited in the tool as XML files containing a textual representation of the objects constituting the complete graph. The object graph that represents the dataflow graph can easily be reconstructed out of such an XML file to be subsequently used by the tool. Subsequently, the programming environment can display it and allows it to be edited by the user, or to deploy the dataflow graph (see section above). It can be seen as the graphical equivalent of an abstract syntax tree of a textual program.

We have not focused on a suitable dedicated file representation that allows editing dataflow graphs over different versions of AmbientTalk/R^V: for now, the XML files generated by our prototype tool must be manually updated if for example a new concept is added to the language.

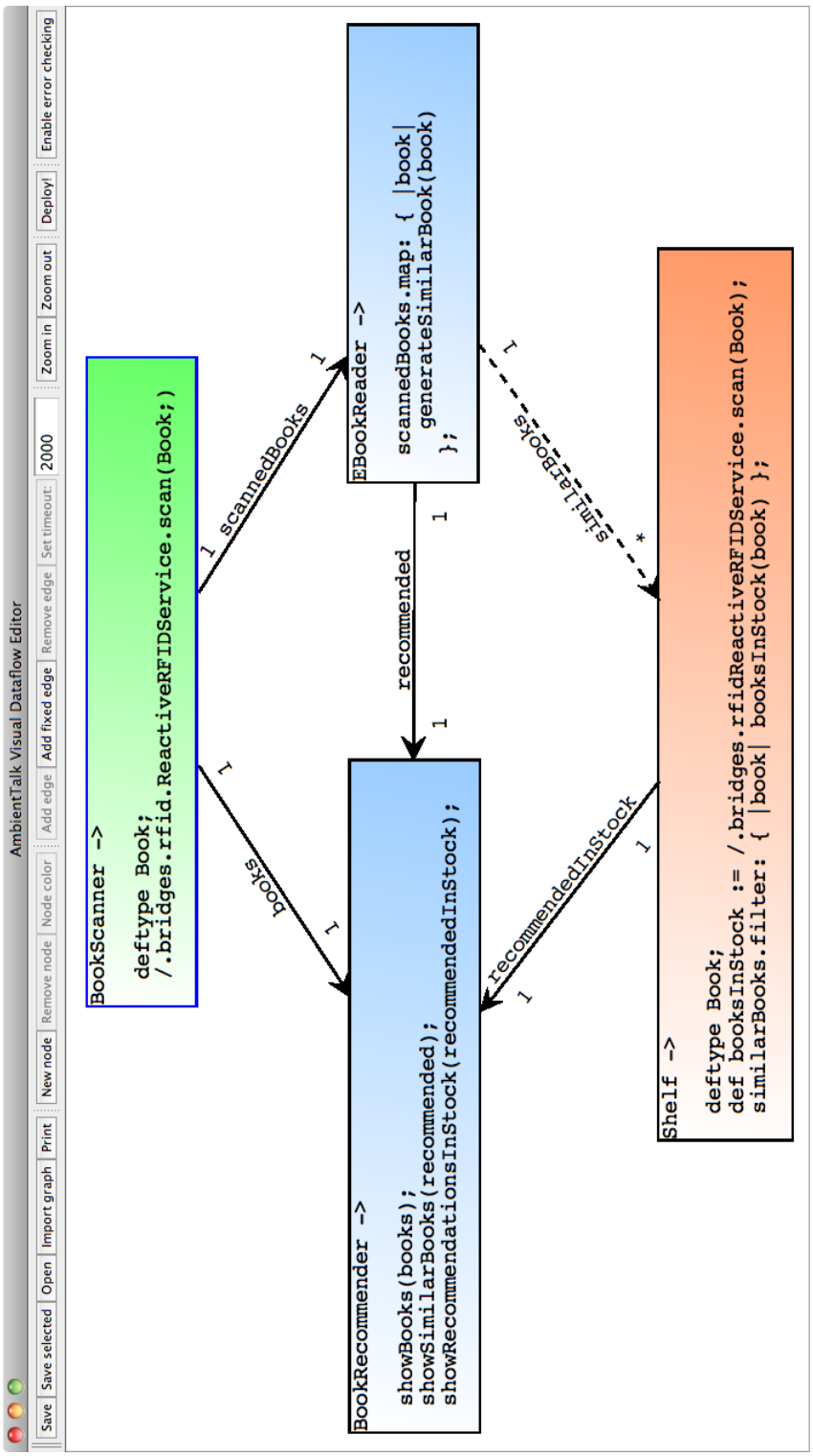

Figure 7.13: The implementation of the RFID-enabled Book Recommender application in the AmbientTalk/R^V prototype programming environment.

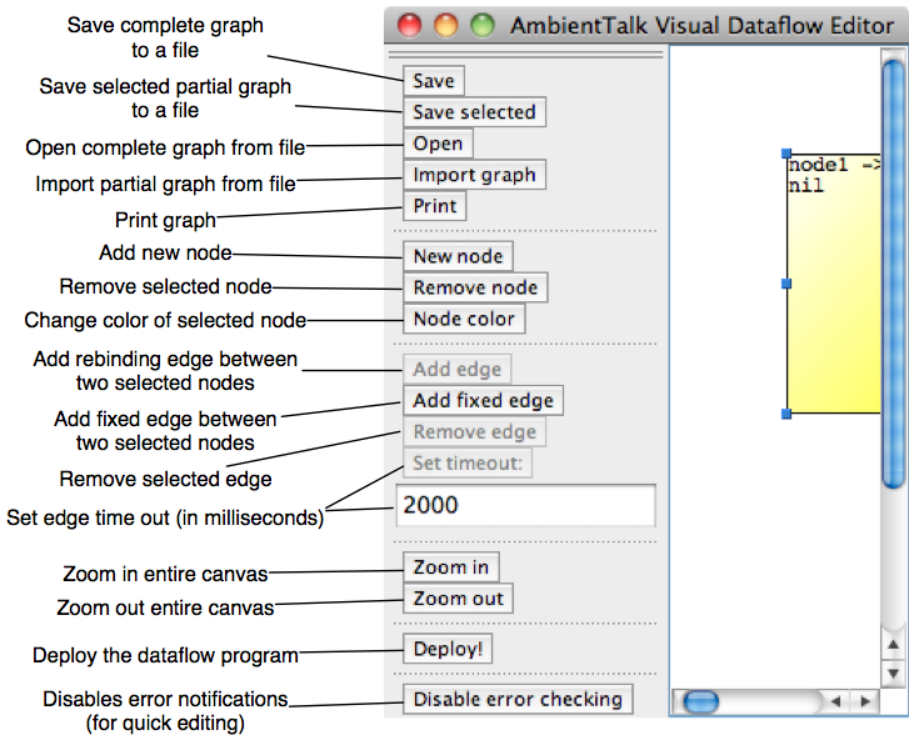

Figure 7.14: Basic operations supported by the AmbientTalk/R^V prototype programming environment.

Instead of saving the entire dataflow program, the programming environment also allows to save only the selected nodes of the graph and the edges that interconnect them. This partial graph can subsequently be imported in another dataflow program and wired to the nodes in that graph. We envision the dataflow programmer to create a catalog of frequently used partial graphs and import and customize them where necessary.

7.5 Implementation

In this section, we elaborate on the implementation of AmbientTalk/R^V.

7.5.1 Representing Distributed Dataflow Graphs

Figure 7.15 gives an overview of the implementation using an informal object diagram[5] of the AmbientTalk/R objects making up the implementation (except `VisualDataflowEditor`, which is the Java class implementing the graphical programming environment discussed above). We do not intend to explain the full implementation since a large part of the implementation consists of – from a scientific point of view – uninteresting graph manipulation and user interface code. Instead, we will just highlight the more interesting methods implemented by these objects. Similarly, we only explain the key changes to implement the variant of the language discussed earlier in section 7.3, since the rest of its implementation consists of some minor modifications that crosscut the entire implementation.

First, we explain the `startLookingForOperatorHost` method implemented by `VisualDataflowNode`, which is immediately called after deploying the dataflow program that contains the `VisualDataflowNode`. It is hosted by the device deploying the dataflow graph and is responsible for detecting dataflow operator hosts that match the role represented by the node. Its implementation is shown below.

```
1  def startLookingForOperatorHost() {
2    if: (isBroadcastNode()) then: {
3      startLookingForMultipleOperatorHosts();
4    } else: {
5      startLookingForSingleOperatorHost();
6    };
7  };
```

It first checks if the when the dataflow operator is executed its return value should be broadcast to all reachable dependent nodes, or propagated to a single dependent node. In the second case, it delegates to `startLookingForSingleOperatorHost`, of which the implementation is shown below.

```
1  def startLookingForSingleOperatorHost() {
2    subscription_ :=
3      when: operatorHostType_ discovered: { |host|
4        currentHosts_ := [ host ];
5        def fromOneEdges := incomingEdges_.filter: { |edge|
6          !(edge.isAccumulatorEdge())
7        };
8        def fromManyEdges := incomingEdges_.filter: { |edge|
9          edge.isAccumulatorEdge()
10       };
11
```

[5] AmbientTalk/R is a classless language. In this informal diagram, we represented prototypical objects as UML classes.

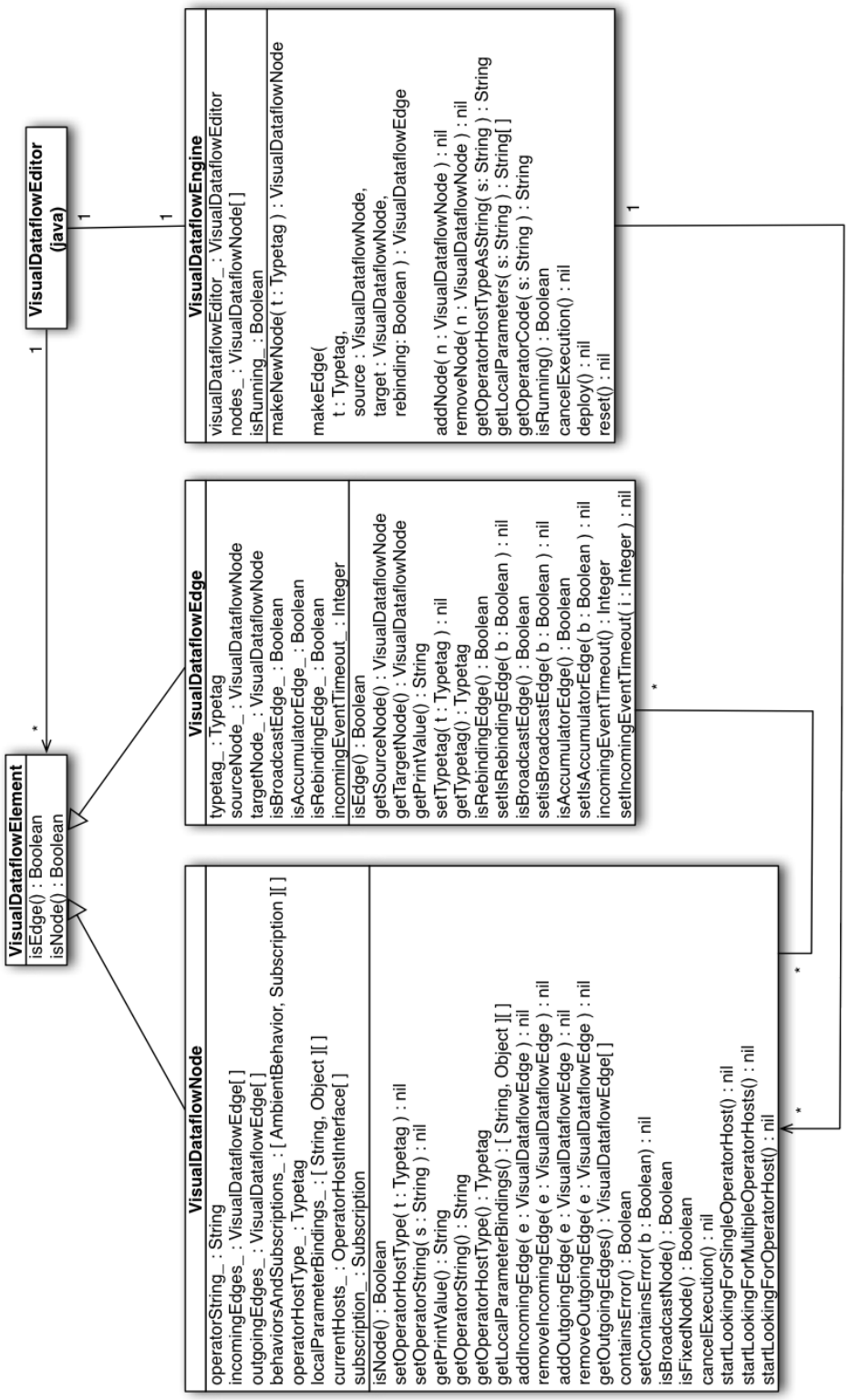

Figure 7.15: Object diagram of the implementation of AmbientTalk/R^V (a).

```
12        host<-execute(
13          operatorString_,
14          fromOneEdges.map: { |edge|
15            [ edge.getTypetag(), edge.incomingEventTimeout() ] },
16          fromManyEdges.map: { |edge|
17            [ edge.getTypetag(), edge.incomingEventTimeout() ]
18          },
19          outgoingEdges_.map: { |edge| edge.getTypetag() },
20          localParameterBindings_);
21
22        if: (!isFixedNode()) then: {
23          when: host disconnected: {
24            if: (nil != subscription_) then: {
25              subscription_.cancel();
26              currentHosts_ := [];
27            };
28            startLookingForOperatorHost();
29          };
30        };
31      };
32  };
```

The method creates a subscription that listens one single time (cf. `when:discovered:` on line 3) for remote objects (implementing `OperatorHostInterface`) in the network that are tagged with a type tag matching the role type `operatorHostType_`. When a reference to such an object representing a role host is obtained, the `currentHosts` list is assigned a list that contains only this single host. Subsequently, the incoming edges are classified into edges that have to listen for events from a single node on which this node depends (`fromOneEdges`, incoming one-to-one and one-to-many edges in the dataflow graph) or into edges that have to listen for all events from every node fulfilling the same role that is discovered (`fromManyEdges`, incoming many-to-one and many-to-many edges in the dataflow graph).

Next, the host is sent the asynchronous `execute` message that will, if the message is successfully sent and processed, execute the role on that specific host. The `execute` message takes the role source code as a string (which it can evaluate), both types of incoming edges together with their typetag (representing their name in the graph), all outgoing edges together with their typetag (idem), and the local parameter bindings which are used to remember state variables over different executions of the same role. Note that sending the execute message does not necessarily mean that the host will fulfill a role in the dataflow program. Using the default implementation of `OperatorHostInterface`, it will simply cause it to subscribe the necessary ambient behaviors, which can potentially cause these behaviors to be bound to a number of publications that represent edges in the graph.

Finally, it is checked if the node has one or more fixed dataflow edges (incoming or outgoing). If this is the case, the host should always remain the same during the entire lifetime of the dataflow program. If this is not the case, a new `when:disconnected:` observer is installed that, when the host is disconnected from the node, cancels the subscription and reinvokes the method to repeat the process with (potentially) a new host.

The implementation of `startLookingForMultipleOperatorHosts` is very similar and is shown below:

```
1   def startLookingForMultipleOperatorHosts() {
2     subscription_ :=
3       whenever: operatorHostType_ discovered: { |host|
```

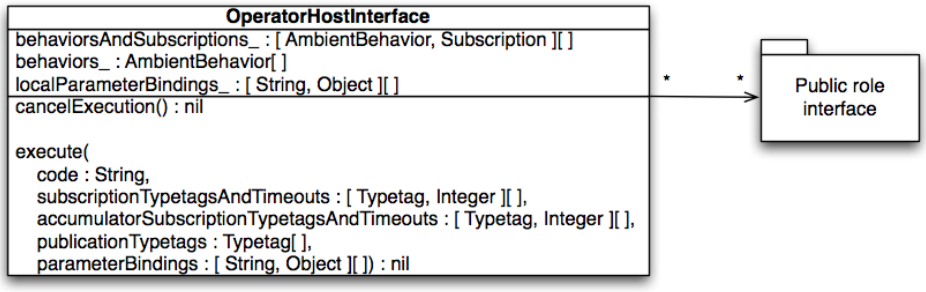

Figure 7.16: Object diagram of the implementation of AmbientTalk/R^V (b).

```
4         currentHosts_ := currentHosts_ + [ host ];
5         def fromOneEdges := incomingEdges_.filter: { |edge|
6           !(edge.isAccumulatorEdge())
7         };
8         def fromManyEdges := incomingEdges_.filter: { |edge|
9           edge.isAccumulatorEdge()
10        };
11
12        host<-execute(
13          operatorString_,
14          fromOneEdges.map: { |edge|
15            [ edge.getTypetag(), edge.incomingEventTimeout() ]
16          },
17          fromManyEdges.map: { |edge|
18            [ edge.getTypetag(), edge.incomingEventTimeout() ]
19          },
20          outgoingEdges_.map: { |edge| edge.getTypetag() },
21          localParameterBindings_);
22
23        if: (!isFixedNode()) then: {
24          when: host disconnected: {
25            currentHosts_ :=
26              currentHosts_.filter: { |aHost| aHost != host };
27          };
28        };
29      };
30  };
```

The only difference is that in this case multiple hosts can be discovered because of the `whenever:discovered:` AmbientTalk/R observer registered on line 3 instead of the `when:discovered:` observer in `startLookingForSingleOperatorHost`. The rest of the code is slightly adapted to this fact, but for the sake of brevity we omit these changes.

7.5.2 The Default Host Interface

The last important part that remains to be explained is the default `OperatorHostInterface` object representing a node that will execute one or more roles in the dataflow program. It implements the infrastructural layer that enables a mobile node to execute AmbientTalk/R^V roles and interact with other dataflow nodes. Such an object is hosted on this physical node and has access to a package that the programmer of the node designated to be public code that can be called from within the role code. This is depicted in an informal object diagram in figure 7.16.

Not surprisingly, its main method is the `execute` method that is invoked by asynchronous `execute` messages sent by `VisualDataFlowNode`s when deploying a dataflow operator, as previously discussed. Its implementation is shown below.

```
1  def execute(
2    roleCode,
3    subscriptionTypetagsAndTimeouts,
4    accumulatorSubTypeTagsAndTimeouts,
5    publicationTypetags,
6    parameterBindings) {
7
8    // Subscribe node-centric ambient behaviors.
9    behaviorsAndSubscriptions_ :=
10     subscriptionTypetagsAndTimeouts.map: { |pair|
11       ambientBehavior: pair[1] @Any(pair[2]);
12     };
13
14   // Subscribe group-centric ambient behaviors.
15   behaviorsAndSubscriptions_ := behaviorsAndSubscriptions_ +
16     accumulatorSubTypeTagsAndTimeouts.map: { |pair|
17       ambientBehavior: pair[1] @All(pair[2]);
18     };
19
20   behaviors_ := behaviorsAndSubscriptions_.map: { |pair| pair[1]; };
21   localParameterBindings_ := parameterBindings;
22
23   // Lifted function taking ambient behaviors as argument.
24   def liftedFunction(@args) {
25     def env := isolate: { };
26     def envMirror := (reflect: env);
27     def index := 1;
28
29     // Define local parameters in role scope.
30     while: { index <= localParameterBindings_.length() } do: {
31       def fieldName :=
32         eval: (read: ("`" + localParameterBindings_[index][1]))
33         in: env;
34       envMirror.defineField(
35         fieldName,
36         localParameterBindings_[index][2]);
37       index := index + 1;
38     };
39     index := 1;
40
41     // Define dataflow variables in role scope.
42     def allTypetags :=
43       (subscriptionTypetagsAndTimeouts +
44       accumulatorSubTypeTagsAndTimeouts).map: { |pair| pair[1] };
45     while: { index <= (allTypetags.length() } do: {
46       def fieldName := (allTypetags[index]).typeName();
47       envMirror.defineField(fieldName, args[index]);
48       index := index + 1;
49     };
50
51     // Evaluate role  code in role scope and store returned behavior.
52     def returnBehavior := eval: (read: roleCode) in: env;
53
54     // Store new value of local parameters for next execution.
55     index := 1;
56     while: { index <= localParameterBindings_.length() } do: {
57       def value :=
58         envMirror.invokeField(
```

```
59            env,
60            eval: (read: ("`" + localParameterBindings_[index][1]))
61              in: env);
62        (localParameterBindings_[index])[2] := value;
63        index := index + 1;
64      };
65
66      // Return result behavior.
67      returnBehavior;
68    };
69
70    // Generate result behavior using lifted function.
71    def resultBehavior := liftedFunction(@behaviors_);
72    // Publish resulting behavior using outgoing edge type tags.
73    publicationTypetags.each: { |typetag|
74      exportBehavior: resultBehavior as: typetag;
75    };
76  };
```

It first subscribes the necessary node-centric and group-centric ambient behaviors to support listening to events coming from incoming edges. The `ambientBehavior:` construct returns a pair: the behavior itself and a subscription object that can be used to cancel the subscription. All pairs are stored in a list, from which the ambient behaviors are extracted into the `behaviors_` field of the `OperatorHostInterface` object. It also modifies the `localParameterbindings_` field to denote the state variables encoded in the current role being executed.

Subsequently, a special function `liftedFunction` is defined that takes an arbitrary number of arguments by means of the `@args` AmbientTalk construct, which causes the `args` formal parameter to be bound to the list of arguments that is passed. The idea is to pass a list of subscribed ambient behaviors to this function and rely on the dataflow evaluation strategy of the AmbientTalk/R interpreter to drive the reactive execution of the role. `liftedFunction` first creates an empty object without lexical scope by means of the `isolate:` AmbientTalk/R construct. This isolate's object scope will be used as an environment to evaluate the role code in. This means that the role code by default has no access to any variable outside this object's scope, unless the node programmer assigns specific packages to contain public code.

The next step (line 30) is to iterate over the state variable bindings from the previous execution of the role and – using AmbientTalk/R's reflection and metaprogramming facilities – redefine them into this fresh isolate object, such that they can be used in the current execution of the role code. Similarly, on lines 42 to 49, all type tags denoting incoming edges are used to define dataflow variables in that same scope. They are bound to the ambient behaviors in the `args` list that were previously subscribed.

From this point on, the role's scope contains the necessary definitions to execute the role's code. On line 52, a new behavior is generated by evaluating the role's code in the role's scope. Since the dataflow variables that are defined in the role's scope are used in the role's code, the resulting value of this evaluation is a new time-varying value or behavior.

Now the only thing that remains to be done is fetch the new values of the (potentially changed) state variables out of the role's scope and store them in the `localParameterbindings_` field of the host object to have them ready for the next execution of the role (lines 56 - 64). Subsequently, the behavior resulting from executing our special lifted function is returned, which is the return value of executing the role code.

On line 71, this behavior is generated by calling the lifted function on the ambient behaviors that were subscribed earlier and that correspond to the incoming datadlow edges. Finally, on line 73, this behavior is published using all type tags associated with outgoing edges, such that other hosts fulfilling their own set of roles and subscribing their own set of ambient behaviors can subscribe to them.

A Sketch of An Alternative Evaluation Strategy Implementation

To sketch how the evaluation strategy is changed for the second flavor of AmbientTalk/R^V that we implemented and that is discussed earlier in section 7.3, consider the following piece of code taken out of the `liftedFunction` of this second implementation.

```
1  def generateTypetagsAndBehaviors(beh) {
2    publicationTypetags.map: { |typetag|
3      def typeName := typetag.typeName();
4      if: (envMirror.respondsTo(typeName)) then: {
5        [ typetag, envMirror.grabField(typetag.typeName()).readField() ];
6      } else: {
7        [ typetag, nil ];
8      };
9    };
10 };
11
12 generateTypetagsAndBehaviors(returnBehavior);
```

The above procedure takes the resulting `returnBehavior` of the previously discussed version of `liftedFunction`, but instead of immediately returning it, it will return a list of behaviors associated with their publication type tags (i.e. outgoing edges names) that are all looked up in the scope of the role (hence, the dataflow programmer must have defined them in the role code).

After executing the now adapted special lifted function, all these behaviors are published with their associated type tag found on the corresponding outgoing edges, as shown below.

```
1  def typetagsAndBehaviors := liftedFunction(@behaviors_);
2  typetagsAndBehaviors.each: { |pair|
3    exportBehavior: pair[2] as: pair[1];
4  };
```

We omit the changes to the rest of the implementation as they crosscut large parts of the implementation of `VisualDataflowNode` and `VisualDataflowEdge`, but do not significantly change the fundamental ideas.

7.6 Limitations

In this section, we discuss the limitations of AmbientTalk/R^V and its variant presented in this chapter. First of all, since the implementation of the language is directly based on the ambient behaviors constructs introduced in the previous chapters, the limitations (see chapter 6 section 6.6) that apply to our approach to node-centric ambient-oriented dataflow programming also apply to the network-centric approach introduced in this chapter. Additional limitations are categorized below.

7.6.1 Subscription

An obvious problem is that dataflow edge names and role names have to be globally unique in a network-centric dataflow program. A solution to this problem could be to automatically generate unique identifiers acting as topics. This way, they can be automatically mapped by the the language to the programmer-supplied and non-unique topics. The language must then maintain a mapping from unique topic-edge and role-operator pairs to such generated unique identifiers.

Another way of alleviating this problem could be to rely on the content-based subscriptions of the node-centric approach, which we did not incorporate in AmbientTalk/R^V. AmbientTalk/R^V currently only supports topic-based publish/subscribe based on role and edge names. We will investigate in future work how the language could be (syntactically) extended to support content-based subscriptions.

7.6.2 Deployment

A limitation of the current implementation is that new devices – or replacement or additional devices in the case of rebinding dataflow dependency edges – always first have to be contacted by the device that invoked the dataflow program before fulfilling their role in a concrete instance of the dataflow program. The reason is that the invoking node hosts all the role scripts and must send these scripts to appearing remote nodes that can execute one of the roles. Once this has happened though, the execution of the dataflow program happens entirely decentralized by having the correct events published and the correct ambient behaviors subscribed. An avenue for future work might be to further decentralize this process by allowing nodes that are looking for other nodes fulfilling a certain role in the dataflow program to dynamically send them the role code once they are discovered.

7.6.3 Visual Scalability

Other limitations that we observe have to do with our visual dataflow programming environment that is an early prototype. Some features that we believe are missing to make it truly usable for larger scale projects are collapsing of dataflow operators (maybe using an icon instead), nesting of dataflow operators (such that dataflow programs can be built hierarchically) and debugging support. The second issue requires a revision of dataflow edge semantics to have meaningful semantics while nesting.

7.6.4 Tooling

Finally, although our textual representation of AmbientTalk/R^V is perfectly executable, we feel that there is room for improvement in this regard, such that the textual representations become smaller and more robust to changes in the tool (and maybe even language). Instead of using an XML-represented object dump of the Java objects representing the dataflow graph, we might consider using a more efficient and suitable representation.

7.7 Conclusion

In this chapter we have introduced the network-centric visual ambient-oriented dataflow language AmbientTalk/R^V specifically geared towards coordinating mobile RFID-enabled

applications. The motivation for using a dataflow language for coordination is that the language offers a coordination model that is very well suited to the dynamic and inherently parallel nature of mobile RFID-enabled applications and allows separating the coarse-grained coordination behavior from the fine-grained application logic. Thanks to its coarse-grained control flow that is unified with its data flow and visually representable, our final remaining requirement is satisfied: global and automatic control flow.

AmbientTalk/R^V represents data dependencies between distributed mobile application components very explicitly and allows them to be visualized and edited graphically, offering a network-centric view on the control flow of the entire application. This network-centric view allows to change the control flow of an application without invasive changes to the node-level code. Since its coordination model is purely based on the satisfaction of these data dependencies, it maps very well on a mobile ad hoc network environment where distributed application components are running in parallel, react to events coming from the outside world, and are interconnected by peer-to-peer connections over which data can only flow when the connection is not broken (which may frequently happen due to the limited communication range and the mobility of the devices).

To allow the different components in the dataflow program to be both decoupled in space, time and arity, it makes use of an underlying broadcast-based publish/subscribe architecture. Execution of dataflow programs happens entirely decentralized by dynamically binding nodes to dataflow operators, which operate on time-varying dataflow variables, which are updated by the underlying publish/subscribe system. This dynamic binding leverages the group-centric event processing mechanisms of the underlying publish/subscribe architecture and the language offers specific syntax and a clear semantics for both fixed, rebinding and dataflow dependencies of different arities.

Both the loose coupling of dataflow nodes and the fault-tolerant delivery of events based on AmbientTalk's asynchronous messages render such dataflow programs resilient to the volatile connections inherent to many wireless communication infrastructures. Events can be any AmbientTalk data type or serializable Java object.

Chapter 8

Conclusion

In this concluding chapter, we revisit our research goals as stated in the introduction with hindsight and we highlight the contributions of this dissertation once more. We discuss problems that remain open and where our proposal could be improved. This leads us to formulate a number of avenues for future research that we deem important in the fields of mobile RFID-enabled applications, ambient-oriented and dataflow programming.

8.1 Research Goals

In section 1.3, we listed a number of research goals that we intended to achieve with our work. We repeat them here and discuss to what extent they have been achieved.

- It was our goal to unify RFID-tagged physical objects with software objects in an object-oriented language while keeping into account the hardware properties of RFID technology that have a strong impact on mobile RFID-enabled applications. Part of the goal was to come up with a number of criteria that specify how this integration should happen such that the programmer can on one hand abstract over the peculiarities of RFID hardware, but on the other hand react on interesting events signaled by RFID devices.
- Given such mobile applications driven by external events, it was our goal to provide an alternative to reacting to these events via callbacks or event handlers, to circumvent a number of software engineering issues such as inversion of control. To this end, a solution would be to integrate two paradigms in event-based programming – dataflow programming and publish/subscribe – to eliminate callbacks and to retain the loose coupling of distributed applications components required for mobile RFID-enabled applications.
- It was our goal to unify the control flow and data flow of mobile RFID-enabled applications and make them explicit on a network-centric level. Such an explicit distributed control flow is expressed in a network-centric coordination language. To integrate with our dataflow execution model, such a network-centric language should rely on a dataflow execution model as well, but offer a network-centric view on the entire application. This coordination language should be applicable to mobile RFID-enabled and ambient-oriented applications and should incorporate the programming model requirements for these applications.

We have adapted an ambient-oriented programming language to allow the implementation of mobile RFID-enabled applications by fine-tuning the built-in abstractions of the language to support RFID technology. Additionally, we have extended the language to allow the development of these applications without suffering from inversion of control by integrating distributed reactive dataflow programming constructs with the event-driven communication mechanisms based on asynchronous messages and publish/subscribe of the language. On top of these distributed dataflow constructs, we have implemented a network-centric visual coordination language. We have shown the feasibility of our approach by a proof by construction. Together, these resulting artifacts form the main contributions of this dissertation:

AmbientTalk/2 was extended for mobile RFID-enabled applications because the ambient-oriented programming paradigm on which it is based already offers some support for open, networked applications interacting over volatile connections. We extended this language to represent RFID-tagged physical objects as true mutable remote software objects called things. The event-driven nature of the language based on communicating event loops allows the implementation to be cleanly integrated in the language, but suffers from typical issues arising in more complex event-driven programs structured around event handlers or callbacks, such as inversion of control.

Node-centric ambient-oriented dataflow programming constructs were added to AmbientTalk/R, a version of the language supporting a variant of dataflow programming called reactive programming. These language constructs allow mobile RFID-enabled applications to interact over volatile connections by means of a loosely-coupled publish/subscribe mechanism while allowing these applications to react to external events using time-varying dataflow values. The explicit control flow of the textual language is to a large extent traded for the automatic tracking of dataflow dependencies between time-varying values by the runtime of the language.

AmbientTalk/R^V, a network-centric visual dataflow coordination language, was developed to offer a network-centric, visual representation of the implementation of mobile RFID-enabled applications to render the data flow explicit and easily adaptable again. The implementation is based on the node-centric dataflow programming constructs mentioned above, keeping mobile RFID-enabled applications loosely coupled. It additionally offers dedicated syntax and semantics for different dataflow dependency styles between distributed application components, such as rebinding dependencies versus fixed dependencies and dependencies of different arities.

Throughout this dissertation we have used a case study of a mobile RFID-enabled application for which we have laid the infrastructure using our RFID programming model in chapter 5. We have subsequently refined it by introducing our node-centric dataflow constructs in chapter 6 which turn the control flow implicit, and have finally made the data flow of the application explicit in chapter 7 using our network-centric dataflow coordination language. We have observed that the issues we are trying to tackle with implementing such applications using traditional approaches are alleviated. Additionally, we have benchmarked the artifacts that we have developed. We have observed that our abstractions induce a computational overhead, but not to such an extent that it hinders scalability or undermines the insights presented in this dissertation.

8.2 Restating the Contributions

We now summarize how each chapter contributes to this dissertation's research goals:

- In chapter 2 we introduced the concept of mobile RFID-enabled applications – which arise from combining mobile technology with RFID technology – and we discussed the software engineering problems inherent to such applications. These problems stem from the fact that these applications have to deal with dynamically changing and unreliable networks. After observing the similarities between RFID technology and mobile ad hoc networking technology, we chose to extend the ambient-oriented programming paradigm with novel event handling constructs to tackle the highly event-driven nature of mobile RFID-enabled applications. This leads us to, based on the ambient-oriented programming criteria, define a list of programming model requirements for mobile RFID-enabled applications.

- In chapter 3, we surveyed a number of technologies that, to various extents, satisfy some of these programming model requirements. These technologies form the related work of this dissertation. We concluded that mobile RFID-enabled applications should be implemented as decentralized event-driven architectures to achieve a loose coupling on the architectural and execution level. Unfortunately, structuring applications this way causes inversion of control. Dataflow programming techniques on the other hand, can be used for reacting on events without suffering from the problems caused by event handlers or callbacks that are uncoordinatedly executed in response.

- In chapter 4, we selected the ambient-oriented programming paradigm embodied by the programming language AmbientTalk as a starting point for implementing mobile RFID-enabled applications. Due to its decentralized and event-driven programming model, it already satisfies some of our requirements. We discussed where the paradigm offers partial support for implementing mobile RFID-enabled applications using modern distributed programming techniques.

- In the same chapter, we discussed where AmbientTalk falls short to deal with all programming issues in mobile RFID-enabled applications. Concretely, the event-driven programming model based on event handlers or callbacks to react to events causes a number of software engineering issues – such as inversion of control – which are discussed at the end of the chapter.

- In chapter 5, we mapped the object-oriented programming model and event loop concurrency model of AmbientTalk to mobile RFID-enabled applications. We provided an overview of a prototype implementation. We successfully applied the discerning principles of the ambient-oriented programming model to mobile RFID-enabled applications. We observed that mobile RFID-enabled applications suffer from the same problems as other event-driven applications of which the implementation is based on callbacks, but to a larger extent.

- In chapter 6, we integrated a variant of dataflow programming (called reactive programming) supported by a variant of the AmbientTalk language called AmbientTalk/R with its underlying decentralized publish/subscribe architecture. This allows structuring mobile ad hoc networking applications as loosely-coupled event-driven distributed programs while supporting reactive programming as a

means to react to events. The constructs strongly rely on AmbientTalk's event loop concurrency model. We have called this style of dataflow programming ambient-oriented dataflow and provided a prototype implementation.

- In the same chapter, we applied these constructs first to a peer-to-peer ambient-oriented application, and subsequently to our mobile RFID-enabled application used as a case study. We observed that, by making the control flow of the application implicit through reactive programming techniques, the typical issues associated with event-driven programs structured around callbacks are solved. In the case of our example mobile RFID-enabled application, we additionally observed that, because of the node-centric view on subscriptions and publications among heterogenous peers, the data flow of the application remains very implicit and leads to a number of issues that call for a network-centric view on the application. In short: one of our requirements is not satisfied: global control flow management.

- In chapter 7, we introduced AmbientTalk/R^V: a visual network-centric coordination language that uses ambient-oriented dataflow as an explicit network-centric coordination mechanism. The resulting node-centric code during execution of such a dataflow program utilizes the node-centric dataflow primitives of the previous chapter, preserving a loosely-coupled and decentralized architecture. To represent different decoupled communication patterns, we introduced two novel concepts in this language: the distinction between fixed and rebinding dataflow edges and different semantics for edge arities. We provided a prototype implementation.

- In the same chapter, we restructured the mobile RFID-enabled application used before as a case study to obtain a network-centric view in AmbientTalk/R^V. We observed that the coarse-grained data flow can be easily modifed by manipulating the explicitly represented dataflow graph of the application by means of its dataflow dependencies represented as different types of dataflow edges. This way, our final requirement – global control flow management – is satisfied.

8.2.1 Fundamental Contributions

In short, our two fundamental contributions are:

We extended the ambient-oriented programming paradigm to support mobile RFID-enabled applications running atop of mobile RFID-enabled devices and using passive RFID tags. This way, the paradigm becomes applicable to ambient systems of which part of the participating entities have no computational power whatsoever. This allows to program mobile RFID-enabled applications in which RFID-tagged physical objects are effectively represented as remote software objects.

We integrated two event paradigms. The publish/subscribe paradigm offers space, time and arity decoupling and is therefore well-suited for implementing distributed systems in general and mobile ad hoc networking applications in particular. Unfortunately, current publish/subscribe systems cause inversion of control. Dataflow programming on the other hand does not cause inversion of control. Current dataflow programming technology however induces a tight coupling which renders it unsuitable for mobile ad hoc networking applications. In this dissertation, we integrated both event paradigms and showed that the result can be used to program mobile RFID-enabled applications in an ambient-oriented style without suffering from inversion of control.

8.3 Limitations of Our Approach

In the foregoing chapters we already discussed specific technical limitations of the approaches that they describe. In this section, we repeat the most important ones and position them in the broader context of this dissertation.

8.3.1 Security and Privacy of Mobile RFID-Enabled Applications

Mobile RFID-enabled applications in many cases have to broadcast information to all reachable peers because it is the only way of communicating, or because previously unknown peers have to be contacted or because it is simply the most appropriate communication abstraction. The resulting uncontrolled dissemination of information could pose a potential privacy risk. This is most apparent from current RFID hardware that offers no adequate support to prevent malicious parties to sniff data broadcasted over radio frequency waves. Similarly, AmbientTalk/R^V matches peers able to fulfill a certain role in the dataflow program simply on their role name and in response they will execute foreign code associated with the corresponding dataflow node in the dataflow program. Currently, there is no protection from malicious dataflow programs that send harmful code to other devices. In this dissertation we have ignored security and privacy and hence suggest that our approach should be complemented with additional infrastructure to be used in scenarios where security and privacy are concerns.

8.3.2 Distributed Glitch Prevention

In dataflow programming, glitches occur because of dependent code that is executed in the wrong order. This can easily happen in a distributed setting where events are communicated over the network and are hence delivered with a delay of which the magnitude depends on different factors, such as the underlying network technology, network congestion, network failures, etc.

To cater to the requirements of mobile RFID-enabled applications that we put forward in the beginning of this dissertation, we opted for an entirely decentralized approach, which precludes us from relying on a reliable, centralized node that is contacted by every single participant in a distributed interaction to take care of event ordering. Hence, in our prototype implementation, we have not taken into account event ordering across dataflow dependencies over multiple nodes (between just two nodes, ordering is

guaranteed). This means that our implementation is free of glitches on a single node, but not free of distributed glitches across multiple nodes.

8.3.3 Overhead of Dataflow Programming

Naturally, tracking dataflow dependencies consumes time and memory resources. A dataflow graph of dependent computations which may refer to arbitrary objects must be maintained by the language runtime, making garbage collecting these objects not always possible [PD10] and introducing a memory overhead. Similarly, there is a computational overhead caused by checking whether values are reactive, if this is the case lifting dependent procedure calls or method invocations, and traversing the dataflow graph to update dependent expressions.

Our implementation is based on AmbientTalk/R, an AmbientTalk interpreter supporting reactive programming in an object-oriented setting. In functional reactive programming, extensive work has been done on optimizing reactive programs, but it is unclear if similar techniques are possible in a non-functional setting. For our implementation, we have not considered thorough optimizations and we have no exact figures on the overhead of our implementation when used for reactive applications compared to the same applications based on classic event-driven techniques.

In chapter 6, we did benchmark our implementation with respect to the amount of messages that are sent over the network compared to a manual approach. It turned out that our implementation requires about two times the number of messages to communicate the same number of events. Although this is not excessive, this can probably be improved by tuning the underlying publish/subscribe system (which we did not adapt) to our model.

8.3.4 Event Processing Bottlenecks

We rely on the event loop model of the AmbientTalk language to serialize asynchronously received events into an event queue such that they can be processed sequentially by a single thread. This makes our model free of low-level race conditions, but of course prevents events to be concurrently processed within a single event loop and makes parallelizing event processing a non-trivial exercise, potentially causing bottlenecks. This could form an issue in cases where a huge amount of events has to be processed within strict time constraints. A way to circumvent this problem could be to allow event consumers to specify how fast they can process certain events. The dataflow language constructs presented in this dissertation, however, offer no way for an event consumer to tell its event producer to limit the events that it wants to receive: a publication or subscription can only be cancelled. Afterwards, the subscription can be re-established.

8.4 Avenues for Future Research

In this section, we discuss how our research can be extended in directions that would address certain limitations or would allow this work to be used in different contexts.

8.4.1 Active RFID Technology

Active RFID tags differ from their passive counterparts by being equipped with an integrated power source (which is in most cases a battery). Because they are less con-

strained with regard to power consumption, they can be equipped with more advanced circuits, such as for example simple 8 bit processors that can run custom firmware. Such active RFID tags allow the tags themselves to actively take part into mobile RFID-enabled applications, sometimes by even allowing tag-to-tag communication without intermediary infrastructure.

One concrete example could be that the tags themselves are responsible for coordinating external access to their memory. This way, the tags could use for example leasing [GC89] to temporarily grant a remote party exclusive access to their memory while keeping it protected from writes from other parties.

Additionally, because of their integrated power source, active tags are able to power up a larger amount of memory, which would allow us to store larger objects on the tags themselves.

As RFID tags get more powerful, the distinction between RFID tags as pure identification devices and full-blown sensor nodes is blurred. Hence, we believe that the work carried out in the field of sensor networks will become more and more important outside of its own research field.

8.4.2 Content-based Publish/Subscribe

In this dissertation, we have presented node-centric ambient-oriented dataflow constructs that use an underlying publish/subscribe system. Matching subscriptions to publications can happen both topic-based and content-based. However, we do not take advantage of optimizations that can be performed when using content-based subscriptions, such as routing, query optimization etc. One interesting avenue would be to investigate how far known optimization strategies can be taken into an entirely decentralized model, or to come up with novel decentralized optimizations.

For the visual network-centric dataflow language AmbientTalk/R^V discussed in the previous chapter, we have only provided syntax for topic-based subscriptions, although technically content-based subscriptions are possible as well since the implementation is directly based on the node-centric primitives. Another avenue could be to investigate how a programmer could express content-based subscriptions on such a network-level view, and how it can be integrated with the semantics of the network-centric language.

8.4.3 Distributed Glitch Prevention

As mentioned before in section 8.3.2, our implementation of dataflow constructs offers no glitch prevention across different distributed nodes. The loosely-coupled nature of a publish/subscribe system such as the one on which our constructs are built relaxes many properties that a non-distributed dataflow programming system exploits to be able to prevent glitches, such as for example turn-based synchronous execution based on a single clock. The absence of this property makes it harder to prevent glitches in a distributed reactive program.

A possible solution could be to use a centralized entity – or an elected leader – to which all parties involved in the distributed dataflow program connect and that is solely responsible for ordering events. This centralized approach of course introduces a single point of failure and possibly a considerable communication overhead as all parties involved in the system have to communicate with a single host for every event they signal and to receive every event propagated to them as well, potentially limiting scalability.

A decentralized solution could be to accept that events in close succession cannot be ordered as a fact of life and take into account a minimum time interval in which events are considered to occur simultaneously. This is similar to ideas found in real-time synchronous languages where the system is assumed to react atomically to events *before any other events occur* and a global clock with a minimal tick rate determines the time interval. This minimal tick rate could be used as the maximal amount of which distributed clocks may diverge in a distributed dataflow program. As long as it can be guaranteed that all the clocks in the system do not diverge more than this time interval, glitches can be prevented while keeping a decentralized architecture. The applicability of this assumption depends on a number of factors such as the number of parties involved in the distributed interaction, the amount of clock divergence, the quality of the network, and, most importantly, the nature of the dataflow program. For programs that have to quickly react on events occurring in very close succession this approach might not be feasible, while for programs that work on human time-scales (such as seconds, minutes...) this assumption might be acceptable. Decentralized solutions to mitigate this problem exist, but they highly depend on the physical layer of the mobile ad hoc network [EGE02].

In short, we believe loose coupling and glitch freedom are two opposing forces and one of our most obvious objectives for future work is exploring the possibilities in which both properties can be combined and taxonomize the resulting trade-offs.

8.4.4 Bidirectional Dataflow

One of the dimensions of dataflow programming that we have not explored is bidirectional dataflow, as in the Coherence language [Edw09]. This way, side effects performed on reactive values derived from other reactive values are propagated back to the original reactive values on which they depend in such a way that their state is mutually consistent, as defined by a programmer-supplied function. This induces a stronger causal connection between distributed reactive code and blurs the distinction with *constraint programming* [SRP91]. It could in some situations reduce manual event detection and handling even further .

It seems an interesting avenue to explore bidirectional dataflow in a distributed setting, where the looser coupling between dependent reactive values and the fact that clients may not always be allowed to just modify remote values will require us to reconsider the semantics of the model. Additionally, local concurrent implementations such as Coherence will most likely have to be adapted to guarantee transactional semantics in a loosely-coupled distributed system.

8.4.5 Object Capabilities as a Security Model

As mentioned in section 8.3.1, we did not consider security in this work. One possible avenue to tackle this problem are *object capabilities* [MTS05b]. Object capabilities enforce security by enforcing the *principle of least authority* using standard object-oriented techniques such as encapsulation. This principle states that no process should receive more authority than required to do what it is intended to do. Object capabilities have been successfully implemented in the E language [MTS05a] and on a dialect of JavaScript called Caja. Since AmbientTalk is inspired by the E language and shares its computational model based on communicating event loops, it could be interesting to add object capabilities to our implementation. Naturally, object capabilities provide security on a programmatic level and assume a secure environment to run secure

programs on. Unfortunately, the RFID technology on which mobile RFID-enabled applications are built does not offer full security yet [RK09].

8.4.6 Complex Event Processing

With the approach presented in this dissertation, event processing happens using a general-purpose programming language, albeit enriched with dataflow semantics for reacting to events. Complex event processing [LF98] is a form of event processing where a declarative query language is used to generate higher level events from logged lower level events, which may be complex events themselves. It explicitly reifies time through concepts such as sliding windows, i.e. time periods in which specific events must occur to be successfully matched to a complex event. Other than having a dedicated query language, complex event processing allows event queries to be highly optimized. This could be useful in a setting where a massive amount of events is signaled, such as for example a network of long-range RFID readers in an environment which is densely populated with RFID-tagged objects [JLKY08]. Integrating complex event processing into a general-purpose programming language requires an event representation that is both useful for standard imperative operations and for complex event queries. Having an efficient mapping or symbiosis between both approaches remains an interesting avenue for future research.

8.4.7 Session Types

Session types [Hon93] are used for statically describing the behavior of processes in terms of sending and receiving messages. A session type describes with which other participants and in what order a participant in a distributed interaction may interact, specified as a list of incoming and outgoing typed messages (which can contain conditional branches and loops). Recently, session types were extended with support for interactions where the participants are not known upfront, but instead can join and leave a session at any point in time and even play different roles in the same session [DY11], which makes them applicable for mobile RFID-enabled applications. In these dynamic multiparty session types, potential participants are denoted by abstract *roles*, which is the same idea as we have used in our network-centric dataflow language to denote classes of participants that share a common behavior.

This work could be used to introduce more type safety into network-centric programs such as the ones defined by our network-centric ambient-oriented dataflow language. Additionally, they could be used as a framework to formally derive and prove certain properties of a network-centric program. In the next section, we look at a concrete tool for formalizing our approach.

8.4.8 Formalization

The approach taken in this dissertation is to extract a number of properties for a computational model to be suitable for mobile RFID-enabled applications. Out of this model, we extracted a number of programming model requirements that together formed the basis for a prototype implementation, which served as a proof by construction. The obtained artifacts and applications give us what we believe to be accurate ideas on how our abstractions should behave and to which applications they can be applied. However, in this dissertation we have not formally proven the properties that we have extracted out of our informal analyses. To this end, it could be interesting to model

our dataflow coordination model in for example Reo [Arb04], a formally backed coordination model based on hierarchically organized channels that connect components, which seems a close fit to our dataflow model. Such a formalization could serve as a formal backing for the work introduced in this dissertation.

8.5 Concluding Remarks

The Internet of Things allows everyday objects to be enriched with computational infrastructure and to be digitally represented in software applications. As a future vision on computing that is slowly becoming reality, it demands suitable software engineering tools and abstractions to manageably interact with a fluid network of heterogeneous devices from a software level. The wealth of knowledge built up over the years on different distributed computing techniques hints at what techniques will be appropriate. A phenomenon that we already observe today is that there is a departure from classic procedure-oriented software to increasingly event-driven software to allow software to be reactive to the plethora of events generated by an increasingly digitalized and interactive physical environment. In this dissertation, we investigated a specific class of such applications that arise when mobile ad hoc networks are combined with RFID technology. We have called these applications *mobile RFID-enabled applications*.

The work presented in this dissertation attempts to bring this vision closer to reality by mapping tried-and-tested object-oriented software engineering principles to one of the cornerstones of the Internet of Things; RFID technology. By doing this exercise, well-known problems – such as inversion of control – caused by combining event-driven execution with imperative programming become clearly identifiable. In hindsight, we observe that an increasingly event-driven nature of software renders the control flow a less optimal concept to use as the primary execution and composition tool for mobile RFID-enabled applications. Instead, the data flow of such software manifests itself as a more practical conceptual tool by which programmers can structure their applications. This led us to integrate the dataflow programming paradigm with distributed programming techniques.

Still, the loose coupling required by mobile RFID-enabled applications is a force that opposes the structured execution of a dataflow program. In this work, we presented an attempt at integrating dataflow programming with ambient-oriented programming by relying on publish/subscribe, which is a distributed event-driven computing technique known to offer a very loose coupling between remote parties and to be highly scalable. We proposed to alleviate the tension between both techniques by offering a network-centric dataflow representation of such a distributed program, an idea sparked by the growing convergence between sensor network applications and general-purpose software. We extended the basic dataflow paradigm with constructs to deal with the dynamicity of loosely-coupled mobile RFID-enabled applications, namely different types of dataflow dependencies. This allows the coordination behavior of such applications to be easily grasped and adapted.

Just like the original work on dataflow programming in the functional programming community spawned a large body of research, steadily improving the state of the art over the years and spreading over to various other research domains, we envision the same to happen in a distributed computing context. We believe distributed glitch prevention is one of the main issues to investigate.

Bibliography

[ABC+05] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884, New York, NY, USA, 2005. ACM.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

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

[AHT+02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[AJAM09] Jameela Al-Jaroodi, Junaid Aziz, and Nader Mohamed. Middleware for rfid systems: An overview. *Computer Software and Applications Conference, Annual International*, 2:154–159, 2009.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004.

[Arn07] D. P. Arnold. Review of microscale magnetic power generation. *IEEE TRANSACTIONS ON MAGNETICS*, 43(11):3940–3951, 2007.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20:519–526, July 1977.

[BA08] B. J. Bowers and D. P. Arnold. Spherical magnetic generators for biomotional energy harvesting. In *Tech. Dig. 8th Int. Workshop on Micro and Nanotechnology for Power Generation and Energy Conversion Apps.*, pages 281–284, September 2008.

[BC04] Gregory Biegel and Vinny Cahill. A framework for developing mobile, context-aware applications. In *PERCOM '04: Proceedings of*

the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04), page 361, Washington, DC, USA, 2004. IEEE Computer Society.

[BCM+99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 262, Washington, DC, USA, 1999. IEEE Computer Society.

[BCPV03] Roberto Baldoni, Mariangela Contenti, Sara Tucci Piergiovanni, and Antonino Virgillito. Modelling publish/subscribe communication systems: Towards a formal approach. In *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), 15-17 January 2003, Guadalajara, Mexico*, pages 304–311. IEEE Computer Society, 2003.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[BGS00] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. *IEEE Personal Communications*, 7:10–15, 2000.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM '01, pages 3–14, London, UK, 2001. Springer-Verlag.

[BH95] Ed Baroth and Chris Hartsough. *Visual programming in the real world*, pages 21–42. Manning Publications Co., Greenwich, CT, USA, 1995.

[BJdL+04] Ioana Burcea, Hans-Arno Jacobsen, Eyal de Lara, Vinod Muthusamy, and Milenko Petrovic. Disconnected operation in publish/subscribe middleware. In *5th IEEE International Conference on Mobile Data Management (MDM 2004), 19-22 January 2004, Berkeley, CA, USA*, pages 39–. IEEE Computer Societ, 2004.

[Ble06] Julian Bleecker. A manifesto for networked objects — cohabiting with pigeons, arphids and aibos in the internet of things, 2006.

[BP07] Biörn Biörnstad and Cesare Pautasso. Let it flow: Building mashups with data processing pipelines. In *Service-Oriented Computing - ICSOC 2007 Workshops, Revised Selected Paper*, volume 4907 of *Lecture Notes in Computer Science*, pages 15–28, Vienna, Austria, September 2007. Springer.

[BPA06] Biorn Biornstad, Cesare Pautasso, and Gustavo Alonso. Control the flow: How to safely compose streaming services into business processes. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '06, pages 206–213, Washington, DC, USA, 2006. IEEE Computer Society.

[Bri88] J.-P. Briot. From objects to actors: study of a limited symbiosis in smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 69–72, New York, NY, USA, 1988. ACM Press.

[BTT+07] S. P. Beeby, R. N. Torah, M. J. Tudor, Glynne P. Jones, Donnell, C. R. Saha, and S. Roy. A micro electromagnetic generator for vibration energy harvesting. *Journal of Micromechanics and Microengineering*, 17(7):1257–1265, 2007.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–343, 2004.

[BVT+09] Engineer Bainomugisha, Jorge Vallejos, Éric Tanter, Elisa Gonzalez Boix, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Resilient actors: a runtime partitioning model for pervasive computing services. In *Proceedings of the 2009 international conference on Pervasive services*, ICPS '09, pages 31–40, New York, NY, USA, 2009. ACM.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Conference on Innovative Data Systems Research*, 2003.

[CBM+02] Licia Capra, Gordon S. Blair, Cecilia Mascolo, Wolfgang Emmerich, and Paul Grace. Exploiting reflection in mobile computing middleware. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6:34–44, October 2002.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–, New York, NY, USA, 2003. ACM.

[CD96] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 3–14, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[CGG+05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Tinylime: Bridging mobile and sensor networks through middleware. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 61–72, Washington, DC, USA, 2005. IEEE Computer Society.

[CGHP04] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pages 230–239, New York, NY, USA, 2004. ACM.

[CGL96] Philip T. Cox, Hugh Glaser, and Benoît Lanaspre. Distributed prograph: Extended abstract. In *International Workshop on Parallel Symbolic Languages and Systems*, pages 128–133, London, UK, 1996. Springer-Verlag.

[CH07] V. Chawla and Dong Sam Ha. An overview of passive rfid. *IEEE Communications Magazine*, 45(9):11 – 17, October 2007.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *15th European Symposium on Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2006.

[CLC+02] Norman H. Cohen, Hui Lei, Paul Castro, John S. Davis II, and Apratim Purakayastha. Composing pervasive data using iql. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 94, Washington, DC, USA, 2002. IEEE Computer Society.

[CLK04] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. *Annual International Conference on Mobile and Ubiquitous Systems*, 0:246–255, 2004.

[CM06] Brian Chin and Todd D. Millstein. Responders: Language support for interactive applications. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 255–278. Springer, 2006.

[CNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM.

[Cou01] Antony Courtney. Frappé: Functional reactive programming in java. In *PADL '01: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, pages 29–44, London, UK, 2001. Springer-Verlag.

[CPWY02] Norman H. Cohen, Apratim Purakayastha, Luke Wong, and Danny L. Yeh. iqueue: A pervasive data composition framework. In *MDM '02: Proceedings of the Third International Conference on Mobile Data Management*, page 146, Washington, DC, USA, 2002. IEEE Computer Society.

[Cro06] Douglas Crockford. The application/json media type for JavaScript object notation (JSON). RFC 4627 (Informational), July 2006.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227, New York, NY, USA, 2000. ACM Press.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[DDD05] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Pico: Scheme for mere mortals. In Jacques Malenfant and Bjarte M. Østvold, editors, *Object-Oriented Technology: ECOOP 2004 Workshop Reader, ECOOP 2004 Workshops, Oslo, Norway, June 14-18, 2004, Final Reports*, volume 3344 of *Lecture Notes in Computer Science*. Springer, 2005.

[Ded06] Jessie Dedecker. *Ambient-Oriented Programming*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2006.

[DGP08] Gwenaël Delaval, Alain Girault, and Marc Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 101–110, New York, NY, USA, 2008. ACM.

[DMS07] Thomas Diekmann, Adam Melski, and Matthias Schumann. Data-on-network vs. data-on-tag: Managing data in complex rfid environments. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 224a–, Washington, DC, USA, 2007. IEEE Computer Society.

[DVCM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 31–40, New York, NY, USA, 2005. ACM.

[DVM+06] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 435–446, New York, NY, USA, 2011. ACM.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[Edw09] Jonathan Edwards. Coherent reaction. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 925–932, New York, NY, USA, 2009. ACM.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36:147–163, December 2002.

[EGH05] Patrick Th. Eugster, Benoît Garbinato, and Adrian Holzer. Location-based publish/subscribe. In *NCA '05: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, pages 279–282, Washington, DC, USA, 2005. IEEE Computer Society.

[EGS00] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 252–276, London, UK, 2000. Springer-Verlag.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, volume 32(8), pages 263–273, 1997.

[EJ09] Patrick Th. Eugster and K. R. Jayaram. Eventjava: An extension of java for event correlation. In *ECOOP 2009 - Proceedings of the 23rd European Conference on Object-Oriented Programming*, Genoa, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.

[Ell09] Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[EPC10a] EPCGlobal. The application level events (ale) specification, version 1.0, September 2010.

[EPC10b] EPCGlobal Standards Overview. http://www.epcglobalinc.org/standards, September 2010.

[Eug07] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29, January 2007.

[FL05] Christian Floerkemeier and Matthias Lampe. Rfid middleware design: addressing application requirements and rfid constraints. In *sOcEUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence*, pages 219–224, New York, NY, USA, 2005. ACM.

[FRL07] Christian Floerkemeier, Christof Roduner, and Matthias Lampe. Rfid application development with the accada middleware platform. *IEEE Systems Journal, Special Issue on RFID Technology*, 1(2):82–94, December 2007.

[FW87] A. A. Faustini and W. W. Wadge. An eductive interpreter for the language lucid. *SIGPLAN Not.*, 22(7):86–91, July 1987.

[FW06] Christian Floerkemeier and Matthias Wille. Comparison of transmission schemes for framed aloha based rfid protocols. In *SAINT-W '06: Proceedings of the International Symposium on Applications on Internet Workshops*, pages 92–97, Washington, DC, USA, 2006. IEEE Computer Society.

[GC89] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[GDH+01] Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam Macbeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Programming for pervasive computing environments. *In Proceedings of the 18th ACM Symposium on Operating Systems Principle*, Chateau Lake Louise, Banff, Canada., October 2001.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Gri04] Robert Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Computer Languages, Systems & Structures*, 32(2-3):109–124, 2006.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HGM04] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wireless Networks*, 10(6):643–652, 2004.

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin / Heidelberg, 1993.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[Jag95] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE Computer Society Press, 1995.

[JDA97] R. Jagannathan, C. Dodd, and Iskender Agi. Glu: A high-level system for granular data-parallel programming. *Concurrency - Practice and Experience*, 9(1):63–83, 1997.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[JLKY08] Xingyi Jin, Xiaodong Lee, Ning Kong, and Baoping Yan. Efficient complex event processing over rfid data stream. In *7th IEEE/ACIS International Conference on Computer and Information Science*, pages 75–81, Washington, DC, USA, 2008. IEEE Computer Society.

[Kal95] Cor Kalkman. Labview: A software system for data acquisition, data analysis, and instrument control. *Journal of Clinical Monitoring and Computing*, 11(1):51–58, 1995.

[KB02] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 72–73, New York, NY, USA, 2002. ACM.

[KBX+04] Porlin Kang, Cristian Borcea, Gang Xu, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal*, 47(4):475–494, April 2004.

[Kim93] Takayuki Dan Kimura. Hyperflow: a uniform visual language for different levels of programming. In *Proceedings of the 1993 ACM conference on Computer science*, CSC '93, pages 209–214, New York, NY, USA, 1993. ACM.

[KK07] T. Kaya and H. Koser. A new batteryless active rfid system: Smart rfid. *RFID Eurasia, 2007 1st Annual*, pages 1–4, 2007.

[KKPG98] John Kymissis, Clyde Kendall, Joseph Paradiso, and Neil Gershenfeld. Parasitic power harvesting in shoes. In *ISWC '98: Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, page 132, Washington, DC, USA, 1998. IEEE Computer Society.

[KLS+08] Nikos Kefalakis, Nektarios Leontiadis, John Soldatos, Kiev Gama, and Didier Donsez. Supply chain management and NFC picking demonstrations using the AspireRfid middleware platform. In *ACM/IFIP/USENIX Middleware '08*, pages 66–69, New York, NY, USA, 2008. ACM.

[KR91] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[KR05] Oliver Kasten and Kay Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *4th international symposium on Information processing in sensor networks*, page 7, Piscataway, NJ, USA, 2005. IEEE Press.

[LB07] Chuan-kai Lin and Andrew Black. Directflow: A domain-specific language for information-flow systems. In Erik Ernst, editor, *ECOOP 2007*, volume 4609 of *Lecture Notes in Computer Science*, pages 299–322. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73589-215.

[LC02] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *SIGPLAN Not.*, 37:85–95, October 2002.

[LD10] Andoni Lombide Carreton and Theo D'Hondt. A hybrid visual dataflow language for coordination in mobile ad hoc networks. In Dave Clarke and Gul A. Agha, editors, *Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6116 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2010.

[LF98] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.

[Lie87] Henry Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[LKKP10] Donggeon Lee, Seongyun Kim, Howon Kim, and Namje Park. Mobile platform for networked rfid applications. *Third International Conference on Information Technology: New Generations*, 0:625–630, 2010.

[LL10] Yuan-Ping Luh and Yin-Chang Liu. Reading rate improvement for uhf rfid systems with massive tags by the q parameter. *Wireless Personal Communications*, pages 1–11, 2010. 10.1007/s11277-010-0198-y.

[LLS+04] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems*, 26:351–368, 2004. 10.1023/B:TELS.0000029046.79337.8f.

[LMVD10] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010.

[LPD10] Andoni Lombide Carreton, Kevin Pinte, and Wolfgang De Meuter. Distributed object-oriented programming with rfid technology. In Frank Eliassen and Rüdiger Kapitza, editors, *Distributed Applications and Interoperable Systems, 10th IFIP WG 6.1 International Conference, DAIS 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6115 of *Lecture Notes in Computer Science*, pages 56–69, 2010.

[LPD11] Andoni Lombide Carreton, Kevin Pinte, and Wolfgang De Meuter. Software abstractions for mobile rfid-enabled applications. *Software: Practice and Experience*, pages n/a–n/a, 2011.

[MÖ1] Gero Mühl. Generic constraints for content-based publish/subscribe. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 211–225, London, UK, 2001. Springer-Verlag.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[Mat05] Friedemann Mattern. *Ubiquitous Computing: Scenarios from an informatised world*, pages 145–163. Springer-Verlag, 2005.

[MC02a] René Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.

[MC02b] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. In *ICDCSW '02: 22nd International Conference on Distributed Computing Systems*, pages 585–588, Washington, DC, USA, 2002. IEEE Computer Society.

[McA95] Jeff McAffer. Meta-level programming with coda. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 190–214, London, UK, UK, 1995. Springer-Verlag.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In Enrico Gregori, Giuseppe Anastasi, and Stefano Basagni, editors, *Advanced Lectures on Networking*, volume 2497 of *Lecture Notes in Computer Science*, pages 506–510. Springer Berlin / Heidelberg, 2002.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[MF02] Samuel Madden and Michael J Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 555–, Washington, DC, USA, 2002. IEEE Computer Society.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[MGB+09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–20, New York, NY, USA, 2009. ACM.

[MMH05] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. Emma: Epidemic messaging middleware for ad hoc networks. *Personal Ubiquitous Comput.*, 10(1):28–36, 2005.

[MMW08] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 335–346, New York, NY, USA, 2008. ACM.

[MPMJPPS05] Alberto Martinez, Marta Patino-Martinez, Ricardo Jimenez-Peris, and Francisco Perez-Sorrosal. Zenflow: A visual web service composition tool for bpel4ws. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 181–188, Washington, DC, USA, 2005. IEEE Computer Society.

[MPR01] A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

[MQZ06] Marco Mamei, Renzo Quaglieri, and Franco Zambonelli. Making tuple spaces physical with rfid tags. In *Symposium on Applied computing*, pages 434–439, New York, NY, USA, 2006. ACM.

[MRO10] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the Observer Pattern. Technical report, École Polytechnique Fédérale de Lausanne, 2010.

[MsS97] Masoud Mansouri-samani and Morris Sloman. Gem: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4:96–108, 1997.

[MTCS07] Adam Melski, Lars Thoroe, Thorsten Caus, and Matthias Schumann. Beyond epc - insights from multiple rfid case studies on the storage of additional data on tag. In *Proceedings of the International Conference on Wireless Algorithms, Systems and Applications*, WASA '07, pages 281–286, Washington, DC, USA, 2007. IEEE Computer Society.

[MTS05a] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[MTS05b] Mark Miller, Bill Tulloh, and Jonathan Shapiro. The structure of authority: Why security is not a separable concern. In Peter Van Roy, editor, *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *Lecture Notes in Computer Science*, pages 2–20. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-31845-32.

[MVTT07] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the Dynamic Languages Symposium - OOPSLA'07: Companion of the 22st annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications.*, pages 222–248. ACM Press, 2007.

[Mye91] Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *UIST '91: Proceedings of the 4th annual ACM symposium on User interface software and technology*, pages 211–220, New York, NY, USA, 1991. ACM.

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *IEEE International Conference on Pervasive Computing and Communications*, page 263, Washington, DC, USA, 2004. IEEE Computer Society.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, New York, NY, USA, 2002. ACM.

[NKSI05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 249–260, New York, NY, USA, 2005. ACM.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 489–498, New York, NY, USA, 2007. ACM.

[OM02] Oat Systems and MIT Auto-ID Center. The savant version 0.1. Technical report, Massachusetts Institute of Technology, 77 Massachusetts Avenue, bldg 3-449, Cambridge, MA02139-4307, USA, 2002.

[OM08] Dominic Orchard and Steve Matthews. Integrating lucid's declarative dataflow paradigm into object-orientation. *Mathematics in Computer Science*, 2:103–122, 2008. 10.1007/s11786-008-0046-6.

[Ora] Oracle (Sun Developer Network). Developing auto-id solutions using sun java system rfid software.

[PA05] Cesare Pautasso and Gustavo Alonso. The jopera visual composition language. *J. Vis. Lang. Comput.*, 16:119–152, February 2005.

[PB02] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[PD10] Tomas Petricek and Sy Don. Collecting hollywood's garbage: avoiding space-leaks in composite events. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 53–62, New York, NY, USA, 2010. ACM.

[PE02] C. Petitpierre and A. Eliens. Active Objects Provide Robust Event-Driven Applications. In *The 2002 International Conference on Software Engineering Research and Practice (SERP)*, 2002.

[PHD11] Kevin Pinte, Dries Harnie, and Theo D'Hondt. Enabling cross-technology mobile applications with network-aware references. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages*, volume 6721 of *Lecture Notes in Computer Science*, pages 142–156. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21464-6_10.

[PHRH01] John Peterson, Paul Hudak, Alastair Reid, and Gregory D. Hager. Fvision: A declarative language for visual tracking. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 304–321, London, UK, 2001. Springer-Verlag.

[PKH05] Ted Phillips, Tom Karygiannis, and Rick Huhn. Security standards for the rfid market. *IEEE Security and Privacy*, 3(6):85–89, 2005.

[PMD08] John Plaice, Blanca Mancilla, and Gabriel Ditu. From lucid to translucid: Iteration, dataflow, intensional and cartesian programming. *Mathematics in Computer Science*, 2:37–61, 2008. 10.1007/s11786-008-0043-9.

[PPS+05] K. Penttila, N. Pere, M. Sioni, L. Sydanheimo, and M Kivikoski. Use and interface definition of mobile rfid reader integrated in a smart phone. In *Proceedings of the Ninth International Symposium on Consumer Electronics 2005 (ISCE 2005)*, pages 353 – 358. IEEE Computer Society, September 2005.

[PS05] Joseph A. Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005.

[PS11] Sarita Pais and Judith Symonds. Data storage on a rfid tag for a distributed system. *International Journal Of UbiComp (IJU)*, 2(2):26–39, April 2011.

[PSR+08] B. S. Prabhu, Xiaoyong Su, Harish Ramamurthy, Chi-Cheng Chu, and Rajit Gadh. Winrfid – a middleware for the enablement of radio frequency identification (rfid) based applications. white paper, UCLA – Wireless Internet for the Mobile Internet Consortium, January 2008.

[RGDMC09] Andrea Ricci, Matteo Grisanti, Ilaria De Munari, and Paolo Ciampolini. Improved pervasive sensing with rfid: an ultra-low power baseband processor for uhf tags. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(12):1719–1729, 2009.

[RK09] George Roussos and Vassilis Kostakos. RFID in pervasive computing: State-of-the-art and outlook. *Pervasive Mob. Comput.*, 5(1):110–131, 2009.

[RSMD04] Kay Romer, Thomas Schoch, Friedemann Mattern, and Thomas Dubendorfer. Smart identification frameworks for ubiquitous computing applications: Pervasive computing and communications (guest editors: Mohan Kumar, Diane Cook and Anand Tripathi). *Wireless Networks*, 10(6):689+, 2004.

[SBC05] Thirunavukkarasu Sivaharan, Gordon S. Blair, and Geoff Coulson. Green: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *Distributed Objects, Middleware, and Applications*, volume 3760 of *Lecture Notes in Computer Science*, pages 732–749. Springer, 2005.

[Ser09] Manuel Serrano. Hop, a fast server for the diffuse web. In *Proceedings of the 11th International Conference on Coordination Models and Languages*, COORDINATION '09, pages 1–26, Berlin, Heidelberg, 2009. Springer-Verlag.

[SFR05] Roberto S. Silva Filho and David F. Redmiles. Striving for versatility in publish/subscribe infrastructures. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 17–24, New York, NY, USA, 2005. ACM.

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008.

[SGP04] Katrine Stemland Skjelsvik, Vera Goebel, and Thomas Plagemann. Distributed event notification for mobile ad hoc networks. *IEEE Distributed Systems Online*, 5(8):2, 2004.

[SJS00] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. In *Proceedings of the 2000 International Workshop on Parallel Processing*, ICPP '00, pages 23–, Washington, DC, USA, 2000. IEEE Computer Society.

[SLU89] Lynn Andrea Stein, Henry Lieberman, and David Ungar. *A shared view of sharing: the treaty of Orlando*, pages 31–48. ACM, New York, NY, USA, 1989.

[Smi84] Brian Cantwell Smith. Reflection and semantics in LISP. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, New York, NY, USA, 1984. ACM Press.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 333–352, New York, NY, USA, 1991. ACM.

[SSYH06] Dong-Her Shih, Po-Ling Sun, David C. Yen, and Shi-Ming Huang. Taxonomy and survey of rfid anti-collision protocols. *Computer Communications*, 29(11):2150 – 2166, 2006.

[SYP+08] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, November 2008.

[TP05] Dirk Trossen and Dana Pavel. Building a ubiquitous platform for remote sensing using smartphones. *Annual International Conference on Mobile and Ubiquitous Systems*, 0:485–489, 2005.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4:223–242, July 1991.

[Uni05] International Telecommunication Union. Itu internet report 2005: The internet of things. 2005.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[VC03] P. Verissimo and A. Casimiro. Event-driven support of real-time sentient objects. In *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Guadalajara, Mexico, January 2003.

[VCDM+06] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez, Theo D'Hondt, and Wolfgang De Meuter. Ambient references: addressing objects in mobile networks. In *21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 986–997, New York, NY, USA, 2006. ACM.

[VCVCDM09] Jorge Vallejos, Pascal Costanza, Tom Van Cutsem, and Wolfgang De Meuter. Reconciling generic functions with actors. In *ACM SIGPLAN International Lisp Conference*, Cambridge, Massachusetts, 2009.

[VDM+06] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez, Theo D'Hondt, and Wolfgang De Meuter. Ambient references: addressing objects in mobile networks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 986–997, New York, NY, USA, 2006. ACM.

[VGC+10] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions: enabling context-dependent method dispatch. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 66–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[VMD07] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between actors and threads. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 222–248, New York, NY, USA, 2007. ACM.

[VMD08] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages Systems & Structures*, 35(1), 2008.

[VMG+07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Computer Science Society*, pages 3–12. IEEE Computer Society, 2007.

[Wal01] Jim Waldo. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 9, 2001.

[WD05] Marcel Weiher and Stéphane Ducasse. Higher order messaging. In *DLS '05: Proceedings of the 2005 conference on Dynamic languages symposium*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.

[Wei93] Mark Weiser. Ubiquitous computing. *IEEE Computer Hot Topics*, 1993.

[Wei05] Ron Weinstein. Rfid: A technical overview and its application to the enterprise. *IT Professional*, 7(3):27–33, 2005.

[WFGH99] Roy Want, Kenneth P. Fishkin, Anuj Gujar, and Beverly L. Harrison. Bridging physical and virtual worlds with electronic tags. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 370–377, New York, NY, USA, 1999. ACM.

[Whi97] K. N. Whitley. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, 8:109–142, 1997.

[WJ09] Vivienne Waller and Robert B. Johnston. Making ubiquitous computing available. *Commun. ACM*, 52(10):127–130, 2009.

[WM04] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 99–110, New York, NY, USA, 2004. ACM.

[WWWK96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1996.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.

[YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31:9–18, September 2002.

Index