Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen
Software Languages Lab

Workflow Abstractions for Orchestrating Services in Nomadic Networks

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

Eline Philips

Promotor: Prof. Dr. Viviane Jonckers

Februari 2013

Print: Silhouet, Maldegem

2013 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 5718 276 1
NUR 986 / 989
Legal deposit D/2013/11.161/036

ABSTRACT

Nowadays we are surrounded by all kinds of computing devices, ranging from smartphones to digital watches, and in the near future even digital glasses. Everyday objects are more and more integrated with computers, bringing us closer to Mark Weiser's vision of the *ubiquitous computing* era. The plethora of these miniaturised devices can be connected in so-called *mobile ad hoc networks*, using a variety of standard technologies such as WiFi, Bluetooth, NFC, 3G, and recently also 4G. Mobile ad hoc networks emerge from the collaboration of nearby devices using wireless network topologies, without the need of any configuration or infrastructure. This is in contrast to *nomadic networks*, a special case of mobile ad hoc networks, where mobile devices try to maintain a connection with a fixed infrastructure. As these kinds of networks are omnipresent (for instance, in shopping malls, airports, etc.), an abundance of interesting applications can be supported.

We envision a new type of rich applications, called *nomadic applications*, which are deployed and executed on the fixed infrastructure of the nomadic network, while communicating with nearby mobile devices. Such a nomadic application is centred around the orchestration of services that are provided by connected servers and nearby mobile devices. Orchestration of services enables the realisation of meaningful complex behaviour out of the interaction between the aforementioned mobile devices and the servers. The orchestration of these services is not without a challenge, since these nomadic networks are built upon volatile connections. Services residing on mobile devices are exposed to (temporary) network failures. Additionally, mobile services are always on the move, meaning that the set of services to be orchestrated is dynamic during the execution of a nomadic application and cannot be determined beforehand.

We propose a programming model that facilitates the development of nomadic applications. This programming model unifies the principles of two research pillars, namely the workflow paradigm, which focusses on the orchestration of services, and the ambient-oriented programming paradigm, which is tailored towards applications running on mobile devices. We define high-level abstractions as workflow patterns that allow the orchestration of services in a nomadic network. Therefore, we revise existing

control flow patterns in the context of nomadic networks where the different services are not necessarily known beforehand, and can become (temporarily) unavailable.

Additionally, the programming model offers a set of new patterns that allows the orchestration of dynamically changing groups of services. This set of patterns ensures that (part of) the application is executed for all services satisfying the group's description. The inherent volatile connections of the network cause communication partners to disconnect making full synchronisation not always possible. Therefore, we include more advanced synchronisation patterns, that enable control over the execution in a way that transcends the individual process of a single member.

Because nomadic networks are built upon volatile networks, failure handling mechanisms are required to overcome communication faults. The programming model we propose incorporates a default failure handling mechanism in order to overcome (network) failures during service invocations. We define additional patterns that allow application developers to specify compensating actions to handle particular failure events.

We contribute the design and implementation of these novel patterns in the context of NOW, a novel workflow language for nomadic networks. This workflow language is implemented as an extra layer of abstraction on top of AMBIENTTALK/2, the state-of-the-art programming language for coordination in mobile ad hoc networks. We also provide a validation of our approach by comparing the code complexity of the implementations of three nomadic applications in NOW and AMBIENTTALK/2. Finally, we present a validation of NOW's performance and scalability.

SAMENVATTING

Het huidige straatbeeld wordt gekenmerkt door de aanwezigheid van geminiaturiseerde apparaten zoals smartphones en digitale horloges, en binnenkort zelfs digitale brillen. Deze alledaagse objecten krijgen steeds meer rekenkracht, wat ons dichter brengt bij Mark Weiser's visie van *ubiquitous computing*. Deze talrijk aanwezige geminiaturiseerde apparaten kunnen verbonden worden in *mobiele ad-hocnetwerken* door gebruik te maken van standaardtechnologieën zoals WiFi, Bluetooth, NFC, 3G en recent ook 4G. Dergelijke mobiele ad-hocnetwerken ontstaan wanneer apparaten spontaan een samenwerking tot stand willen brengen door middel van een draadloos netwerk, zonder hiervoor configuratie of infrastructuur nodig te hebben. Dit is in tegenstelling tot *nomadische netwerken*, waar een groep mobiele apparaten een verbinding probeert te behouden met een vaste infrastructuur. Omdat dit type netwerken alomtegenwoordig is (denk bijvoorbeeld aan winkelcentra, luchthavens, etc.), ontstaan rijke toepassingsmogelijkheden.

Wij stellen ons een nieuw soort toepassingen voor, genaamd *nomadische toepassingen*, die ontwikkeld en uitgevoerd worden op de vaste infrastructuur van de nomadische netwerken en ondertussen complexe interacties met de nabije mobiele apparaten orkestreren. De focus van dergelijke nomadische toepassingen is het orkestreren van de verschillende diensten die worden aangeboden door verbonden servers en door nabije mobiele apparaten. Het orkestreren van diensten leidt tot complex gedrag dat ontstaat uit de interactie tussen deze aanwezige mobiele apparaten en de servers. Orkestratie van diensten in deze context is niet triviaal aangezien nomadische netwerken onderhevig zijn aan volatiele verbindingen. Dit houdt in dat diensten op mobiele apparaten gevoelig zijn voor (tijdelijke) netwerkfalingen. Bovendien zijn dergelijke mobiele apparaten steeds in beweging, wat wil zeggen dat de groep van diensten die georkestreerd wordt dynamisch is tijdens de uitvoering van een nomadische toepassing en dus bijgevolg niet op voorhand bepaald kan worden.

Wij stellen een programmeermodel voor dat de ontwikkeling van nomadische toepassingen vereenvoudigt. Dit programmeermodel unificeert de principes van twee onderzoekszuilen, namelijk het workflowparadigma dat zich toespitst op het orkestreren van diensten en het ambient-geöriënteerd programmeerparadigma dat gericht is op

toepassingen die worden uitgevoerd op mobiele apparaten. We definiëren hoogniveau abstracties als workflowpatronen die de orkestratie van diensten in een nomadisch netwerk ondersteunen. Daarvoor dienen we bestaande *control flow* patronen te herbekijken in de context van nomadische netwerken, waar niet noodzakelijk alle diensten op voorhand gekend zijn en waar deze (tijdelijk) onbeschikbaar kunnen zijn.

Bovendien biedt het voorgestelde programmeermodel een nieuwe groep van patronen aan die zich richten tot de orkestratie van een dynamische groep van diensten. Deze verzameling patronen zorgt er voor dat (een deel van) de toepassing uitgevoerd wordt voor alle diensten die aan de groepsbeschrijving voldoen. De volatiliteit die aanwezig is in nomadische netwerken zorgt er voor dat gehele synchronisatie niet altijd mogelijk is. Daarom stellen we meer geavanceerde synchronisatiepatronen voor die de uitvoering van de toepassing controleren op een manier die de individuele uitvoering van een enkel groepselement overstijgt.

Omdat nomadische netwerken gebouwd worden op volatiele netwerkverbindingen zijn mechanismen nodig om falingen tijdens de communicatie af te handelen. Het programmeermodel dat wij voorstellen, ondersteunt het automatisch afhandelen van falingen om zo te kunnen omgaan met (netwerk) falingen veroorzaakt tijdens het oproepen van een dienst. We definiëren eveneens extra patronen die toepassingsontwikkelaars toelaten om toepassingsspecifieke compensaties te specifiëren voor bepaalde soorten falingen.

We presenteren het design en de implementatie van deze nieuwe patronen in de context van NOW, een nieuwe workflowtaal voor nomadische netwerken. Deze workflowtaal is geïmplementeerd als een abstractielaag bovenop AMBIENTTALK/2, de state-of-the-art voor coördinatie in mobiele ad-hocnetwerken. We voorzien ook een validatie van onze aanpak door de codecomplexiteit van de implementaties van drie nomadische toepassingen in NOW en AMBIENTTALK/2 te vergelijken. Tenslotte presenteren we ook een validatie van de performantie en schaalbaarheid van NOW.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to everyone who helped me to complete this dissertation. First and foremost, I would like to thank my promotor Prof. Viviane Jonckers for genuinely supporting my work. Many thanks for your advice and encouragement throughout all these years.

I would also like to thank the members of my jury: Prof. Mario Südholt, Prof. Yolande Berbers, Prof. Kris Steenhaut, Prof. Ann Nowé, Prof. Tom Van Cutsem, and Prof. Wolfgang De Meuter. Thanks for critically reading my dissertation and improving the quality of this text. I want to thank Wolfgang De Meuter for helping me write my IWT proposal.

A big thanks goes out to the members of my reading-committe: Ragnhild Van Der Straeten, Andoni Lombide Carreton, and Jorge Vallejos. Thanks a lot for your valuable feedback on my text and your guidance throughout the years.

I want to thank all members of the Software Languages Lab, especially the members of the System and Software Engineering Lab and of the ambient group. Special thanks goes to Ragnhild Van Der Straeten, Mario Sánchez, Oscar González, Niels Joncheere, Andoni Lombide Carreton, Dries Harnie, and Tom Van Cutsem. I also owe a a lot of gratitude to my office mates, and in particular to Stefan Marr. Sharing an office with someone who is also writing his dissertation could have caused unnecessary stress. For us, it did not. I really enjoyed sharing this process with someone who was experiencing the exact same things. The daily “How many pages did you write?”, “Do you have to make a lot of adjustments?”, etc. questions were a motivation to keep on writing. Thank you, Stefan, for being there for me and for providing me with the really delightful chocolate!

I also want to thank the assistants with whom I shared teaching tasks. Special thanks to Ragnhild Van Der Straeten for helping me survive my first classes and thanks to Mattias De Wael for giving me an extra hand during these last months. I also want to express my genuine gratitude to all students who attended my classes during the years:

thanks for your interest and making it a pleasure to teach!

I also want to express my gratitude to some people who supported me, even though they might not be fully aware of how much they did. Special thank goes out to Charlotte Herzeel, Mattias De Wael, Reinout Stevens, Stefan Marr, Dries Harnie, Kevin Pinte, Lode Hoste, Yves Vandriessche, Nicolás Cardozo, and Andy Kellens.

Last, but not least, I would like to thank my family and friends who were there for me when I needed them. First of all I want to express my genuine gratitude to my parents who unconditionally supported me. Without their support, nothing of this would have been possible. During these last months I sometimes did honour to my nickname of "mimosa pudica", but they still managed to be there even though I was not always pleasant company. I also want to thank my brother Michaël and sister Laure for supporting me and providing me with the necessary distractions.

I want to thank my friends who joined me during my swimming sessions, dragged me to the local pub, or entertained me with a night of board game fun! Thanks Bart, Sarina, Reinout, Nicky, Tas (aka "Thomas"), Robrecht, Lilith, Brecht, Glenn, Hendrik, Judith, Florence, Joeri, Robin, and Jourik.

There is one person to whom I am deeply indebted, she is not only my little sister, but also my best friend, and now even my colleague. Last year I was really touched when she thanked me in the acknowledgements of her master's thesis and mentioned that I was her inspiration. Well, this time it's my time to thank her for everything she has done for me. Thank you Laure for supporting me, pushing me when I needed it, convincing me that I needed to take a break, for providing me with some nice guitar play and the a-bit-more-annoying noise that came out of that piccolo of yours, and for just being there when I needed it the most! I hope I can be the same support for you when it is your time to write your dissertation.

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

CONTENTS

1 Introduction **1**
- 1.1 Research Context . . . 1
- 1.2 Research Vision . . . 3
- 1.3 Research Objective . . . 5
- 1.4 Research Methodology . . . 7
- 1.5 Contributions . . . 8
- 1.6 Dissertation Roadmap . . . 10

2 Orchestration in Nomadic Networks **13**
- 2.1 Workflows and Orchestration . . . 13
  - 2.1.1 Service Orchestration . . . 14
  - 2.1.2 Workflows . . . 14
  - 2.1.3 Terminology . . . 19
- 2.2 Ambient-Oriented Programming . . . 20
  - 2.2.1 Terminology . . . 22
- 2.3 Orchestration in Nomadic Networks . . . 23
  - 2.3.1 Scenarios of Nomadic Applications . . . 23
  - 2.3.2 Definitions . . . 25
  - 2.3.3 Criteria for Service Orchestration . . . 26
  - 2.3.4 Criteria for Group Orchestration . . . 28
  - 2.3.5 Criteria for Failure Handling . . . 29
- 2.4 Conclusion . . . 30

3 Ambient-Oriented Programming in AMBIENTTALK **31**
- 3.1 Ambient-Oriented Programming . . . 32
- 3.2 AMBIENTTALK . . . 33
- 3.3 Object-Oriented Programming in AMBIENTTALK . . . 34
  - 3.3.1 Delegation . . . 35
  - 3.3.2 Scoping . . . 36

3.3.3 Encapsulation . . . 37
3.4 Concurrent Programming in AMBIENTTALK . . . 38
3.4.1 Asynchronous Message Sending . . . 40
3.4.2 Isolates . . . 40
3.4.3 Futures . . . 40
3.5 Distributed Programming in AMBIENTTALK . . . 41
3.5.1 Exporting and Discovering of Objects in AMBIENTTALK . . . 42
3.5.2 Dealing with Failures . . . 44
3.6 Reflective Programming in AMBIENTTALK . . . 45
3.6.1 Mirrors . . . 46
3.6.2 Mirages . . . 48
3.7 Ambient References . . . 50
3.8 Limitations . . . 52
3.9 Conclusion . . . 56

4 Patterns for Orchestration in Nomadic Networks 59
4.1 Activities . . . 60
4.2 Data Flow . . . 62
4.3 Patterns for Service Orchestration . . . 67
4.3.1 Standard Patterns . . . 69
4.3.2 Synchronisation Patterns . . . 72
4.3.3 Trigger Patterns . . . 76
4.4 Patterns for Group Orchestration . . . 77
4.4.1 Definition of Group Membership . . . 78
4.4.2 Synchronisation Mechanisms . . . 84
4.4.3 Relation to Existing Research . . . 90
4.5 Patterns for Failure Handling . . . 92
4.5.1 Automatic Failure Handling . . . 92
4.5.2 Specification of Compensating Actions as a Failure Handling Mechanism . . . 93
4.5.3 Failure Handling for Group Orchestration . . . 97
4.5.4 Relation to Existing Research . . . 105
4.6 Conclusion . . . 108

5 A Workflow Language for Orchestration in Nomadic Networks 111
5.1 Motivation . . . 112
5.2 Activities . . . 113
5.3 Data Flow . . . 116

5.4 Service Orchestration . . . 118
5.4.1 Composition of Synchronisation Patterns . . . 121
5.4.2 Implementing the iMPASSE Application in NOW . . . 122
5.5 Group Orchestration . . . 127
5.5.1 Definition of Group Membership . . . 128
5.5.2 Implementing the SURA Application in NOW . . . 131
5.6 Failure Handling . . . 135
5.6.1 Failure Handling for Service Orchestration . . . 136
5.6.2 Failure Handling for Group Orchestration . . . 138
5.6.3 Implementing the SWOOP Application in NOW . . . 139
5.7 NOW Related to the State of the Art . . . 144
5.7.1 Workflow Languages . . . 144
5.7.2 Coordination Languages . . . 155
5.7.3 Summary . . . 163
5.8 Conclusion . . . 165

6 Implementing NOW **167**
6.1 Activities . . . 168
6.2 Data Flow . . . 170
6.3 Service Orchestration . . . 174
6.3.1 Standard Patterns . . . 176
6.3.2 Synchronisation Patterns . . . 182
6.3.3 Trigger Patterns . . . 185
6.4 Group Orchestration . . . 187
6.4.1 Definition of Group Membership . . . 187
6.4.2 Patterns for Group Orchestration . . . 190
6.5 Failure Handling . . . 195
6.5.1 Automatic Failure Handling . . . 196
6.5.2 Patterns for Failure Handling . . . 199
6.6 Conclusion . . . 201

7 NOW Up and Running **203**
7.1 Application stressing Service Orchestration . . . 204
7.2 Application stressing Group Orchestration . . . 204
7.3 Application Stressing Failure Handling . . . 207
7.4 Scalability Results . . . 212
7.4.1 Language Scalability . . . 213
7.4.2 Scalability of Language with Failure Detection . . . 217
7.4.3 Scalability of Example Scenarios . . . 220

7.4.4 Discussion 223
7.5 Conclusion 224

8 Conclusion **225**
8.1 Summary and Contributions 226
8.2 Discussion and Future Work 229

Bibliography **233**

Index **247**

LIST OF FIGURES

2.1 Example "order fulfilment" workflow [Obj12] (expressed using BPMN [Obj11]). . . . 16

3.1 AMBIENTTALK actor. . . . 38
3.2 Concurrency model of AMBIENTTALK. . . . 39
3.3 AMBIENTTALK: mirrors. . . . 46
3.4 AMBIENTTALK: mirages. . . . 49

4.1 Lifecycle of an activity. . . . 61
4.2 Data flow: data environment passed simultaneously with incoming and outgoing control flow edges. . . . 63
4.3 Example sequence diagram of starting an activity. . . . 64
4.4 Merging strategies for synchronisation patterns. . . . 65
4.5 Basic control flow patterns: Parallel Split with a Sequence in its first branch. . . . 70
4.6 Sequence diagram of the execution of the workflow depicted in Figure 4.5. . . . 71
4.7 Example of a multiple instances pattern. . . . 72
4.8 Basic control flow patterns: Parallel Split followed by multiple synchronisation patterns. . . . 73
4.9 Sequence diagram of the execution of the workflow depicted in Figure 4.8. . . . 75
4.10 Trigger patterns: Sequence pattern using a Persistent Trigger pattern. . . . 76
4.11 Sequence diagram of the execution of the workflow depicted in Figure 4.10. . . . 77
4.12 Federated fact spaces of colocated devices. . . . 82
4.13 Example of a Group pattern. . . . 83
4.14 Sequence diagram of the execution of the workflow depicted in Figure 4.13. . . . 83

4.15 Restricting the members of the group by using a filter. 84
4.16 Synchronisation: the Barrier pattern. 86
4.17 Synchronisation: execution of a Cancelling Barrier pattern. 87
4.18 Synchronisation: a Group Join pattern to terminate the group. 88
4.19 Synchronisation: a Synchronised Task pattern executing a task once for all instances. 91
4.20 Lifecycle of an activity (version II). 93
4.21 Failure pattern. 94
4.22 Failure pattern: overriding default failure handling strategies with more specific compensating actions. 95
4.23 Nesting of Failure patterns to acquire more accurate failure handling strategies. 97
4.24 Different compositions of failure handling for group orchestration. . . 98
4.25 Failure detection: normal failures versus participant failures. 100
4.26 Failure Handling for Group Orchestration - Composition 1. 102
4.27 Failure Handling for Group Orchestration - Composition 2. 103
4.28 Failure Handling for Group Orchestration - Composition 3. 104
4.29 Failure handling at the level of a work item [RtEv05a]. 105

5.1 Composition of synchronisation patterns: one Synchronization pattern. 121
5.2 Composition of synchronisation patterns: several Synchronization patterns. 122
5.3 Workflow implementing the iMPASSE application. 123
5.4 Federated fact spaces of colocated devices. 129
5.5 Workflow implementing the SURA application. 132
5.6 Compensating actions specified for different kind of failures. 137
5.7 Workflow implementing the SWOOP application. 140

6.1 Prioritise merging strategy for synchronisation patterns. 172
6.2 Object diagram for service orchestration patterns. 174
6.3 Object diagram for group orchestration patterns. 190
6.4 Application-specific failure handling by nesting Failure patterns to override (default) compensations. 195

7.1 Lines of code for the code complexity of the SWOOP application in NOW and AMBIENTTALK. 211
7.2 Measurement of overhead introduced by patterns. Results for sequence pattern (a) and parallel split pattern (b). 215

7.3 Measurement of overhead introduced by failure detection. Results for sequence pattern (a) and parallel split pattern (b). 219
7.4 Measuring the addition of workflow patterns compared to plain AMBIENTTALK. Results are shown for a multiple instances pattern wrapping the workflow implementing the iMPASSE application. 221
7.5 Measuring the addition of workflow patterns compared to plain AMBIENTTALK. Results are shown for a multiple instances pattern wrapping the workflow implementing the SURA application. 222
7.6 Measuring the addition of workflow patterns compared to plain AMBIENTTALK. Results are shown for a multiple instances pattern wrapping the workflow implementing the SWOOP application. 223

LIST OF TABLES

4.1 Control Flow Patterns 68
4.2 Effect of a compensating action depending on the composition that is used for failure handling for group orchestration. 99

5.1 Control Flow Patterns supported by NOW. 119
5.2 Failure handling for group orchestration in NOW. 138
5.3 Survey of related work. 164

1 INTRODUCTION

1.1 Research Context

Nowadays we are surrounded by all kinds of computing devices, ranging from smartphones to digital watches, and in the near future even digital glasses. Weiser's vision [Wei91, Wei93, Wei99] is becoming more and more reality seeing that everyday objects, such as refrigerators and product scanners, are more and more integrated with computers and are able to provide digital information to its users. Consider for instance electronic dresses that give light depending on the sound level, scarfs that calm autistic kids when they are overstimulated, etc. The small connected computing units, dubbed *ambient devices*, have characteristics which are significantly different from the conventional devices. One of these characteristics is that connections cannot be assumed stable, this stems from the fact that these devices are mobile and can leave a certain area at any moment in time. The integration of these ambient devices into everyday items is called *ubiquitous computing* [Wei91].

With the use of a variety of standard technologies like WiFi, Bluetooth and NFC we can connect these miniaturised devices in a *mobile ad hoc network* (MANET) [MRV98]. Mobile ad hoc networks spontaneously emerge by the colocation of devices with wireless networking capabilities, without the need of an additional infrastructure, and, furthermore, no substantial configuration is required. These characteristics allow software for ubiquitous computing to interact spontaneously and unobtrusively.

Nomadic networks are a special case of mobile ad hoc networks. Nomadic networks consist of a set of mobile devices that try to maintain a connection with the fixed

infrastructure, that serves as a central backbone [MCE02]. Nomadic networks are omnipresent, for instance, in shopping malls, airports, train stations, universities, hospitals, hotels, movie theatres, football stadia, festival areas, opera buildings, etc. Concretely, the hospital of the Vrije Universiteit Brussel ("Universitair Ziekenhuis Brussel") has a nomadic network. The server of the hospital runs several applications that interact with services provided by computers/devices that are connected via (un)reliable communication links. There are services that are provided by servers in the hospital itself, such as an application running on the computers at the check-in desk. Other services are running on mobile devices, consider for instance the calendar application running on the mobile phone of a nurse.

Since nomadic networks are omnipresent, an abundance of interesting applications can be supported. However, the development of such applications is not straightforward as special properties of the communication with mobile devices, such as connection volatility, have to be considered. Application developers can rely on software for ubiquitous computing to develop applications for nomadic networks. Ubiquitous computing software must provide the necessary hooks to deal with the high dynamicity of the network. The software must not only deal with intermittent network connections, it must also allow applications to react upon events, such as other users and sensors in the neighbourhood. Several models for programming these types of distributed applications have been developed. The model we use as a foundation for our work is the ambient-oriented programming paradigm [DVM+06]. Ambient-oriented programming is a paradigm specifically sculpted towards applications running on mobile devices since it takes characteristics such as connection volatility and zero infrastructure into account.

Although the ambient-oriented programming paradigm is suited for the development of applications in mobile ad hoc networks, the cooperation between the different entities in the environment is programmed ad hoc. In order to facilitate the development of complex nomadic applications, there is a need for composition techniques and high-level language abstractions that allow the specification of the interactions between the entities in the network. Therefore, the principles introduced by workflow languages can be used, since they provide patterns that specify the composition of services and the interactions between these services at a high level. This way, the control flow of the application is not interwoven with the fine-grained application logic that is provided by the services.

1.2 Research Vision

We envision a new type of rich applications, which we call *nomadic applications* from now on, that are deployed and executed on the fixed infrastructure while communicating with the mobile devices in the network. Therefore, the backbone of the nomadic network runs software that allows the discovery of nearby mobile devices. These mobile devices are equipped with software such that their applications, called *services*, can be published on the network. In order to allow discovery of nearby services, a WiFi access point is required.

During the execution of a nomadic application, services can be invoked to perform a certain task. We distinguish two categories of services that can be used by nomadic applications, namely stationary services and mobile services.

Stationary services are services that reside on a device that is part of the fixed infrastructure of the network. An example of a stationary service is the application running on a computer at the information desk at the airport that is responsible for announcements.

The second category of services, the *mobile services*, are services residing on mobile devices. For these category of services we make the distinction between registered services and user services. *Registered services* are services that are part of the predefined infrastructure of the nomadic network, whereas *user services* are services that are not known a priori. An example of a registered service is the calendar application of a specific airline company that is running on the mobile phone of a pilot of that company. The infrastructure of the airport knows this service beforehand, since personnel at the airport is registered, and therefore, the backbone has knowledge about the services associated with applications running on the mobile device of a personnel member. Examples of user services are the applications running on the smartphones of passengers. User services also grasp the services that are made available for download by the nomadic applications, running on the fixed infrastructure of the network. Since passengers and visitors are not registered, the backbone at the airport does not have knowledge about all applications running on their devices. Interaction with these users is achieved by discovering the services that are published on the network. Consider applications that can signal alerts for certain events. The nomadic application can invoke this service when necessary.

Both stationary and registered mobile services are services that are part of the infrastructure and are responsible for the inner workings of the nomadic application. Nomadic applications depend more on those types of services because they are in charge of the more “predictable” processing. The failure of one of those services, like the unavailability of a service, has an influence on the entire application. Consider the nomadic application that contacts a tourist guide, before notifying the passengers of

his/her guidance help. When a disconnection occurs during communication with the (registered) service, running on the guide's mobile device, the nomadic application's execution is affected. In this case, some compensating action is required, before contacting the passengers.

Mobile services, on the other hand, actually drive the need for processing. Think about an application that is responsible for reminding a passenger about the continuity of his/her flight. The discovery of such a new (user) mobile service (i.e., application running on a passenger's mobile device), can trigger the execution of an application running on the airport's backbone. Mobile services are the services that steer these applications, and the unavailability of such a service does not affect the correct working of the application nearly as much.

Consider for instance the following example application where a tourist guide is contacted to inform him/her that he/she can start gathering all passengers that will join him/her to Italy. When the tourist guide, which is considered a registered service, is unavailable, the nomadic application should perform the necessary actions to ensure that the application's task can be executed. The applications running on the mobile phones of the passengers in the example are considered user mobile services. When one or more passengers are unavailable, the entire application can continue its execution (i.e., the other passengers can be contacted), although it might also be useful to provide compensations in case a user mobile service is unavailable.

Nomadic applications are applications that run on a stationary backbone and execute tasks by invoking both stationary and mobile services. These nomadic applications have some special characteristics:

- Nomadic applications control the different types of services in the network by invoking them when necessary. The emphasis of a nomadic application is the orchestration of the different services in the nomadic network. The orchestration of these services describe the control flow of a nomadic application separately from the fine-grained application logic, which is implemented by the services.

- Because mobile services (registered services and user services) are residing on mobile devices, it is likely that communication failures occur. For example, such a service can become unavailable when the mobile device goes out of communication range. Therefore, nomadic applications should take into account that nomadic networks are dominated by volatile connections and provide means to compensate for such errors and failures.

- Nomadic applications make use of user services, which are unknown to the infrastructure before they become connected. These applications need to discover the services in the network that can execute particular tasks. Moreover, nomadic

applications need to react upon changes in the network topology. The discovery of a mobile service in the network is an event that can cause the nomadic application to behave differently. The number of services a nomadic application wants to invoke can vary each time the same application is executed. For example, the application that addresses all passengers of a particular flight is most likely to communicate with a different number of services each time the application is executed.

Executing a nomadic application results in the invocation of services in the nomadic network in order to perform particular tasks. Nomadic applications are applications that orchestrate the different types of services residing in the network.

1.3 Research Objective

In this dissertation we investigate how the development of the nomadic applications we described in the previous section can be facilitated. We propose a programming model that attends to the characteristics of nomadic applications. These special characteristics, such as the fact that not all services are known beforehand, and that failures can occur due to intermittent network connections, impose several criteria the programming model must adhere to. These characteristics can be divided in three categories, namely service orchestration, group orchestration, and failure handling.

- **Service orchestration**: Nomadic applications are applications running on the fixed infrastructure of the nomadic network that communicate with services (possibly) running on mobile devices. These nomadic applications interact with three types of services, amongst which user services that are not known beforehand.

  During the execution of a nomadic application, services can go out of communication range at any moment in time. Nomadic applications handle intermittent disconnections by allowing services to continue executing while not necessarily connected to the backbone.

  Moreover, nomadic applications cannot remain unresponsive for a significant amount of time, such as when a service is temporarily disconnected.

  Nomadic applications must specify the composition of tasks (i.e., service invocations) at a high-level and ensure that the control flow of the application is not interwoven with the fine-grained application logic, which is provided by the services.

- **Group orchestration**: The increasing popularity of mobile devices fosters the omnipresence of services in mobile environments, such as MANETs and nomadic networks. Nomadic applications have to manage a dynamically changing group of services, such as the services running on the mobile devices of an aircrew. Managing this logical group of services is achieved by orchestrating the execution of a particular process for all group members. For example, ask all passengers to fill in a survey, gather their answers, etc.

  Since not all services are necessarily known a priori, an extensional description of the services is not always possible. Therefore, nomadic applications must intensionally describe the members of a group. Possible intensional descriptions are "all passengers of a certain flight who have children younger than 12 years, and are seated in rows 15 to 30", "the personnel of Brussels Airlines", etc.

  Additionally, the number of services a group constitutes of, is not known beforehand and can fluctuate over time.

  Moreover, during the orchestration of a group of services, it must be possible to redefine the group members. For instance, restricting the members of the group to only a subset of the services that originally satisfied the group's description by adding an extra constraint.

  Furthermore, the execution of the process for each group members must be controlled, such that the identity of the service that is executing a particular task can be managed. Moreover, the number of times a specific task is executed must be controllable. Therefore, there is a need for synchronisation mechanisms that let processes wait, redirect, or abort in order to guarantee a successful group orchestration.

- **Failure Handling**: In a nomadic network, the challenge is to make the large heterogeneity of services co-operate and deal with transient and permanent failures. To communicate in a fault tolerant manner, all common failures such as disconnections, timeouts, unavailability of services, and service errors need to be considered and handled.

  There is a need for automatic failure handling, such that a nomadic application can recover from (network) failures. It must also be possible to specify application-specific compensating actions for certain types of failure.

  In mobile environments where users and services enter and leave at will, it must also be possible to orchestrate a group of services by managing the effects of failures on the group. Nomadic applications must react upon a failure during the execution of the process of a single group member, and must also manage

failures at the level of the entire group. Failure handling for groups must allow compensations to affect the execution of a group in a way that exceeds the execution of an individual group member.

1.4 Research Methodology

The characteristics of nomadic applications, which we described in the previous section, impose criteria a programming model must adhere to. Because we are targeting nomadic networks, some of these criteria originate from the ambient-oriented programming paradigm. Other criteria either stem from the fact that nomadic applications are centred around the notion of the orchestration of services in the network, or are a combination of both.

The work we present in this dissertation fits in the context of two paradigms, namely the workflow paradigm and the ambient-oriented programming paradigm, which we both discuss in Chapter 2. This research is centred around the development of nomadic applications, which are applications running on the fixed infrastructure of a nomadic network while orchestrating services residing in the network.

Since nomadic applications are a special type of distributed applications, we are inspired by technologies used to develop distributed applications that function in highly dynamic networks. Existing middleware [MCE02] and programming languages, such as AMBIENTTALK [DVM+06], are specifically tailored towards the development of applications for dynamic network environments. The cooperation between the different entities in the environment is, however, programmed ad hoc. In order to ease the development of these applications, there is a need for composition techniques and high-level patterns that allow the specification of the interactions between the different entities in the network. In stable networks, complex distributed applications can be developed using technologies such as service-oriented computing [PG03]. The composition of services and the interactions between these services can be achieved using the principles of workflow languages. Workflow models and languages are already successfully received in research domains like business processes and traditional distributed environments, because they support the composition of services, the description of the control flow, and parallel execution.

Although existing workflow languages, such as WS-BPEL [JE+07] and YAWL [vdAtH05], are suited for distributed environments, they are not fully equipped to describe nomadic applications. First of all, these languages interact with services that have a fixed location and are known a priori, for instance, through a URL. Nomadic applications must be able to interact with three types of services, amongst which user services that are not known beforehand. Secondly, interactions between the different

services are typically synchronous. Because of the high dynamicity of the nomadic network, not having access to a service must be considered the default.

This dissertation is focussed on language design. We identify the criteria a programming model must adhere to in order to facilitate the development of nomadic applications. We define language abstractions that comply with these criteria, and implement a proof-of-concept workflow language supporting the proposed language abstractions. We validate these language abstractions by employing them to implement representative nomadic applications.

1.5 Contributions

The thesis we set forth in this dissertation is

> By introducing language abstractions for service orchestration, group orchestration, and failure handling, the development of nomadic applications is facilitated.

We summarise the main contributions of this dissertation:

- **Identification of Criteria for Orchestration in Nomadic Networks**: The research described in this dissertation is founded by two paradigms, namely the workflow paradigm and the ambient-oriented programming paradigm. Based upon the principles and characteristics described by languages implementing these paradigms and the characteristics of nomadic applications, we distill criteria we argue are necessary for the orchestration of services in a nomadic network.

- **Definition of Abstractions for Developing Nomadic Applications**: After having identified criteria necessary for orchestration in nomadic networks, we define a set of patterns that enable orchestration in nomadic networks.
  - We first revisit existing control flow patterns [RtHvdAM06] in the context of nomadic networks;
  - We present novel patterns that allow the orchestration of a dynamically changing group of services; and
  - We present patterns that allow the detection and handling of failure events through the execution of compensating actions.

- **Implementation of Proof-of-Concept Workflow Language**: The postulated patterns for orchestration are implemented in a proof-of-concept workflow language (NOW). This workflow language for nomadic networks is built as a li-

brary on top of the ambient-oriented programming language AMBIENTTALK/2. NOW incorporates the three sets of patterns our programming language consists of, namely a subset of existing control flow patterns, patterns for group orchestration, and patterns for failure handling.

- **Validation of Nomadic Applications developed using NOW**: We present the implementation of three example scenarios that are revolved around one specific set of criteria. The first example application pays attention to the orchestration of services, and the composition of control flow patterns. The second application is centred around the necessity for group abstractions that allow the orchestration of a set of services that form a logical group. The last application addresses the notion of detecting and handling failures, both for service orchestration and group orchestration.

Supporting Publications

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

- **NOW: A Workflow Language for Orchestration in Nomadic Networks** [PVJ10]
  Eline Philips, Ragnhild Van Der Straeten, Viviane Jonckers
  12th International Conference on Coordination Models and Languages (COORDINATION 2010)
  This paper proposes a nomadic workflow language built on top of the ambient-oriented programming language AMBIENTTALK/2. The proposed language, called NOW, provides high-level workflow abstractions for control flow and supports network and service failure detection and handling through compensating actions. The paper also introduces the variable binding mechanism employed by NOW, which enables dynamic data flow between services in a nomadic network.

- **NOW: Orchestrating Services in a Nomadic Network using a dedicated Workflow Language** [PVJ13]
  Eline Philips, Ragnhild Van Der Straeten, Viviane Jonckers
  Science of Computer Programming, 2013
  This paper extends the aforementioned paper with implementation details, and by providing a comparison of the code complexity of the implementations of two example applications in a dedicated language for mobile networks and in our proposed workflow language.

- **Group Orchestration in a Mobile Environment** [PVVJ12]
  Eline Philips, Jorge Vallejos, Ragnhild Van Der Straeten, Viviane Jonckers
  14th International Conference on Coordination Models and Languages (COORDINATION 2012)
  This paper presents high-level abstractions for group orchestration in a nomadic network as a new set of workflow patterns. In this paper, we explain how these patterns are integrated in the existing workflow language NOW. That workflow language handles network and service failures at the core of the language. By extending this fault tolerance to the new group abstractions, we illustrate how to conduct these in a reliable way.

1.6 Dissertation Roadmap

Chapter 2: Orchestration in Nomadic Networks describes the context of this dissertation, which is supported by the workflow paradigm and the ambient-oriented programming paradigm. Both paradigms are introduced in this chapter, and the necessary terminology is defined. Thereafter, we postulate criteria that are necessary for service orchestration in a nomadic network. These criteria are distilled from three example applications, which are used throughout this dissertation.

Chapter 3: Ambient-Oriented Programming in AMBIENTTALK presents the programming language AMBIENTTALK/2. We describe the language features that are necessary to understand the technical contribution of this dissertation, namely the implementation of the nomadic workflow language NOW. A dedicated chapter on AMBIENTTALK/2 is required because this nomadic workflow language is built as an abstraction layer on top of this ambient-oriented programming language. Moreover, AMBIENTTALK/2 is considered as related work of the work presented in this dissertation, because this ambient-oriented programming language is targeted towards mobile ad hoc networks. The nomadic networks we target in this dissertation can be regarded as a special case of these mobile ad hoc networks. Lastly, part of the motivation for the nomadic workflow language NOW is rooted in the difference between mobile ad hoc networks and nomadic networks. In this chapter we also discuss AMBIENTTALK/2's limitations, concerning service orchestration.

Chapter 4: Patterns for Orchestration in Nomadic Networks puts forward a set of patterns that allows the orchestration of services in a nomadic network. First, we discuss an interpretation of activities suitable for nomadic networks. Afterwards, we present a data flow mechanism, which allows data to be passed between activities

in a workflow. Subsequently, we discuss the necessity of patterns to allow service orchestration in a nomadic network. We start by presenting a set of existing control flow patterns that are widely used for orchestration. Thereafter, we introduce a set of novel patterns that are specifically sculpted for nomadic networks: We introduce patterns for group orchestration, and patterns that allow (automatic) failure handling.

Chapter 5: A Workflow Language for Orchestration in Nomadic Networks introduces a workflow language that has support for the patterns we proposed earlier. This workflow language, called NOW, is targeted towards nomadic networks and is built as an abstraction layer on top of the ambient-oriented programming language AMBIENTTALK/2, presented in Chapter 3. In this chapter we first motivate why we chose AMBIENTTALK/2 as the platform to build this workflow language on. Before presenting the patterns that NOW supports, we describe how services can be implemented in NOW and explain the data flow mechanism the workflow language employs. We also present NOW from the point of view of the application developer: We introduce the implementation of three example applications, and, illustrate how the proposed workflow patterns can be used. Given the criteria identified in Chapter 2, we survey a number of workflow languages and coordination languages that offer interesting solutions for orchestrating services, or coordination in mobile networks, or a combination of both. We evaluate these languages and discuss how they fail to meet all our postulated criteria for orchestration in nomadic networks.

Chapter 6: Implementing NOW describes the patterns for service orchestration from the implementor's point of view. In this chapter we describe the technicalities of the workflow language and point out how the language is built on top of AMBIENTTALK/2. We also discuss how new patterns can be added to the nomadic workflow language.

Chapter 7: NOW Up and Running discusses the comparison of the code complexity of the implementation of three example applications in NOW and AMBIENTTALK/2. From these experiments, we can conclude that the implementation in NOW is shorter and contains less nested code. We also present the results of experiments we conducted to measure the overhead of introducing patterns on top of the ambient-oriented programming language.

Chapter 8: Conclusions summarises the contributions made in this dissertation. In this chapter we also discuss the limitations of our work and outline some possible future work directions.

2

ORCHESTRATION IN NOMADIC NETWORKS

In this chapter we sketch the context of this dissertation, which is built upon two research pillars, namely the workflow paradigm and the ambient-oriented programming paradigm. We first discuss how orchestration of services can be achieved by using principles from the workflow paradigm. Subsequently, we describe the context of our work, namely nomadic networks, and describe how the criteria put forward by the ambient-oriented programming paradigm also hold for these types of networks. Afterwards we distill criteria from both paradigms and postulate criteria that are necessary for orchestration in nomadic networks.

2.1 Workflows and Orchestration

The term *orchestration* is used to designate the automated coordination and management of services, middleware and computer systems. Orchestration is employed in a wide range of domains, like service-oriented architectures, web services, and virtualisation.

2.1.1 Service Orchestration

A service-oriented architecture (SOA) enables the integration of several heterogeneous applications in a network [PG03]. These applications provide their functionality as services to other applications. Service-oriented architectures organise and utilise services in a way that promotes the visibility of services, the interaction with services, and the effects of services [MLM+06]. The services within a service-oriented architecture are loosely coupled, and the interaction between services is typically described using explicit service orchestrations.

Web services are commonly used to provide the functionality of the applications in a SOA to other applications and clients [ACKM04]. A web service is defined as "*a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [CCMW01]). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages [NMM+03], typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*" [HB04].

Communication in heterogeneous environments like the internet is made possible as these web services use open standards, platforms and software-independent technologies. These web service technologies ensure that services and clients are not tightly coupled, and hence enable a volatile integration. The fact that web services allow the interaction between heterogeneous software systems in a network leads to the development of programs that compose several web services to provide some new functionality. This composition of web services is called *web service orchestration*. Service orchestration is defined by Peltz [Pel03] as "*a business process that interacts with both internal and external web services*".

The orchestration of the different (web) services of a SOA can be specified using a (web) service orchestration language, such as WS-BPEL [JE+07], which is built upon the principles of workflow languages. Workflow languages allow the specification of the different activities that must be executed, and have a focus on control flow, data flow and exception handling. In the remainder of this section we describe workflow languages and introduce the terminology related to the workflow paradigm.

2.1.2 Workflows

Workflow management was introduced to model and control the execution of business processes, where the workflow describes those aspects of a process that are relevant to controlling and coordinating the execution of its tasks [GHS95, WV98]. According to the Workflow Management Coalition a *workflow* is "*the computerised facilitation*

or automation of a business process, in whole or part" [Hol95]. A *workflow management system* is defined as "*a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic*" [Hol95].

We introduce an "order fulfilment" process [Obj12], which we use to explain the most relevant concepts of workflows. The order fulfilment process starts when an order message has been received. When the process is started, it first verifies whether the ordered article is available in stock or not. When the article is available, it is shipped towards the customer and a financial settlement is performed. In case the ordered article cannot be found in stock, the customer must be informed either that the delivery will be later, or that the article cannot be delivered. When the article is undeliverable, it must also be removed from the catalogue. This process is modelled using the Business Process Model and Notation (BPMN), as can be seen in Figure 2.1.

Workflow languages aim at capturing workflow-relevant information of application processes with the objective at their controlled execution by a workflow management system [GHS95]. Workflow languages are used to describe the particular tasks (called *activities*) that need to be executed and also specify the order in which these tasks need to be performed (i.e., sequential, in parallel, etc.). An activity is the description of a piece of work that needs to be performed. Activities are classified as either an atomic task or as a sub process. An *atomic task* represents a single unit of work that cannot be broken down to a smaller task. A *sub process*, on the other hand, has its own start and end event, and consist of several task that are composed into a larger activity. In BPMN, an activity is represented with a rounded-corner rectangle and describes the kind of work which must be done. In the example workflow depicted in Figure 2.1, "Check availability" and "Inform customer" are examples of activities.

The specification of the order in which activities need to be executed is referred to as the *control flow perspective* of the workflow. van der Aalst [vtea12] defines five kinds of perspectives that need to be supported by a workflow management system. Besides the control flow perspective, a workflow language should also support the data flow, resource, exception handling, and presentation perspective. The Workflow Pattern Initiative [vtea12] provides a description of several *patterns* sculpted for each of those perspectives and examines existing workflow languages with respect to the proposed pattern abstractions.

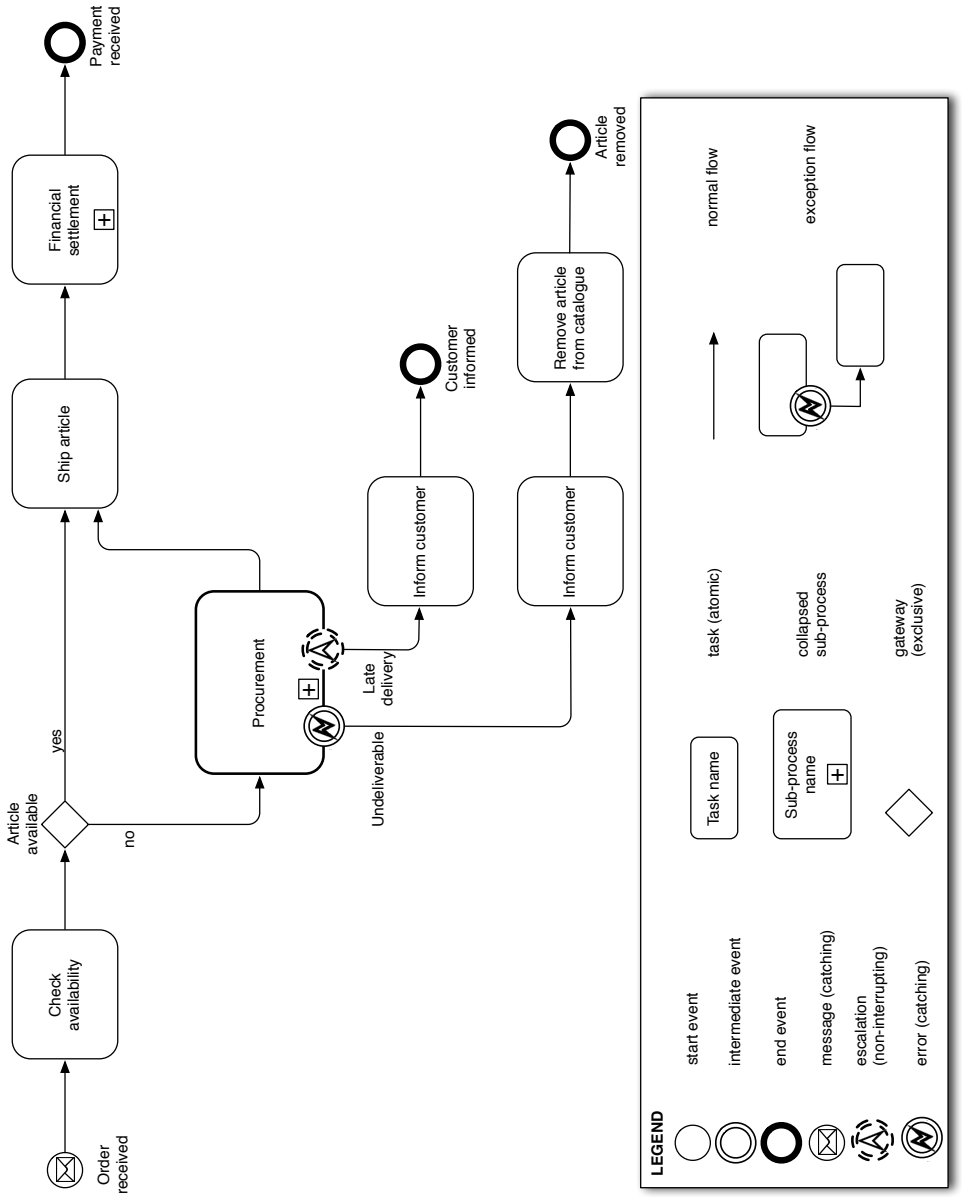


Figure 2.1: Example “order fulfilment” workflow [Obj12] (expressed using BPMN [Obj11]).

We now explain the difference between the five proposed perspectives of the Workflow Pattern Initiative [vtea12].

- **control flow perspective** van der Aalst defines 43 control flow patterns that specify the control flow of the application, i.e., how different tasks constituting the process must be linked together. The proposed patterns are divided into several categories, ranging from basic control flow patterns (like sequence, synchronization) to more advanced patterns such as multiple instances and iteration patterns.

  In the "order fulfilment" process we can distinguish the following patterns:

  – A sequence pattern is used to ensure that a task in the process is only enabled after the completion of the preceding task in the process. This pattern is depicted using arrows between the activities in the workflow, denoting the order in which these activities must be executed. In our workflow example, the "Ship article" activity precedes the "Financial settlement" activity.

  – An exclusive choice pattern models the divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is passed to only one of these outgoing branches, namely the branch that is selected using a decision mechanism. This pattern is modelled using a gateway, which is represented with a diamond shape and determines forking and merging of paths, depending on the expressed conditions. In the example workflow depicted in Figure 2.1, an exclusive choice pattern is used to ensure that the necessary tasks are executed, depending on whether the article is available, or not.

- **resource perspective** This perspective focusses on resources, which are defined as "entities that can perform a task" [RvtE05]. The patterns defined for the resource perspective are, for instance, concerned with the allocation of resources for the tasks of a process.

  The association of a task or sub process to a specific resource can be modelled in BPMN using swim lanes. Just like data flow, this perspective is not explicitly depicted in Figure 2.1.

- **data flow perspective** For the third perspective, Russell [RtEv05a] specifies how information is passed, variables are scoped, etc. More specifically, patterns concerning *data visibility*, *data interaction*, *data transfer*, and *data-based routing* are defined.

  Although data flow is not explicitly depicted in Figure 2.1, data flow can be modelled in BPMN as property attributes of a single task or a sub process.

- **exception handling perspective** This perspective explores the different origins of exceptions and the possible actions to handle occurred exceptions. Russell [Rvt06] outlines four exception types, namely *deadline expiry*, *resource unavailability*, *external trigger*, and *constraint violation*. For such exceptions three recovery strategies are defined: *no action*, *rollback*, and *compensate*.

  In the example workflow, exceptional flow occurs outside the normal flow of the process and is based upon an intermediate event attached to the boundary of an activity. The arrow between the activities "Procurement" and "Inform customer" is an example of such an exceptional flow where a compensation is specified for an exception type.

- **presentation perspective** The last perspective is concerned with both the structure and representation of the process model. First of all, patterns are introduced that reduce the complexity of the process model at the level of the abstract syntax [LWM+11]. A second collection of patterns is proposed in order to change the visual representation of a process model by modifications to its concrete syntax [LtW+11].

We evaluate these perspectives with respect to orchestration in nomadic networks. The control flow perspective decouples the application logic (i.e., the different tasks that must be executed) from the control flow of the application. This way, the control flow of the application is the centre of attention: not only the knowledge of the different tasks that must be performed, but also the order in which these tasks must be executed. Furthermore, the information that is passed between the tasks of the application is controllable.

Moreover, identifying smaller units of work facilitates the recovery in case a failure occurs, by compensating the effects caused by these smaller units. In order to allow orchestration of services in a nomadic network, the exception handling perspective is also relevant.

In contrast to the other perspectives, the resource perspective and the presentation perspective is less relevant for the orchestration in nomadic networks. This dissertation is mainly concentrated on the definition of abstractions that allow the orchestration of services in a nomadic network. Since no visual representation is presented in this dissertation, this perspective is, at the moment, less important.

2.1.3 Terminology

The workflow community has made efforts to standardise the terminology in order to prevent several terms being used for various concepts. In this dissertation, we mainly use the terminology that was standardised by the Workflow Management Coalition [The99]. In this section we list the terms most relevant to this dissertation.

- **business process** is defined as "*a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships.*";
- **workflow** is "*the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*";
- **workflow management system** is "*a system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications.*";
- **process** is defined as "*a formalised view of a business process, represented as a coordinated (parallel and/or serial) set of process activities that are connected in order to achieve a common goal.*" We use the term **application** in the remainder of this dissertation to refer to a process as it is defined here;
- **sub process** is "*a process that is enacted or called from another (initiating) process (or sub process), and which forms part of the overall (initiating) process.*" In the remainder of this dissertation we use the term **sub workflow** to refer to a sub process;
- **activity**, also known as **task**, is defined as "*a description of a piece of work that forms one logical step within a process*";
- **instance** is "*the representation of a single enactment of a process* (process instance)*, or activity within a process* (activity instance)*, including its associated data. Each instance represents a separate thread of execution of the process or activity, which may be controlled independently and will have its own internal state and externally visible identity, which may be used as a handle, for example, to record or retrieve audit data relating to the individual enactment.*";

- **workflow participant** (also known as **user**) is defined as "*a resource which performs the work represented by a workflow activity instance*";
- **process definition** is specified as "*the representation of a business process in a form which supports automated manipulation, such as modelling, or enactment by a workflow management system*";
- **workflow engine** is "*a software service that provides the run time execution environment for a process instance*".

2.2 Ambient-Oriented Programming

The research conducted in this dissertation is centred around the notion of a nomadic network. Programming applications for nomadic networks is not trivial, as certain hardware characteristics, like volatile connections, must be taken into account. Ambient-oriented programming (AmOP) is a paradigm that takes these hardware characteristics into consideration, making it suitable for programming peer-to-peer mobile applications. Peer-to-peer mobile applications are applications that enable point-to-point communication without relying on a third party. Note however that the AmOP paradigm does not require all applications to be peer-to-peer, it is also allowed to structure the applications using the client-server pattern.

The ambient-oriented programming paradigm led to the development of programming languages (like AMBIENTTALK/2 which we present in Chapter 3) that incorporate potential network failures in the heart of their computational model, enabling the development of applications in mobile ad hoc networks [DVM+06]. Two hardware characteristics are inherent to mobile ad hoc networks [VC08]:

- **volatile connections** (also known as *intermittent connections* [MCE02]) Due to the limited communication range of the mobile devices in a mobile ad hoc network, *partial failures* can occur. As devices can go out of range at any moment in time, disconnections should be considered the rule rather than the exception. Often, applications do not want to block communication when such a partial failure occurs, but want to continue after the connection is re-established.
- **zero infrastructure** In mobile ad hoc networks there is often no infrastructure available. In this type of distributed system, services must be dynamically discovered in their environment. Hence, the location of a device has a significant role.

These hardware phenomena were used by Van Cutsem [VC08] to define criteria for coordination abstractions in a mobile ad hoc networks. Because nomadic networks are a special case of the more dynamic mobile ad hoc network, some of these criteria are also relevant for our research. The criteria postulated by Van Cutsem can be divided into three categories:

1. **Decentralised discovery (criterion 1)**
   The first criterion tackles the fact that for mobile ad hoc networks, a central lookup service is too restricted. First of all, a lookup service (or name server) requires some form of infrastructure, which is often not available in a mobile ad hoc network. Moreover, the network topology of a mobile ad hoc network changes frequently as devices go in and out of range. This change is not reflected by a lookup service, which does not notify clients of any changes to a registered service. Decentralised discovery is defined as "*processes require a decentralised service discovery protocol that enables them to* autonomously *act upon the (un)availability of nearby services*".

2. **Decoupled communication**
   - **Decoupling in time (criterion 2)**: Van Cutsem argues that processes should be able to express communication independently of their connectivity. The *decoupling in time* criteria states that "*communicating processes do not necessarily need to be online at the same time*".
   - **Decoupling in space (criterion 3)**: A second communication-related criterion is called *decoupling in space*. This criterion is concerned with the fact that "*communicating processes do not need to know each other beforehand*". In mobile ad hoc networks it is not opportune to have knowledge about the exact address or location of all processes with which communication will take place.
   - **Synchronisation decoupling (criterion 4)**: In mobile ad hoc networks processes should not be blocked when information is requested from other services. By letting processes be responsive to other events while waiting for such information, an application does not remain unresponsive for a considerable time period. This criterion is known as *synchronisation decoupling* which indicates that "*the control flow of communicating processes is not blocked upon sending or receiving*".
   - **Arity decoupling (criterion 5)**: A last communication-related criterion is concerned with the number of processes communicated with. Van Cutsem argues that in mobile ad hoc networks "*processes do not necessarily need to know the total number of processes communicated with*".

3. **Connection-independent failure handling (criterion 6)**
   In mobile ad hoc networks, which are liable to volatile connections, transient failures happen frequently. As we already mentioned, it is often useful to resume computation after the connection is reestablished. Therefore, Van Cutsem [VC08] argues that network failures should be considered as an ordinary event instead of a failure event. That way, transient failures can be abstracted, and failure handling becomes more robust, because failure handling code can equally be triggered even if there is no physical network failure. However, Van Cutsem declares that abstractions to handle connection-dependent failures are opportune, as the information about the underlying network connection is often useful. The last criterion for coordination in mobile ad hoc networks is therefore formulated as "*processes should be able to perform failure handling independent of any network failures*".

Before we adapt these criteria for nomadic networks (see Section 2.3.3), we list the terms that are most relevant to this dissertation.

2.2.1 Terminology

We now recapitulate the definitions of the terms that are used throughout the remainder of this dissertation.

- **mobile ad hoc network** (MANET): "*a transitory association of mobile nodes which do not depend upon any fixed support infrastructure*" [MRV98];
- **nomadic network**: "*a network consisting of both mobile devices and fixed support infrastructure*";
- **time decoupling**: "*communicating processes do not necessarily need to be online at the same time*";
- **space decoupling**: "*communicating processes do not necessarily need to know each other beforehand*";
- **synchronisation decoupling**: "*the control flow of communicating processes is not blocked upon sending or receiving*";
- **arity decoupling**: "*processes do not necessarily need to know the total number of processes communicated with*".

2.3 Orchestration in Nomadic Networks

Based upon the characteristics of programming languages implementing either the workflow paradigm or the ambient-oriented programming paradigm, we postulate criteria for orchestration in nomadic networks. Before describing these criteria, we give three examples of nomadic applications that are used throughout the remainder of this dissertation, and we present our definition for orchestration in a nomadic network.

2.3.1 Scenarios of Nomadic Applications

In Section 1.2 we described the characteristics of nomadic applications. These applications are deployed and executed on the fixed infrastructure of a nomadic network and communicate with services that are residing in the network. These services can be either located on stationary or mobile devices. Recall that services can be either known beforehand, when they are part of the infrastructure, or can be unknown a priori.

We present three examples of such nomadic applications. The first application, called iMPASSE, focusses on the orchestration of services, the second application, SURA is targeted towards the orchestration of a group of services, and the third application (SWOOP) is more concentrated on the handling of failures during orchestration in a nomadic network.

Application with focus on Service Orchestration

The infrastructure of the Brussels National Airport is equipped with several applications to assist its personnel. iMPASSE (the Missing Person ASSistancE application) is an application which implements the necessary actions that must be performed when a passenger is too late for boarding. Consider the following example scenario:

Peter lives in Brussels and wants to spend his holidays in New York city. His plane leaves Brussels International Airport at 13:50 and he makes a transit at the airport of Frankfurt. Ten minutes before boarding he has not yet entered the boarding area. At the airport, Peter is announced as a missing passenger and a flight assistant is informed to start looking for him. Peter is sent a reminder on his smartphone. After ten minutes, the personnel responsible for boarding closes the gates and informs Aviapartner (the company that takes care of the luggage) to remove Peter's suitcase from the plane. Brussels' airport also ensures that the airport of Frankfurt is notified of the free seat, so a last minute offer from Frankfurt to New York becomes available. Peter gets notified that he can return home and catch another flight later.

Application with focus on Group Orchestration

The second application, called SURA (the SURpise Act application) is more focussed on group orchestration.

The headliner of a festival decides to surprise its fans with a special concert by letting them vote for the songs that will be played. In order to accomplish this, the festival's infrastructure is used to communicate with the mobile phones of the fans who are present in the festival area. All fans who are interested in participating in this vote receive a list of the band's discography. They are able to vote until two hours before the band's concert is scheduled. All votes that are cast afterwards are considered invalid. As a special bonus, the voters have the benefit of receiving the band's final playlist before the start of the concert.

Application with focus on Failure Handling

The third application is an application used to help organising workshops at the university. The application is called SWOOP (Student WOrkshOPs). During the National Week of Science the university organises several workshops to get students from secondary school acquainted with their courses. Because the registered students are not familiar with the university campus, students meet at the university's welcome hall. Afterwards, student volunteers (i.e., students of the university) will guide groups of visiting students to the place where the workshop they registered for takes place. In order to help the organisation of this event, the university employs a dedicated application that can communicate with the assistants that teach these workshops, the volunteering students that guide students through the campus, and the students who registered for a workshop.

The application sends a reminder to every assistant of a workshop, informing him/her of the workshop's location, and also contacts the administration desk to inquire for a student volunteer that can guide the registered students of that assistant's workshop. All students that registered for that particular workshop receive a message, informing them of the location of the workshop. When the administration desk has assigned a student volunteer for a group of registered students, the volunteer is contacted with the relevant information. Otherwise, the assistant of the workshop is contacted and informed that he/she must go and fetch the students oneself.

Students from the secondary school are registered as a class, and the university only has knowledge of the number of participants for each workshop. Therefore, the only way to contact each individual student (for instance, to inform him/her about the workshop's location) is by making use of the network connection in the welcome hall. When

students enter the university's welcome hall, they receive a request to download a service on their mobile device, such that the application can interact with them.

This set up is dominated by volatile connections, and, therefore, the nomadic application must deal with network failures that can occur. First of all, when the assistant of a workshop cannot be contacted (for example, because his/her mobile device is not in communication range), he/she can be contacted via email. This is not possible for the visiting students, since the university does not have contact information of an individual student. Therefore, when a particular student cannot be contacted, an announcement is made (once) informing the students of that workshop.

2.3.2 Definitions

In Section 2.1.1 we used Peltz' definition of service orchestration, which is defined as "*a business process that interacts with both internal and external web services*" [Pel03]. In this section we introduce a new definition for service orchestration, which is more applicable for orchestration in nomadic networks. Not only does this new definition consider orchestration of non-web services, the definition also incorporates the necessity for failure handling. We use the following definition for service orchestration:

Definition 1. Service orchestration *in a nomadic network is a process (or application) of which the description comprises the order in which services must be invoked, as well as a description of the data that must be available throughout the execution of the process. During the execution of a particular process, both transient and permanent (network) failures must be dealt with. Default recovery strategies must be at hand since nomadic networks are liable to volatile connections, and hence, network failures happen frequently. The specification of more accurate recovery strategies for detected failures must also be permitted.*

Besides service orchestration, we also want to orchestrate groups of services, for instance, interact with the mobile devices of all patients at a hospital. Existing approaches that interact with a group of services can be either classified in the domain of group communication or group behaviour. *Group communication* [LEH04] addresses technologies that enable effective communication between various groups in the network, using, for instance, multicasting. *Group behaviour* [GR06] is the capability of services to coordinate with each other. In order to coordinate a group of services, the dependencies between the members of the group must be managed in order to let them collaborate.

We define group orchestration in a nomadic network as follows:

Definition 2. Group orchestration *in a nomadic network is the management of a set of services that form a logical group where all its members execute the same process. The execution of the group members must be controllable in a way that transcends the individual process. Group orchestration should be able to deal with both the voluntary and involuntary removal and addition of group members. It is also essential that there are ways to synchronise and streamline the execution process of several group members.*

Group orchestration varies from group communication which is only concerned about the low-level protocols that can be used for the underlying communication. Group orchestration also differs from group behaviour. Even though group behaviour also needs to take the necessary precautions to handle unforeseen network failures, it focusses on the collaboration between the group members unlike group orchestration which aims at the management of the execution of a process by all the members of the group.

Finally, we define orchestration in a nomadic network as:

Definition 3. Orchestration *in a nomadic network is the combination of both service orchestration and group orchestration.*

In the remainder of this section we present the different criteria orchestration must fulfil. These criteria can be divided into three main categories, namely "service orchestration" (discussed in Section 2.3.3), "group orchestration" (Section 2.3.4), and "failure handling" (Section 2.3.5).

2.3.3 Criteria for Service Orchestration

In this section we postulate criteria that are concerned with the orchestration of services.

1. **Decoupled Communication**
   Processes must abstract from the intermittent connectivity of the underlying network. This criteria consists of three of Van Cutsem's criteria (we discussed in Section 2.2):
   - **Time decoupling (criterion 1)**: It is not necessary that the fixed infrastructure of the nomadic networks and the services that are used to interact with are online at the same time. It is possible that once a service is invoked, the service disconnects and performs its computation offline. Upon

reconnection, the result of the service invocation can be sent to the fixed infrastructure.

Consider the scenario at the airport. When a message is sent to the missing passenger, it is not required that his device is online at the moment the message is being sent. The message is stored and once the device connects, it is sent to the passenger.

- **Space decoupling (criterion 2)**: It is not necessary for the fixed infrastructure to know all services a priori. As we already mentioned in Section 1.2, three types of services can be distinguished in a nomadic network: stationary services, registered services and user services. The first two categories of services are typically known beforehand because they are part of the infrastructure. The latter category, the user services, are not known in advance.

  In the airport scenario, the iMPASSE application does not know the passenger that will be contacted ahead of time.

- **Synchronisation decoupling (criterion 3)**: Not having access to a remote service should be considered the default in a nomadic network. Nomadic applications, and the services that are used to execute certain tasks, should not remain unresponsive for a considerable amount of time, for instance, when a service is (temporarily) unavailable.

  For example, in the iMPASSE application, the application can only communicate with the services that are nearby (such as the service running on the smartphone of the flight assistant). In the SURA application it is possible that a fan who is exchanging his/her votes can leave the festival area and only come back after hours. For both examples we do not want the application and the services to remain unresponsive for a large amount of time.

2. **Explicit Control Flow (criterion 4)**
   There should be a focus on the control flow of the process, i.e., which tasks must be executed in which order. Identifying these different units of work eases the compensation of possible failures of these tasks, as we discuss in the third category of criteria (see Section 2.3.5). Since the focus of nomadic applications is the orchestration of its services, explicit control flow is crucial during the development and maintenance of said applications.

   For example, in the iMPASSE application, three tasks needs to be executed *in parallel*, namely sending a reminder to the passenger, notifying the assistance personnel, and announcing the missing passenger.

2.3.4 Criteria for Group Orchestration

The criteria for group orchestration in a mobile environment can be divided into two categories, namely "definition of group membership", and "synchronisation mechanisms".

1. **Definition of Group Membership**

   - **Intensional definition (criterion 5)**: Users should be able to define processes that are executed by a set of services that form a logical group. This group can be either defined extensionally, by enumerating all its members, or intensionally by giving a description all members must fulfil.

     In the SURA application the services that are contacted are not known a priori, and, hence, cannot be described extensionally. The group is defined intensionally, namely all fans of the headliner who are located in the festival area.

   - **Arity decoupling (criterion 6)**: First of all, users want to orchestrate a group of services as if they are one single unit. Moreover, as we are targeting mobile environments, this quantity of group members can fluctuate over time as new services join and disjoin the group. This requirement is known in existing literature as the need for encapsulating plurality [BI93] or arity decoupling [VC08].

     As we show in the festival scenario, the members of the group are not known a priori and can change over time. For instance, fans can arrive at the festival area at a later point in time than other fans who were there earlier and who have already received the request to vote. Those fans that arrive later will also receive the request to participate in the voting process.

   - **Dynamic modification (criterion 7)**: During the execution of the group it should be possible to redefine the members of this group. It should be feasible to restrict the members of the group by filtering out members based on a certain condition and also to change the group's description causing the arity of the group to change.

     In the SURA example, the group initially consists of all fans of the headliner. Later on, only those fans who are interested in voting are being addressed. So, in this case, the number of fans that are addressed is (possibly) decreased.

2. **Synchronisation Mechanisms (criterion 8)**
   In order to streamline all executing processes of the group members, the execution of the process's tasks should be controlled. For example, it should be managed and controlled which services can perform a certain task at a particular time. Moreover, the number of times a specific task is executed and the data needed during this execution should be controllable. Synchronisation mechanisms can let processes wait, redirect and even abort in order to let given criteria persist. This way, synchronisation mechanisms influence the amount of members of the group.

   In the festival scenario, all results need to be gathered two hours before the headliner's concert is going to start. This task should only be performed by a single service at a specific point in time. Therefore, all data needs to be collected before the execution of that task can start.

2.3.5 Criteria for Failure Handling

The last category of criteria for orchestration is called "failure handling". These criteria have an influence on both service orchestration and group orchestration.

1. **Automatic Failure Handling (criterion 9)**
   In a dynamically changing environment, the challenge is to make the large heterogeneity of services co-operate and deal with their transient and permanent failures. Services residing on mobile devices are exposed to (temporary) network failures, which must be considered the rule rather than the exception. There must be rich network and service failure detection, and a default failure handling mechanism through compensating actions, for example, transparently rediscovering a service of the same type.

   In the iMPASSE scenario, a default compensating action is executed in case a failure occurs: when a message is not received by the assistant, the message is resent (he/she is reminded again to look for the passenger).

2. **Explicit Failure Handling (criterion 10)**
   Besides automatic failure handling, the programmer must also have the possibility to specify compensating actions to overcome specific failures. Hence, it must be possible to formulate different compensation strategies on different levels of granularity by overriding and/or extending the default behaviour.

   For the SWOOP application, the default compensating action that retries to execute the task that failed is overridden. When an assistant of a workshop cannot

be contacted, because he is not in communication range, an email is sent to that person.

3. **Failure Handling for Group Orchestration**
   - **Individual failure handling (criterion 11)**: As mobile environments are liable to volatile connections, ways to detect and handle failures must be available. First of all, it should be possible to react upon a failure that occurs during the individual process execution of a single member of the group.

     For the festival application the following rule applies: in case a failure occurs within an individual process execution of a single fan, there should be a compensating action that tries to re-execute the process (for instance, resending the message).
   - **Failure handling for groups (criterion 12)**: There must be mechanisms to detect and handle failures at the group level and even propagate individual failures to the group level.

     At the festival, when something goes wrong when the votes of all fans are being gathered, the compensation should apply for all fans, hence the entire group's execution. A possible compensating action could be to ask all fans to cast their votes again. In the SWOOP application, the disconnection of the mobile device of one visiting student is handled by making an announcement. Note that this compensation is only executed once, namely the first time a disconnection occurs for a student of a particular workshop.

Here we presented the first contribution of this dissertation where we extracted criteria out of three representative nomadic applications. These criteria are divided in three categories, namely service orchestration, group orchestration, and failure handling.

2.4 Conclusion

In this chapter we presented the context of this dissertation, which is supported by both the workflow and the ambient-oriented programming paradigm. First, we introduced the main concepts of the workflow paradigm and explained how principles from this paradigm can be used for service orchestration. Secondly, we described three nomadic applications that are used throughout the remainder of this dissertation. Thirdly, we presented a definition for orchestration in nomadic networks and postulated twelve criteria we argue are necessary to allow the orchestration of services in such a network.

3

AMBIENT-ORIENTED PROGRAMMING IN AMBIENTTALK

The ambient-oriented programming paradigm is one of the pillars upon which the research of this dissertation is built. In this chapter we describe the ambient-oriented programming language, called AMBIENTTALK/2, which is a concrete implementation of this paradigm. This programming language is the successor of AMBIENTTALK [Ded06] which was introduced to meet the hardware characteristics of mobile ad hoc networks that we discussed in Chapter 2. In this chapter we focus on AMBIENTTALK/2, an updated version of the language as it is introduced by Dedecker et al. [DVM+06]. From now on we use the name AMBIENTTALK to refer to the updated language because it replaces its predecessor while staying true to its fundamental characteristics.

In this chapter we present features of AMBIENTTALK that are necessary to understand the implementation details we show in subsequent chapters of this dissertation. The reason why a dedicated chapter on AMBIENTTALK is introduced is threefold:

- The nomadic workflow language NOW is built on top of AMBIENTTALK. In order to understand the implementation details of the workflow language, an introduction to AMBIENTTALK and its language features is necessary.
- AMBIENTTALK is a language targeted towards mobile ad hoc networks. As nomadic networks can be seen as a special case of MANETs, the programming language is also part of the related work of this dissertation.
- Part of the motivation for our nomadic workflow language NOW is rooted in the difference between MANETs and nomadic networks. We discuss some drawbacks of AMBIENTTALK, namely the lack of dedicated abstractions for coordination.

3.1 Ambient-Oriented Programming

In Chapter 2 we introduced the ambient-oriented programming paradigm [Ded06]. This paradigm is motivated by the hardware phenomena that are inherent to MANETs, namely volatile connections, and zero infrastructure. Every application deployed for a mobile ad hoc network should deal with these hardware phenomena:

- **volatile connections** The phenomenon of connection volatility is also relevant for nomadic networks, as during the execution of the application, the backbone of the networks need to communicate with mobile devices in the neighbourhood.
- **zero infrastructure** This phenomenon can be neglected, as we are targeting a special case of MANET, namely a nomadic network that does consist of a fixed infrastructure.

In his dissertation, Dedecker [Ded06] put forward criteria for the ambient-oriented programming paradigm which are aimed to come to grips with these hardware phenomena. The language characteristics that Dedecker defined are [DVM+06]:

- **Classless object models**: Ambient-oriented programming languages preferably disallow the use of classes, as known in class-based languages like JAVA. In class-based languages, classes need to be copied when an object is moved from one device to another. This way, a single class can become duplicated between several devices in the network. As devices can go out of communication range at any moment in time, synchronisation between all versions of the single class becomes impossible. Ambient-oriented programming languages require that objects are self-descriptive such that no shared identity needs to be copied. Classless objects models are categorised under prototype based languages.
- **Non-blocking communication primitives**: Ambient-oriented programming languages require that the primitives for sending and receiving messages are non-blocking. The execution thread should not block when no immediate response is received. This characteristic is opportune in mobile ad hoc networks, where communicating parties can often become temporary unavailable and delays are not acceptable.
- **Reified communication traces**: Ensuring that communication primitives are non-blocking, allows devices to be out of sync while communicating. However, synchronisation is needed in order to prevent the communicating parties to be in an inconsistent state. Therefore, ambient-oriented programming languages best provide an explicit representation (a reification) of the communication that has happened.

- **Ambient acquaintance management**: Ambient-oriented applications are based upon *distributed naming* [Gel85]. Distributed naming implies that communicating parties do not need to have an explicit reference to one another a priori. In mobile ad hoc networks where devices can join and disjoin at any moment in time, this is a desired property. An ambient-oriented programming language should allow objects to spontaneously become aware of previously unknown objects based on an intensional description of that object.

In the remainder of this section we discuss AMBIENTTALK, an ambient-oriented programming language that fulfils the above language characteristics.

3.2 AMBIENTTALK

In this section we present a concrete implementation of the ambient-oriented programming paradigm, namely the AMBIENTTALK programming language in which we carried out our work. This chapter describes the language features that are necessary to understand the remainder of this dissertation. It is a shortened and reworked explanation of the language [VC08].

AMBIENTTALK is an object-oriented distributed programming language specifically aimed at mobile ad hoc networks. The programming language is implemented as an interpreter on top of JAVA. A symbiotic relationship exists between AMBIENTTALK and JAVA, meaning that AMBIENTTALK can use the class libraries from JAVA and AMBIENTTALK objects can be accessed from within JAVA. Although the language is implemented on top of JAVA, AMBIENTTALK deals with concurrency and network programming in a very different way. JAVA is a multi-threaded language which has a low-level socket API and a high-level RPC API (JAVA RMI). In contrast, AMBIENTTALK is a fully event-driven programming language which provides both event-loop concurrency and distributed object communication.

AMBIENTTALK offers direct support for the different characteristics of the ambient-oriented programming paradigm [DVM+05]:

- In a mobile ad hoc network, objects must be able to discover one another without any infrastructure (such as a shared naming registry). Therefore, AMBIENTTALK has a *discovery engine* that allows objects to discover one another in a peer-to-peer manner.

- In a mobile ad hoc network, objects may frequently disconnect and reconnect. Therefore, AMBIENTTALK provides *fault-tolerant asynchronous message passing* between objects: if a message is sent to a disconnected object, the message is buffered so it can be resent when the object reconnects.

In the following sections we describe the language's object model, its ability for meta-programming and reflection, and both its concurrent and distributed programming model. Before concluding this chapter, we discuss AMBIENTTALK's lack of abstractions for orchestration.

3.3 Object-Oriented Programming in AMBIENTTALK

Although AMBIENTTALK's main focus is its distribution model which allows the language to function in a mobile ad hoc network with unreliable connections, AMBIENTTALK is also an object-oriented programming language. AMBIENTTALK is a dynamically typed object-oriented programming language where communication between objects happens through message sends. Based upon SELF's notion of a *prototype* [UCCH91], AMBIENTTALK has a class-less model where objects can be either created *ex nihilo* or by *cloning* an existing object.

Listing 3.1 shows the definition of an object ex-nihilo. The following naming convention is used throughout the remainder of this chapter:

- An upper case is used for unique prototype objects that are only used to instantiate other objects.
- A lower case is used for objects that can have multiple versions.

```
1  def Personnel := object: {
2      def name := "";
3      def personnelNbr := 0;
4      def role := "Personnel";
5      def seniority := 0;
6
7      def init(aName, aNbr, aRole, aSeniority) {
8          name := aName;
9          personnelNbr := aNbr;
10         role := aRole;
11         seniority := aSeniority;
12     };
13
14     def  calculateSalary(salaryMapping) {
15         /* salaryMapping is a data structure mapping factors to roles. */
16         def factor := salaryMapping.get(role);
17         seniority * factor;
18     };
19 };
```

Listing 3.1: AMBIENTTALK: Definition of an object ex-nihilo.

In Listing 3.1 a new object, called `Personnel`, is defined ex-nihilo. The `object:` keyword that is used for the creation of a new object takes as its argument a *closure*, which is used to instantiate the object. An AMBIENTTALK object consists

of fields, that represent the object's state, and methods representing the behaviour of that object. In the example above, four fields (lines 2-5) and two methods (lines 7-12, 14-18) are present. In AMBIENTTALK the keyword `def` is used to associate values with variables by using the following syntax: `def variable := value`.

As we already mentioned, objects can also be created by cloning and adapting an existing object.

```
def assistant := Personnel.new( "Alice", 151, "flight assistant", 17 );
```

In the above code snippet a new object is instantiated by cloning the existing prototype `Personnel`. Every object understands the `new` method which makes a shallow copy (i.e., a clone) of the receiver object and initialises the clone (`assistant` in our example) by invoking its `init` method with the arguments that were passed (in the example `"Alice"`, `151`, `"flight assistant"`, and `17`).

AMBIENTTALK also allows taking a clone without calling the `init` method. This can be realised using the `clone:` keyword, as is shown in the following code snippet.

```
def assistant := clone: Personnel;
```

3.3.1 Delegation

AMBIENTTALK features *delegation* in order to allow code reuse. An object can be created by extending an existing (parent) object, as is shown in the following code snippet.

```
def FlightPersonnel := extend: Personnel with: {
    def flightNbr := 0;
};
```

The `extend: with:` construct that is used on the first line of the example, is used to create a new object with a *super* field set to the given parent object (`Personnel` in our example). In AMBIENTTALK, messages that objects do not understand are delegated to the object stored in their *super* field. In the example above, any message that `FlightPersonnel` does not understand is *delegated* to the `Personnel` object. For objects that are created ex-nihilo, the `super` field is by default set to *nil*. It is important to note that in the example above both `Personnel` and `FlightPersonnel` remain separate objects. The `extend: with:` construct is used to define a relationship between a parent and a child, and when the child is cloned, the cloned object's *super* field is bound to a clone of the

parent bound in the *super* field. So, concretely, when the `FlightPersonnel` object is cloned, that cloned object has its own `Personnel` object bound to its *super* field, with its own fields (such as `name` and `personnelNbr`).

3.3.2 Scoping

AMBIENTTALK makes the distinction between two kinds of scope, namely lexical scope and object scope. *Lexical scope* is the set of all variables that are lexically visible in the program text: all variables in an enclosing scope are part of the lexical scope of the enclosed (nested) scope. The *object scope* is introduced to define the scope in which methods and fields of an object are looked up. This scope constitutes a chain of delegation: the object scope is defined as all variables of the object extended with those of its parent object (the object referenced by the object's *super* field). AMBIENTTALK defines two rules which determine the kind of scoping that should be applied:

1. Unqualified access to a variable (or a method invocation) is always resolved in the lexical scope. For instance, `x` and `f()` is resolved in the lexical scope.
2. Qualified access to a variable (or a method invocation) is always resolved in the object scope. Hence, `obj.x` and `obj.f()` is resolved in the scope consisting of all variables of `obj` and its parent object.

The lexical scoping can be determined statically whereas the object scope is subject to late binding. The way in which fields of an object can be accessed varies depending on the interaction between object inheritance (delegation) and the object scope. Consider the following code example:

```
def Personnel := object: {
    def role := "Personnel";

    def getStaticRole() { role; };

    def getDynamicRole() { self.role; };
};
```

In the code snippet above, two different accessor functions are defined to access the object's `role` field. The first accessor function performs an unqualified access and applies lexical scoping, whereas the second one is a qualified accessor function, resulting in looking up the variable `role` in the object scope. The `getDynamicRole` accesses the field through a *self-send*. As we already mentioned, AMBIENTTALK uses *super* to indicate the parent object. *Self* is used to indicate the receiver object. In contrast to *super*, *self* is a pseudo-variable, not a variable

that can be assigned to. In the above example, both accessors will give the same result.

When we extend this example with delegation, the difference between the two accessor functions becomes clear.

```
def FlightPersonnel := extend: Personnel with: {
    def role := "FlightPersonnel";
};
```

When we now invoke `FlightPersonnel.getStaticRole()` the result will still equal `"Personnel"`. On the other hand, the invocation of `FlightPersonnel.getDynamicRole()` will result in `"FlightPersonnel"`.

3.3.3 Encapsulation

All fields and methods in AMBIENTTALK are considered *public*, however through lexical scoping it is possible to make a field or method *private* to a scope. The code in Listing 3.2 shows the definition of an object inside a function definition. A *function* in AMBIENTTALK is defined using the keyword `def` and has the following syntax `def functionName( <arglist> ) { <body> }`. The `arglist` is a list of local variables which are evaluated in an applicative order (i.e., the arguments are evaluated one by one, from left to right).

The fields and methods of the object that is defined inside a function definition are still public, and in addition the object can make use of fields and methods that are lexically visible. In this example, the object uses the `age` field from the outer function `personnel`.

```
1  def personnel(age) {
2      object: {
3          def personnelNbr := 0;
4          def seniority := 0;
5          ...
6
7          def ageHired() {
8              age - seniority;
9          };
10     };
11 };
```

Listing 3.2: AMBIENTTALK: Encapsulation.

3.4 Concurrent Programming in AmbientTalk

AmbientTalk uses the *actor model* to support concurrency [Agh86]. The actor model AmbientTalk exploits is based upon the programming language E's communicating event loops [MTS05]. This model differs from the traditional actor model that was proposed by Agha [Agh86], as E's model has a unified concurrency model that consists of both actors and objects. The E language introduces the notion of a *vat*, which is a container of regular objects. Inside such a single vat computation happens sequentially. Each vat has an *event loop*, a thread of execution that processes events from an event queue. Processing these events is realised by invoking its corresponding event handler. Vats can communicate by exchanging messages between their event loops. In the remainder of this chapter we use the term *actor* to denote a single unit of concurrency, and this terminology is interchangeable with the terms event loop and vat.

A single AmbientTalk virtual machine can host multiple actors that run concurrently. Every actor is represented as an event loop and the concurrency mechanism of AmbientTalk is based on the notion of communicating event loops. An actor consists of a message queue, a thread to process incoming messages, and a collection of objects that are hosted by that actor, as can be seen in Figure 3.1. The dotted lines in the figure represent the actor's event loop thread which takes messages from its message queue and synchronously executes the corresponding methods on the actor's hosted objects.

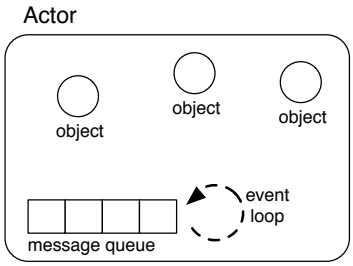


Figure 3.1: AmbientTalk actor.

When an actor is created, it initially hosts a single object, which is called the actor's *behaviour*. An actor can be created using the `actor:` keyword, as can be seen in the following code snippet.

```
actor: {
    def name = "Brussels National Airport";

    def print() { system.println(name); };
};
```

Actors are created in a similar way objects are created, they are also instantiated using a closure. Like objects, actors can also be nested, although there are some restrictions concerning access to the enclosing lexical scope. In order to disable race conditions, actors are not allowed to directly access the enclosing lexical scope. Once an actor is created, the creating actor and the created actor run in parallel, and the creating actor retrieves a so-called *far reference* to the newly created actor. A far reference is an object reference that crosses actor boundaries in AMBIENTTALK. As we discuss in Section 3.5, far references may refer to remote object references. The difference between a regular object reference and a far reference is the fact that for the former synchronous access is possible while for the latter access is necessarily asynchronous. Figure 3.2 shows the concurrency model employed by AMBIENTTALK, where an actor is represented as a communicating event loop.

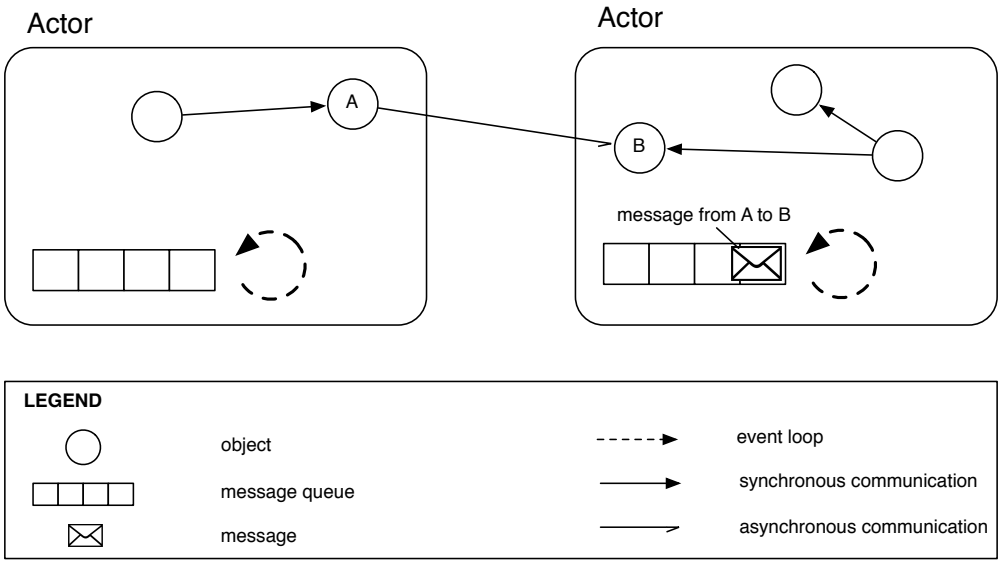


Figure 3.2: Concurrency model of AMBIENTTALK.

In figure 3.2 two actors that run parallel are depicted. As we can see, the message queue from the second actor contains a message that has been sent from A to B. This is an example of a message that is asynchronously sent via a far reference to an object that is located in another actor. Such an asynchronous

message sent via a far reference also results in a new message being added to the receiving actor's message queue.

3.4.1 Asynchronous Message Sending

Asynchronous message sending, denoted as `obj<-m()`, does not return any value by default.[1] As we already mentioned, method invocations on a far reference must be accomplished through asynchronous message sends. When such an asynchronous message is being sent, its parameters are either passed by copy or by far reference:

- Native data types are passed by copy.
- Objects and closures are always passed by far reference.

3.4.2 Isolates

Because objects are passed by far reference, the recipient actor needs to access the object asynchronously and needs to perform inter-actor communication. In order to allow the recipient actor to operate on a copy of the object synchronously, AMBIENTTALK introduces the notion of *isolates*. An isolate is an object that is isolated from its surrounding lexical scope and is passed by copy. Isolates differ from regular objects in three ways:

- Isolates have no access to their surrounding lexical scope.
- Isolates are passed by copy.
- External method definition on isolates is not allowed.

Isolates are created similarly to objects, only now the keyword `isolate:` should be used instead of `object:`.

3.4.3 Futures

As stated before, asynchronous message sends do not return any value by default. In order to enable computation with the actual return value of such sends, AMBIENTTALK uses so-called *futures*. A future is a place-holder object for the actual return value. When an asynchronous invocation is completed, the future is replaced with the actual return value. We say that the future is *resolved* with that value. Every object that refers to the future will transparently refer to the actual return value once the future is resolved.

Objects are only allowed to send asynchronous messages to futures. In case the future is not resolved yet, these message sends are buffered. When the future is

[1] An asynchronous message send returns `nil`.

eventually resolved, these buffered messages are asynchronously forwarded to the actual return value of the original message send. Actors can register their interest in the resolved value of a future by registering an observer that is invoked later, when the future is resolved. These observers allow other operations than asynchronous message sends on a future. Once the future is resolved those registered observers are asynchronously notified.

When a certain computation that relies on the actual return value of an asynchronous message send needs to be executed, that computation awaits the future being resolved. Consider the following example where all passengers of flight `"BA432"` that have already checked in need to be contacted.

```
when: checkinService<-retrievePassengers("BA432") becomes: { |collection|
    collection.each: { |passenger| ... /* contact passenger */ };
};
```

In the code snippet above the AMBIENTTALK `when: becomes:` event handler is installed to await the actual return value of the `retrievePassengers` message. All AMBIENTTALK event handlers are instantiated with a *block closure*, which has the following syntax `{ |<parlist>| <body> }`. If the block closure does not require any parameters, the `|<parlist>|` can be omitted.

When the future is resolved, the actual return value (in this example a collection containing all passengers) is bound to the variable `collection`. Only after the future is resolved, the body of the block closure is executed. Note that inside the body of this block closure, another block closure is used to iterate over all the passengers in the collection.

3.5 Distributed Programming in AMBIENTTALK

In the previous section we presented AMBIENTTALK's concurrency model which is based upon the notion of event loop actors. By introducing far references, actors residing on a single device are able to communicate with each other. Recall that references between objects that are owned by different actors are always far references which only permit asynchronous access. In this section we discuss how distributed programming in AMBIENTTALK can be realised, allowing communication between actors that are located on different devices. The objects that are residing on different devices must be owned by different actors, that can communicate with each other through far references. The use of far references to establish inter-actor communication (between actors possibly residing on different devices) guarantees that all distributed communication is asynchronous.

Distributed programming is subject to partial failures. We give an overview of AMBIENTTALK's units of operations:

- **Object**: unit of designation
- **Actor**: unit of concurrency
- **Interpreter**: unit of partial failure
- **JAVA Virtual Machine (JVM)**: unit of termination

As we already mentioned in Section 3.4, an actor can *host* several objects. Objects that are owned by the same actor are said to be *local*.

Similar to the way a single actor can host several objects, an AMBIENTTALK interpreter can *host* multiple actors for which inter-actor communication is realised through far references. Two objects that are owned by different objects are considered to be *remote*, even if those actors are hosted by the same interpreter. Within one interpreter there is no notion of partial failure, i.e., connections between actors within a single interpreter never fail. Therefore, interpreters are the unit of partial failure.

Because AMBIENTTALK is built on top of JAVA, a single JAVA Virtual Machine can *host* one or more AMBIENTTALK interpreters. JAVA Virtual Machines are the unit of termination because either *all* interpreters within the JVM are terminated, or *none* of them are.

In the remainder of this section, we first explain how remote objects can be discovered in the network. Afterwards we describe the language constructs provided by AMBIENTTALK in order to overcome (partial) failures.

3.5.1 Exporting and Discovering of Objects in AMBIENTTALK

In order to allow objects of another actor to be discovered, those objects must be made accessible to other devices in the network. Objects in AMBIENTTALK can be made remotely accessible by using the `export: as:` construct that publishes an object under a given service type on the network, as is shown by the following code snippet.

```
1  deftype Personnel;
2
3  def service := object: {
4      def role := "Personnel";
5
6      def print() {
7          system.println("Personnel : " + role);
8      };
9  };
10
11  export: service as: Personnel;
```

In this example, an object is exported with the service type `Personnel` (line 11). AMBIENTTALK uses *type tags* to represent service types. Objects can be tagged with zero or more type tags. Such a type tag is defined using the `deftype` construct, as we can see on line 1 in the code snippet above. It is possible to define a hierarchy of type tags: `type1 <: type2` denotes the relation that `type1` is a subtype of `type2`.

Type tags are used as a lightweight classification mechanism, they do not give any guarantees on the interface of the published object. Hence, a type tag is not associated with a set of methods and they cannot be used for static type checking.

Now that we have shown how objects can be published on the network using a type tag, we explain how they can be discovered by actors residing in the network. AMBIENTTALK uses a peer-to-peer discovery lookup mechanism based upon publish-subscribe. Recall that as AMBIENTTALK is specifically sculpted towards mobile ad hoc networks, the language cannot rely on any centralised lookup infrastructure. In order to react upon discovery of objects in the network, AMBIENTTALK provides a `when: discovered:` observer. This observer is instantiated with a block closure that is executed at the moment a service of the correct type tag is discovered. Consider the following example code

```
def people := [];   /* A table */

when: Personnel discovered: { |person|
    /* Add the object to the table */
    people := people + [person];
};
```

where all personnel that are present at the airport are registered. So, when an object with the given type tag `Personnel` is published on the network, the installed `when: discovered:` observer is triggered and its block closure is executed with the variable `person` bound to the discovered object.

We must clarify that the `when: discovered:` observer is only triggered *once*, namely the first time an object with type tag `Personnel` is being discovered. Hence, when all personnel present at the airport must be collected in some kind of data structure (the AMBIENTTALK table defined on the first line of the code snippet), a `whenever: discovered:` observer needs to be installed. The difference between the `when: discovered:` and the `whenever: discovered:` observer is that the latter one is triggered *for each* discovery of an object with the specified type tag.

It is important to note that the observers for discovery are not triggered by objects that are published in the same actor as the one that specified the observers. Hence, an actor does not discover its own published objects.

3.5.2 Dealing with Failures

Since AMBIENTTALK is sculpted to function in MANETs where volatile connections dominate, both transient and permanent failures happen frequently. As we mentioned before, AMBIENTTALK uses far references to allow communication between remote objects. Using these far references, the current state of the network (i.e., whether the remote objects are connected or not) is abstracted away:

- When a message is sent from object A to B, and the far reference to B is connected, the message is asynchronously sent to B.
- When a message from object A is sent to object B, and the far reference to B is disconnected, the message is buffered in the inbox of the far reference until it reconnects. Upon reconnection, all messages stored in the far reference are transmitted.

This way, AMBIENTTALK's far references deal with transient failures. However, sometimes the programmer wants to be able to know the current state of the network and react upon disconnections and reconnections. To this end, AMBIENTTALK provides the `whenever: disconnected:` and `whenever: reconnected:` observers. The following code snippet extends the previous example and shows how these observers can be used.

```
1  def people := [];
2
3  when: Personnel discovered: { |person|
4      people := people + [person];
5
6      whenever: person disconnected: {
7          /* Remove the object from the table */
8          people := people.filter: { |element| ! (element == person) };
9      };
10
11     whenever: person reconnected: {
12         /* Add the object to the table */
13         people := people + [person];
14     };
15 };
```

In this example, whenever a personnel disconnects, it is removed from the collection that contains all personnel that is present at the airport (line 6-9). Whenever the person reconnects, the far reference pointing to that person is again added to the collection (line 11-14).

As we have explained, AMBIENTTALK deals with transient failures by using far references to communicate between remote objects. Moreover, the language provides constructs to allow the programmer to react upon the disconnection and reconnection of objects. In order to deal with permanent failures, AMBIENTTALK introduces *leased object references* [GVV+09]. A leased object reference is a far reference that allows access to a remote object for a restricted time. When the time period elapses, the access to that remote object is retracted.

Leasing is integrated into futures in order to let an asynchronous message expire either due to a timeout or when the computed return value is received. The timeout for the lease on a future can be set by annotating the asynchronous message with a `@Due` annotation as we show in the following example:

```
def searchFuture := assistant<-searchPerson(passenger)@Due(minutes(10));
when: searchFuture becomes: { |reply|
    if: reply then: {
        system.println("passenger is found");
    } else: {
        system.println("passenger not found");
    }
} catch: TimeoutException using: { |e|
    // unable to find person
};
```

In this code example an assistant of a flight company is contacted and ordered to look for a missing passenger. When the passenger is not found within ten minutes, some other procedures need to be started.

When the future is resolved, the value of the variable `reply` is a boolean value denoting whether the person has been found or not. However, when the return value is not received before the leased time elapsed, a `TimeoutException` is raised when the future's lease expires.

3.6 Reflective Programming in AMBIENTTALK

AMBIENTTALK supports *reflection* [Mae87] which allows the adaptation of the programming language as well as the addition of new language constructs. The programming language has support for two types of reflective access, namely introspection and intercession. *Introspection* is the ability of a program to inspect its own state at runtime. *Intercession* allows the semantics of meta-level operations to be changed. In order to allow reflective programming, AMBIENTTALK relies on the notion of *mirrors*, objects that allow reflection on the object's state and behaviour.

3.6.1 Mirrors

AMBIENTTALK employs a *mirror-based architecture* to provide support for reflection [BU04]. A *mirror* object offers reflective access to a single object, called the *reflectee*, which does not have a field to access the mirror.

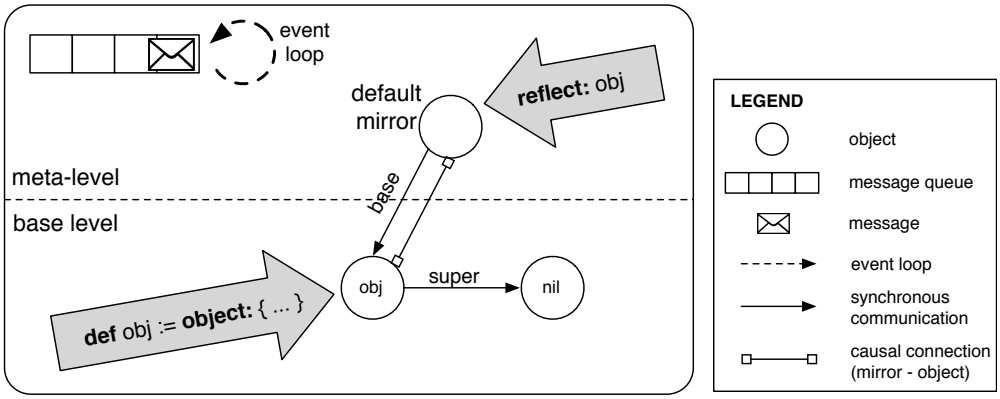


Figure 3.3: AMBIENTTALK: mirrors.

Figure 3.3 shows a number of objects residing in an AMBIENTTALK actor. An actor consists of a meta-level and a base level. In the actor depicted in Figure 3.3 an object is defined ex-nihilo. Therefore, the actor's super field refers to `nil`. The actor's meta-level has a single object, namely the default mirror provided by AMBIENTTALK. This mirror object can be accessed using the `reflect:` keyword, as we show in the following code snippet.

```
1 /* Define a personnel object */
2 def assistant := Personnel.new( "Alice", 151, "flight assistant", 12 );
3 /* Retrieve its mirror object */
4 def mirror := reflect: assistant;
```

On the second line of code in the above code snippet we define an AMBIENTTALK object, by cloning the prototype that is given in Listing 3.1. By using the `reflect:` keyword the mirror object of the `assistant` object is retrieved, as is shown in the third line of the code snippet above. Several methods are provided by AMBIENTTALK that allow the state of an object to be inspected through a mirror object. Consider for instance the following piece of code

```
1 /* Access all of the fields defined on the mirror's base object */
2 mirror.listFields();
3 /* Reify a field of the mirror's base object */
4 grabField(`super);
5 /* Reification of the definition of fields */
6 mirror.defineField(`foo, 42);
```

```
7  /* Reify the operation for field-selection */
8  mirror.invokeField(assistant, 'foo);
```

Listing 3.3: AMBIENTTALK: facilities of mirrors.

The code in Listing 3.3 shows some facilities provided by mirrors. The facilities that are provided by the mirror objects allow accessing and reification of methods, fields, and slots[2]. In the code example we only show those methods related to the object's fields. On line 2 we retrieve all fields of the mirror's base object, namely the fields of the `assistant` object. The result is a table containing the following fields: `<field:super>`, `<field:name>`, `<field:personnelNbr>`, `<field:role>`, and `<field:seniority>`. On the fourth line, we show how a field can be retrieved. The result of this method invocation is the field `<field:super>`. It is also possible to define a field in the mirror's base object, as is shown on line 6. After this line of code is executed, the `assistant` object has a new field, namely `foo`. On line 8, this field is invoked which gives the number `42` as a result.

As we already said, the operations in Listing 3.3 are only a subset of all operations that can be performed on an object's mirror object. Those methods belong to one of the following protocols [MVT+09] that form together AMBIENTTALK's meta-object protocol (MOP) [KR91].

- **Message Invocation Protocol**: allows the interception upon receiving an asynchronous message or upon the invocation of a synchronous message.
- **Object Marshalling Protocol**: consists of the methods `pass` and `resolve` that describe how the object is passed to other actors and how it needs to be resolved upon arrival.
- **Slot Access and Modification Protocol**: provides methods that allow the modification of slots and capture the dynamic access to an object's slots.
- **Structural Access Protocol**: has methods that either list all available slots, allow access of a slot, or add a new slot to the mirror's base object.
- **Object Instantiation Protocol**: has two methods, `clone` and `newInstance`, that are invoked when `clone: obj` or `obj.new(@args)` are used respectively.

[2] Once a mirror has been created, it can be used to inspect an object as a collection of so-called *slot objects*. Slot objects are objects which bind a name to a value (for instance, a method slot is simply a slot that binds a name to a method object).

- **Relational Testing Protocol**: provides the methods `isCloneOf` and `isRelatedTo` that verify whether the object is a clone or a combination of cloning and object extension.
- **Type Tag Protocol**: consists of the method `isTaggedAs` that verifies whether the mirror's base object is tagged with a particular type and the method `listTypeTags` to retrieve all type tags of the mirror's base object. The former method is *transitive*, meaning that type tags of parent objects are also considered, whereas the latter operation does only consider the object's *own* type tags.
- **Evaluation Protocol**: has two methods, namely `eval` and `quote`, in order to ensure that any object can be part of a parse tree.

3.6.2 Mirages

In the previous section, we have shown the facilities AMBIENTTALK provides to allow introspection through mirrors. In order to allow intercession, AMBIENTTALK introduces so-called *mirages* that enable the programmer to change the operations provided by AMBIENTTALK's MOP. The programming language provides a `defaultMirror` which encapsulates the default semantics of its MOP. When only small changes to the semantics of the MOP need to be made, this default mirror can be adapted by overriding its methods, as we show in the following code snippet.

```
1 def createTracingMirror(baseObject) {
2     extend: defaultMirror.new(baseObject) with: {
3         def invoke(slf, invocation) {
4             system.println("invocation of " + invocation.selector);
5             super^invoke(slf, invocation);
6         };
7     };
8 };
```

We define a `TracingMirror` that allows tracing method calls. To this end, the `invoke` method of AMBIENTTALK's `defaultMirror` needs to be overridden. This is done on lines 3-6, where before the actual method invocation (line 5), the logging takes place (line 4)).

AMBIENTTALK provides the `mirror:` primitive which is syntactic sugar for extending the default mirror. The implementation of the above example can therefore also be written as

```
def TracingMirror := mirror: { def invoke(slf, invocation) { ... }; };
```

When we want to trace the method invocations of an object, for instance, the `assistant` object defined earlier in Listing 3.1, that object's mirror needs to be the one we defined above. Such an object which has a custom mirror is called a *mirage*. Mirages can be constructed in two ways, as we show in the following code.

```
1 object: {
2     def role := "Personnel";
3     ...
4 } mirroredBy:{ |emptyBase| createTracingMirror(emptyBase) };
5
6 object: {
7     def role := "Personnel";
8     ...
9 } mirroredBy: TracingMirror;
```

Listing 3.4: AMBIENTTALK: construction of mirages.

A first way to define a mirage is shown on lines 1-4 in Listing 3.4. The block closure that is passed to the `mirroredBy:` primitive is called a *mirror construction closure*. Such a closure is applied to a new mirage object and returns a new mirror that reflects upon that mirage. When the mirror is created, the block closure that is used to instantiate the object with is executed.

On lines 6-9 an alternative way to construct a mirage object is presented. Here, a prototype mirror (`TracingMirror`) is passed as an argument to the `mirroredBy:` primitive.

In Figure 3.4 we depict an actor and all the objects that are involved.

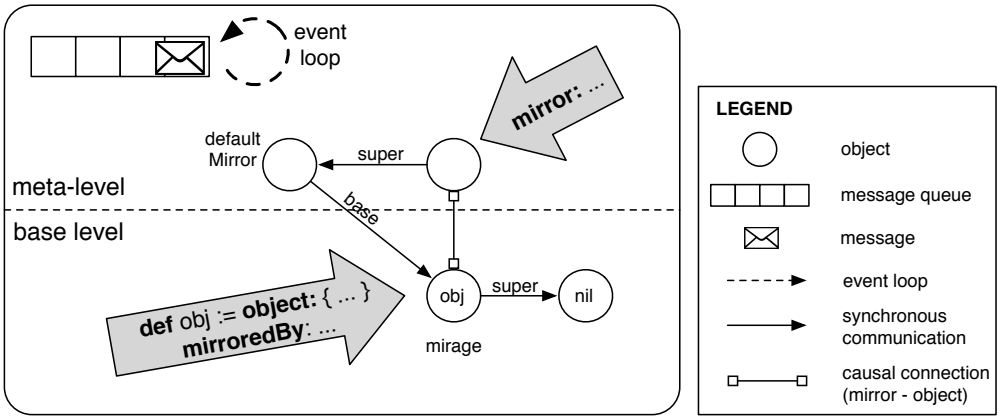


Figure 3.4: AMBIENTTALK: mirages.

The base level consists of a mirage (object with a custom mirror) which is created ex nihilo. Hence the object's super refers to `nil`.

The mirage has a custom mirror, which is created using the `mirror:` keyword. So, at the actor's meta-level we find two objects, namely the newly created mirror and the default mirror. As the new mirror is created using the `mirror:` primitive, this object is created by extending the default mirror (i.e., its super refers to the `defaultMirror` object).

The mirage on the base level of the actor is created using the `mirroredBy:` primitive. The mirror construction closure that is passed to that primitive is applied to the mirage object. Hence, the mirror's `base` pointer refers to the mirage object.

3.7 Ambient References

People everywhere are surrounded by a wide range of mobile and stationary devices that can perform all kinds of services. For instance, today's smartphones are able to determine temperature, location, orientation, etc. Since devices and services are everywhere nowadays, there is a need to address several services or devices at the same time. For instance, to determine the current temperature at a certain location, one could retrieve that information from several temperature services in the neighbourhood and calculate the average. AMBIENTTALK introduces a dedicated data type, namely *ambient references*, to designate a group of objects that are within reach [VC08]. An ambient reference refers to a collection of nearby services of the same type which is kept up-to-date. When new services of the specific type are discovered, they are added to the collection and unresponsive services are removed.

In the following code example we show how an ambient reference can be created.

```
deftype Passenger;
def passengers := ambient: Passenger;
```

In this example the ambient reference, created using the `ambient:` primitive, collects all services of type `Passenger` that are in reach.

Sometimes, one wants to take a *snapshot* of the object references contained in the ambient reference. Such a snapshot results in a collection of references to nearby objects that are in reach at the moment is taken, and once the snapshot has been taken the collection is not kept up-to-date. Such a snapshot of an ambient reference can be created as follows:

```
def people := snapshot: passengers;
```

The variable `people` is a reference to a table that contains all passengers that are within reach. This table can contain no elements when the snapshot was taken at a moment no passenger services where within communication range. It s possible that, when iterating over this retrieved collection, a service has gone out of communication range. However, as this collection consists of far references to services, communication is still decoupled in time and synchronisation.

Communication with ambient references is achieved by sending asynchronous messages that can be annotated. These annotations are used to vary the number of services that are addressed:

- **@One**: point-to-point communication is realised by annotating the ambient message with the `@One` annotation.
- **@All**: one-to-many messages are expressed by annotating a message with the `@All` annotation.
- **@Any**: using the `@Any` annotation result in the message being sent to *any* discovered service with the given type tag.

Furthermore, it is possible to annotate a message with an expiration period. This annotation allows you to send messages to the ambient references, and this message is also sent to all services that are newly discovered during this expiration period. However, once this period has elapsed, the message is no longer sent to matched services. The following code snippet shows an example where a message is annotated with an expiration period.

```
deftype Passenger;
def passengers := ambient: Passengers;
whenAll: passengers<-getFlightInfo()@Expires(minutes(5)) becomes: { |reply|
    // process the retrieved flight information
};
```

In this example, the flight information of all passengers needs to be accumulated. To this end, AMBIENTTALK introduces the notion of *multifutures* that provide more possibilities for synchronisation than regular futures. Multifutures are a convenient abstraction to gather all of the replies to a broadcast at a single point in the code.

In this code example a special kind of event handler is registered on the multifuture, namely `whenAll: becomes:`. This construct allows the programmer to gather accumulated answers of all the far references contained in the ambient references. In this example, when the future is resolved the value of the variable `reply` is an array containing the flight information of all passengers.

It is also possible to register a `whenEach: becomes:` event handler on a multifuture. This observer is triggered on each value or exception of the multifuture. Note that both the `whenAll: becomes:` and `whenEach: becomes:` observers can be registered on regular futures as well.

In all the above examples, we used a type tag to classify the services that belong to the ambient reference. However, ambient references can be classified in three ways:

1. **type tag**: represents a service type;
2. **protocol**: is a description of the set of selectors (message names) to which an object responds;
3. **filter**: represents a predicate (unary block closure that returns a boolean value).

In the code snippet below, each of these classification mechanisms is used:

```
1  deftype Passenger;
2
3  def passengers :=  ambient: Passenger;
4
5  def PassengerProtocol := protocol: {
6      def getFlightInfo();
7  };
8  def passengers := ambient: PassengerProtocol;
9
10 def passengers := ambient: Passenger where: { |p| p.flight == "BA472" };
```

On line 3 in the code snippet an ambient reference is defined using the type tag `Passenger`. An ambient reference defined using the protocol categorisation mechanism is shown on line 8. The protocol that is used to categorise the service (defined on lines 5-7) specifies that the service must have a `getFlightInfo` selector. The third categorisation mechanism is shown on the last line of the code snippet where an ambient reference is defined using the predicate which evaluates to true when the service's field `flight` equals `"BA472"`.

3.8 Limitations

Now that we have presented AMBIENTTALK and the language features that are necessary to understand the remainder of this dissertation, we present the issues involving service orchestration. This dissertation focusses on the orchestration of asynchronously distributed processes in a nomadic network where volatile connections dominate. Although the ambient-oriented programming paradigm formulates language characteristics that allow the development of applications in mobile ad hoc networks, developing large-scale complex application is far from trivial. We demonstrate this by implementing the example application described in Section 2.3.1 and illustrate several problems using its implementation in AMBIENTTALK.

We now show part of the implementation of the iMPASSE application in AMBIENTTALK in Listing 3.5. In this code snippet we implement the first part of the

application, namely a passenger is missing, and the procedure to start looking for him/her is started. For this procedure three tasks need to be executed: the person must be announced, a reminder text message must be sent to his/her mobile phone, and an assistance personnel member must go look for the passenger. Either one of these tasks leads to an acknowledgement that the person is found, or after some time the procedure is finished and it is concluded that the person has not been found.

```
1  deftype ReminderService;
2  deftype AnnouncementService;
3  deftype AssistancePersonnel;
4  def reminderS := ambient: ReminderService;
5  def announcementS;
6  def assistanceP := ambient: AssistancePersonnel;
7
8  def missingPerson(passenger, flight) {
9      def results := [];
10     def numberNotFounds := 0;
11
12
13     def check (reply) {
14         def found := false;
15         if: (! found) then: {
16             if: reply then: {
17                 found := true;
18                 personFound(passenger, flight);
19             } else: {
20                 numberNotFounds := numberNotFounds + 1;
21                 if: (numberNotFounds == 3) then: {
22                     personNotFound(passenger, flight);
23                 };
24             };
25         };
26     };
27
28
29     def hulp(future, service) {
30         when: future becomes: { |reply|
31             results := results + [reply];
32             check(reply);
33         } catch: TimeoutException using: { |exception|
34             personNotFound(passenger, flight);
35         };
36         when: service disconnected: {
37             raise: XServiceDisconnection.new("The service disconnected.");
38         };
39     };
40
41
42     def f := reminderS<-missingPerson(passenger)@Due(minutes(10));
43     hulp(f, reminderS);
44
45
```

```
46      when: AnnouncementService discovered: { |svc|
47          f:= svc<-missingPerson(passenger)@Due(minutes(10));
48          hulp(f, svc);
49          when: seconds(30) elapsed: {
50              raise: XServiceNotFound.new("Required service cannot be found.");
51          };
52      };
53
54      f := assistanceP<-missingPerson(passenger)@Due(minutes(10));
55      hulp(f, assistanceP);
56  };
```

Listing 3.5: Implementation of part of the airport scenario in AMBIENTTALK.

Before we describe the implementation in detail, it is important to know that some services are known beforehand, whereas for other services in the example scenario it suffices to discover a service of the correct service type when needed. In the part of the example application we implemented, three services must be invoked (the reminder service, the announcement service, and the assistance personnel). Both the reminder service and assistance personnel that must be addressed are known beforehand, and a reference to that service is available. As we can see on line 4 and line 6 in Listing 3.5, references to these two services are retrieved and bound to the variables `reminderS` and `assistanceP` respectively.

The `missingPerson` function that is defined on lines 8-56 implements the invocation of the three services in parallel. The implementation of the invocation of the reminder service (lines 42-43) and the assistance personnel (54-55) is very similar: Because a reference to the service is available, service invocation is achieved by sending an asynchronous message send to the given far reference. This asynchronous message is annotated (with `@Due`), such that a timeout exception is caught after ten minutes. When the future, returned by the asynchronous message send, is resolved, the received reply is stored in a table containing the replies of all service invocations (line 31). The function `check` will then verify whether all three services have replied that the passenger cannot be found, or if the reply was positive, meaning that the passenger is found and that the boarding personnel should be informed to wait for the passenger.

It is important that the application can react upon certain failures, such as the disconnection of a service. The failures we want to detect are timeout exceptions, disconnections of services and the unavailability of services. As we already mentioned, timeout exceptions can be caught using the `catch: using:` event handler (on lines 33-35). The last type of failure, the unavailability of a service, is not applicable for these two services, because a reference to them is available.

The invocation of the announcement service differs from the other two, because this service needs to be discovered before the invocation. On line 46 a `when: discovered:` event handler is installed. When a service that is tagged with the type tag `AnnouncementService` is discovered, the code inside the event handler is evaluated. The invocation of that service (lines 47-48) is similar to the ones of the other two services.

Besides the registering of observers to handle timeouts and disconnections, a third observer (`when: elapsed:`) is installed to react appropriately when a service is unavailable (see lines 49-51).

From this example we can distill some general problems that arise when writing applications in AMBIENTTALK:

1. **Inversion of control**: An unwanted property of AMBIENTTALK is the fact that the application logic is divided amongst several event handlers that can be triggered independently of one another [CM06], since the language has an event-driven architecture. The control flow of an application is thus no longer determined by the programmer but by external events. This phenomenon is known as *inversion of control*. This makes the event-driven programming style not always straightforward [HO06].

   In Section 2.3.3 we argue that there should be a focus on the (control flow) of the process when orchestrating services in a nomadic network (criterion 4: explicit control flow). The inversion of control phenomenon makes it very hard to grasp the order in which tasks are executed.

2. **No automatic failure handling**: AMBIENTTALK provides the necessary language abstractions for handling failures, which can be caught by registering the appropriate event handlers. In a mobile ad hoc network, which is a highly dynamic environment where devices can join and disjoin at any moment in time, failures must be considered the rule rather than the exception. Therefore, in AMBIENTTALK failures are hidden. However, the language has no support for multiple failure handling mechanisms or high-level recovery (criterion 9).

3. **Limited support for groups**: This last limitation cannot be deduced from the above code example. AMBIENTTALK introduces the notion of ambient references to enable communication with a volatile group of proximate objects by means of asynchronous message sends [VMG+07]. Both definition by means of an intensional description and arity decoupling are supported by

this language construct. Ambient references provide synchronisation mechanisms by providing observers that are triggered either when the first service has answered or when all services have responded. However, as communication with the set of services is expressed by means of an atomic message send, it is for instance not possible to redefine the members of the group. In Section 2.3.3 this requirement is formulated as the "dynamic modification" criterion (criterion 7). Moreover, there are no mechanisms provided to express failure handling on an ambient reference, for instance, express the action that must be performed when a service of the group disconnects. In Section 2.3.3 we discuss in criterion 11 (individual failure handling) and criterion 12 (failure handling for groups) that for nomadic networks it is opportune to have failure handling when orchestrating groups of services.

3.9 Conclusion

In this chapter we introduced the ambient-oriented programming paradigm and its concrete implementation AMBIENTTALK. We did not only present the language features that are necessary to understand the remainder of this dissertation, we also discussed some limitations of the language. In this dissertation we argue that a workflow language that is built as a library of this ambient-oriented programming language can provide the necessary abstractions to orchestrate services in a nomadic environment, and can overcome the limitations we discussed in Section 3.8. We do not only present novel abstractions that adhere to the criteria we postulated for orchestration in nomadic networks (Section 2.3.3), we also present a proof-of-concept implementation on top of AMBIENTTALK. By adding these extra layers on top of AMBIENTTALK the language's limitations are tackled:

- By adding workflow patterns that describe very explicitly the control flow of the application, the order in which tasks are executed can be determined. These so-called control flow patterns abstract away AMBIENTTALK's event handlers and hence overcome the language's inversion of control problem.
- Novel abstractions that allow the default compensation for specific kinds of failures ensure that (network) failures are considered the rule rather than the exception. However, the abstractions allow application developers to specify more accurate behaviour by overriding this default failure handling mechanism.

- Patterns are added to allow the execution of (several) task(s) for a volatile set of services. These patterns also provide mechanisms to synchronise the execution of these tasks for each individual service, and for the entire group of services. Special patterns are also added in order to overcome both individual failures and failures that have an influence on the entire group of services.

4

PATTERNS FOR ORCHESTRATION IN NOMADIC NETWORKS

This dissertation advocates the use of high-level workflow patterns for orchestration in nomadic networks. These high-level abstractions ensure that the control flow of the application and the fine-grained application logic are not interwoven, and hence facilitate the development of large-scale complex applications in a nomadic environment.

In this chapter we first discuss an interpretation of activities, which is suited for nomadic networks (Section 4.1).

Secondly, we revise an existing mechanism that allows data to be passed between activities in the workflow. We present an extension of this data flow mechanism such that possible data conflicts upon synchronisation can be handled (Section 4.2).

Thirdly, we discuss patterns for orchestration in nomadic networks. We start by presenting standard workflow patterns that are used for orchestration (Section 4.3). We make use of the control flow patterns of van der Aalst [RtHvdAM06] that are widely utilised for the design and development of workflows. Thereafter, we present a specific set of novel patterns that are sculpted for nomadic networks. We introduce high-level abstractions for group orchestration as a new set of workflow patterns (Section 4.4). We also show the necessity for automatic failure handling and discuss how more application-specific failure handling can be achieved by specifying compensating actions (Section 4.5). Finally, we describe how this failure handling mechanism can be applied to the abstractions for group orchestration we proposed.

4.1 Activities

In Section 2.1.3 we present a definition of activity, which is defined as "*a description of a piece of work that forms one logical step within a process*". We refine this definition and state that an *activity* is a placeholder for a service invocation. When an activity is started, it must perform a service invocation, and process the result of this invocation.

Before explaining the different execution steps an activity must perform when started, we explain how activities are executed in a nomadic network. Activities are part of a workflow description, which resides on the fixed infrastructure of the nomadic network. The fixed infrastructure is responsible for executing the entire workflow, ensuring that the workflow description cannot become unavailable during the execution. An activity is used to invoke a service in the nomadic network. Recall that nomadic applications interact with services that can be categorised as stationary services, registered services and user services. Stationary services and registered services are known beforehand because they are part of the infrastructure. In case a *specific* service needs to be addressed, the activity has a reference to that service (as we explain in Section 4.2). Consider the example where you want to address the pilot of a particular flight. The service residing on the mobile phone of the pilot can be regarded as a registered service, and is known beforehand. Note that this does not imply that no failures can occur: it is for instance possible that the service is unresponsive. Sometimes the service that needs to be addressed is not known beforehand (user service), in which case the activity must first discover a service of the correct type. For example, when in need of assistance it suffices to discover a service running on the smartphone of *a* (random) assistance personnel who is in the neighbourhood.

In general, an activity must perform several execution steps:

1. **Service discovery**: discover a service of the right type in the network.
2. **Service invocation**: invoke the found service.
3. **Response management**: process the result of the service invocation.

The lifecycle of an activity is schematically presented by the state diagram in Figure 4.1. When no reference to the service is given, a service of the right type needs to be discovered. Service discovery either results in a reference to the service being retrieved, or results in indefinitely trying to discover a service of the correct type. In case service discovery is required, there is decoupling in space (criterion 2), because the service and the activity do not necessarily know each other a priori. The first matching service that is discovered by the activity will be

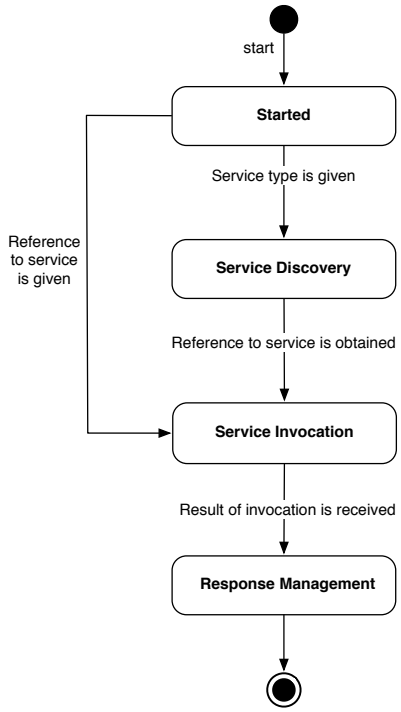


Figure 4.1: Lifecycle of an activity.

invoked. A service is said to match when the type of the service is a subtype of the type specified by the activity. After such a matching service is discovered, the activity automatically cancels the discovery mechanism such that no more matching services are discovered.

When no service of the correct type can be discovered, the activity is not doing any useful computation until a service is discovered. In order to avoid this behaviour, built-in support to handle these kinds of "failures" is needed, as we discuss in Section 4.5. In that section we present an extended state diagram that models the lifecycle of an activity with failure handling for the case where no service of the right type can be discovered.

When a reference to the service is obtained, either after discovery or because it was given from the start, the service can be invoked. Service invocation is achieved by sending an asynchronous message to the service, ensuring that both the workflow and the service are not blocked upon sending or receiving messages (criterion 3: synchronisation decoupling). It is possible that the service becomes temporarily unavailable while the activity is waiting for the result of the invocation.

This ensures decoupling in time (criterion 1), because the workflow and the service do not need to be online at the same time.

Once a return value of the service invocation is received, the result can be processed.

4.2 Data Flow

Next to the control flow perspective (see Section 4.3), which emphasises the specification of the interactions with services, an application must also specify its state, namely the data that is used by the application logic (handled by the data flow perspective). Workflow research is typically focussed on the control flow perspective, whereas the data flow perspective is rather treated stepmotherly.

Existing research has identified the following approaches for passing data between activities in the workflow [RtEv05b]:

- **Integrated control and data channels**: For this approach, data is passed simultaneously with the control flow of the workflow. An activity has only access to the data that has been passed to it.
- **Distinct control and data channels**: In this approach, data can be passed between activities using links that are unconnected to the control flow. An activity can access only the data it receives through such a data link.
- **No data passing**: For this approach, all activities of a workflow share the same data. Typically, data is accessible via a shared scope. An activity can only access the data that is defined in the surrounding scope of an activity.

In this section we extend the "integrated control and data channels" approach that enables passing information between activities in the workflow. This mechanism fits nicely with the refinement of activities which perform asynchronous invocations of services in the nomadic networks, as we explained in Section 4.1. The data flow mechanism we put forward employs a straightforward concept in which values are passed between activities in the workflow. This mechanism adheres to Sadiq's [SOSF04] requirements that must be fulfilled by a data flow model. A data flow model must have the ability to:

- manage both the input and output data of activities;
- ensure that data produced by one activity is available for other activities; and
- ensure consistent flow of data between activities.

We now describe our data flow mechanism and show how this mechanism satisfies the requirements of Sadiq. Instead of using a simple global or static

environment for our workflow language, we developed a dynamic system where the environment flows through the workflow graph and is dynamically adapted. To this end, we introduce a *data environment*, an object with a unique identifier and a dictionary associating variables with values. Each time a workflow is started, a new data environment is instantiated and passed to this workflow instance.

When an activity is executed, the service's method is invoked with its formal parameters bound to their values, which are looked up in the *data environment*. Using this concept of a data environment, the data flow is tightly coupled to the control flow of the application. We show how the data environment gets updated when an activity is started and a service is invoked. In Section 4.1 we already explained the different execution steps that must be performed when an activity is started. In case no reference to a particular service was given, a service of the right type must be discovered.

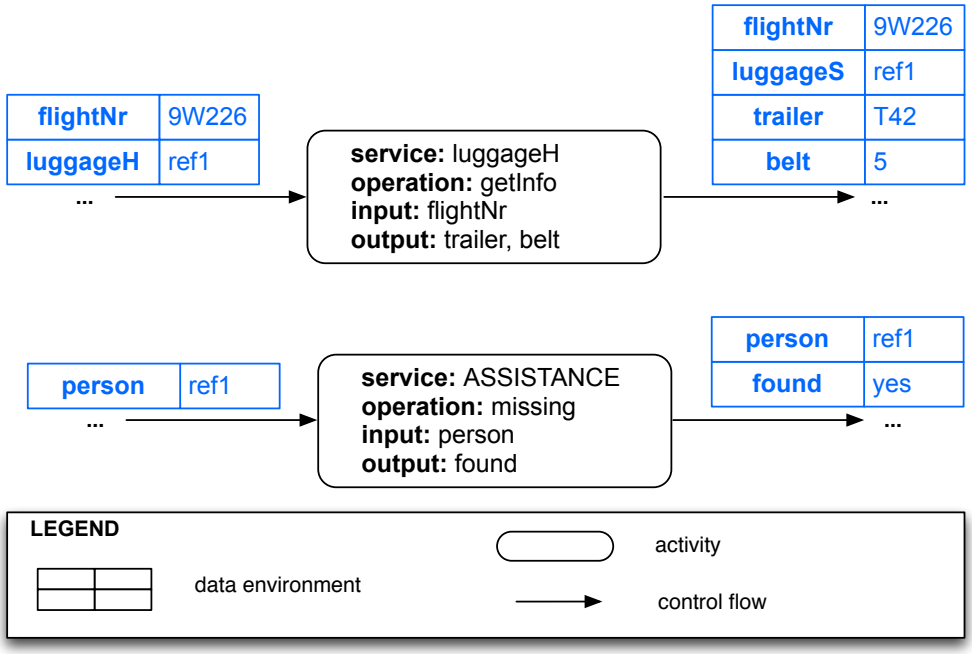


Figure 4.2: Data flow: data environment passed simultaneously with incoming and outgoing control flow edges.

In Figure 4.2 we show two examples of an activity that is started with a data environment. The first example wants to address a specific luggage handler and ask for a particular flight which trailer will transport the luggage, and to which belt

this luggage is transported. In this example, a *particular* luggage handler needs to be addressed. The second example contacts an assistance personnel member to notify him/her that a certain person is missing. In this case, any assistance personnel member that is in the neighbourhood suffices for this task.

The difference between a specific service and a service type is made by the usage of capital letters. Capital letters are used to denote a service type whereas non-capital letters are used when a reference to the service is available in the data environment. In the remainder of this chapter we use this convention to differentiate between those types of service descriptions.

The activity depicted at the top of Figure 4.2 depicts the first example scenario. This is an example of an activity where a reference to a particular service is available (`luggageH`). The reference to that service can be found in the data environment that is used to start the activity's execution. The activity shown at the bottom in the figure shows the second example where the activity is instantiated with a service type (`ASSISTANCE`). So, before a service can be invoked, a service of the right type needs to be discovered. As we can see in the figure, the data environment is extended after the activity has finished its execution. The variable bindings that are added to the environment are specified by the activity (`output`). This binding happens during the "response management" execution step of an activity (see Section 4.1).

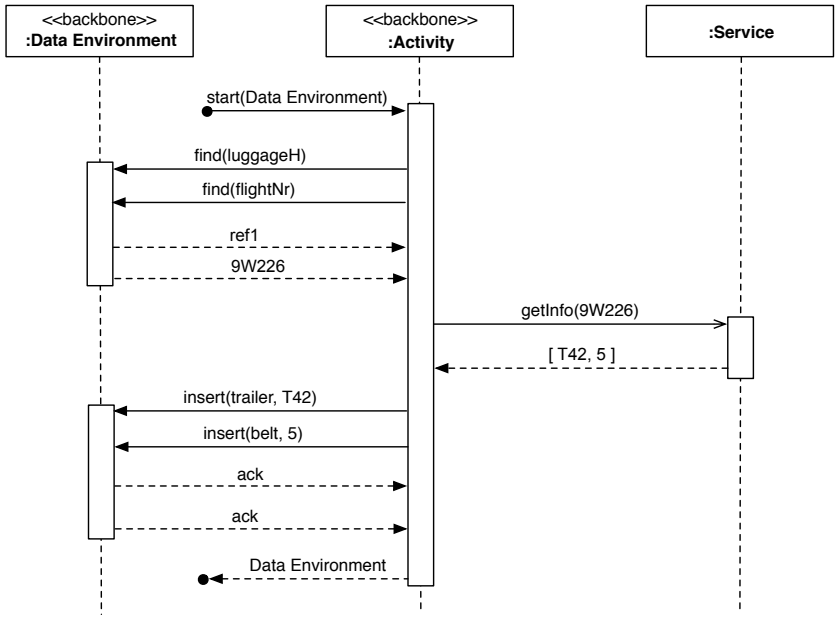


Figure 4.3: Example sequence diagram of starting an activity.

We show a sequence diagram for the top activity in Figure 4.3. When the activity is started with a data environment, the values of the formal parameters of the service invocation, `luggageS` and `flightNr` are looked up in this dictionary. Recall that for this activity a reference to the service that must be invoked is stored in the data environment. When the values are retrieved, the service is invoked by sending an asynchronous message. When the result of the invocation is received, the activity starts the response management: the received values are bound to the corresponding output variables and these new variable bindings are put in the data environment. The activity's execution is finished at the moment this updated data environment is returned as a result.

This data flow mechanism can be thought of as dynamic scoping but with special semantics for patterns such as a Synchronization, which merges multiple incoming branches, as is illustrated in Figure 4.4. If part of a workflow can be reached by more than one path, it is possible that the data environments are (completely) different. This is one of the flaws of the "integrated control and data channels" approach. Russell et al. [RtEv05b] state that in case an activity receives (potentially) different copies of data, the task must decide which data is the correct one. We propose a mechanism that allows the specification of a data merging strategy for synchronisation patterns.

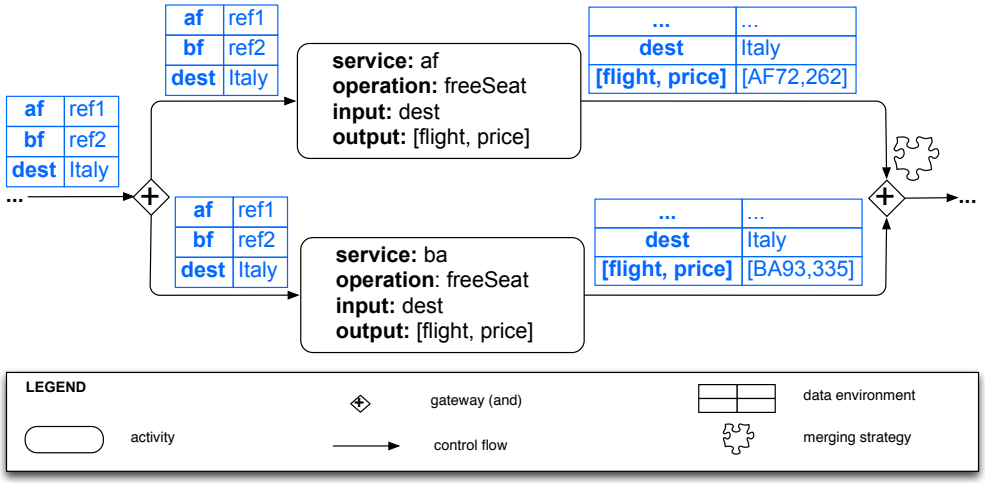


Figure 4.4: Merging strategies for synchronisation patterns.

As is shown in the example in Figure 4.4, when a Parallel Split is encountered, the data environment is conceptually duplicated for each outgoing branch. Further adaptations of the data environment are local to each branch. However, when a

synchronisation point is reached, the data environments from all incoming branches are merged. We have identified several possible merging strategies that are useful in different cases:

- Prioritise one of the incoming branches when resolving conflicts.
- Pick the data environment of one incoming branch and ignore the others.
- Merge conflicts into a collection containing the different values (with or without duplicates).
- Remember the "scope" from before splitting and restore it (not always desirable).
- Employ a programmer-defined function to resolve conflicts.

The small workflow depicted in Figure 4.4 represents a scenario where a booking agent looks for a free seat on a flight to Italy. This is achieved by invoking the services of two airline companies (Air France and British Airways). The result of this service invocation is a *tuple* containing both the flight number and the price. Therefore, the output variables are also specified as a tuple, namely `[flight, price]`.

When both results are received (i.e., when the activities have finished their execution), their outgoing data environments need to be merged. When the first merging strategy is used, and Air France has a higher priority than British Airways, the resulting data environment is the data environment that is passed from the Air France activity to the Synchronization pattern. The second merging strategy will just randomly pick one of those two data environments. The third strategy merges all values of a variable into a collection. When the third merging strategy is used, the resulting data environment contains the following binding: `[ [flight, price], [[AF72, 262], [BA93, 335]] ]`. As we can see, the value for the tuple `[flight, price]` is a container of two tuples. The fourth strategy results in the data environment with only one variable binding `[dest, Italy]`. When the fifth merging strategy is used and the user-provided function is to *take the flight with the lowest price*, the resulting data environment is the one where `price` equals 262.

By introducing this *environment passing style*, we satisfy the key requirements of a data flow mechanism in a workflow model, as stated by Sadiq [SOSF04].

- **Manage both the input and output data of activities.**
  The first requirement is fulfilled, since the actual parameters are looked up in the data environment before starting the execution of an activity. After this execution, the output values are associated with their variable names and added to this dictionary.

- **Ensure that data produced by one activity is available for other activities.**
  By introducing the notion of an environment, which flows through the entire workflow, we ensure that the second requirement is satisfied.
- **Ensure consistent flow of data between activities.**
  The last requirement is fulfilled because of the passing of data in sequence patterns, and the merging strategies we provide in case of multiple incoming branches. The different branches of a split pattern have their own separated data environments which can possibly be merged at specific synchronisation points.

4.3 Patterns for Service Orchestration

Now that we have explained how activities are executed, we describe how these activities can be linked together in order to form a workflow description. The *control flow perspective* defined by van der Aalst et al. [RtHvdAM06] categorises 43 patterns which enable descriptions of the control flow dependencies between several activities.

These 43 patterns are a result of the Workflow Patterns Initiative was established with the aim of delineating the fundamental requirements that arise during business process modelling on a recurring basis and describing them in an imperative way. We adopt the definition and semantics of the control flow patterns that was given by van der Aalst et al. [RtHvdAM06], except for the fact that the execution of the activities can result in the invocation of a *mobile* service. Therefore, the invocation of services must be realised through sending asynchronous messages, as we explained in Section 4.1. Just as activities, the result of executing a control flow pattern is a (possibly) updated data environment. When a control flow pattern is started with a data environment, its components (activities and/or patterns) are executed in the correct order, and the updated data environment is returned as a result.

Basic Control Flow Patterns	
Sequence	standard
Parallel Split	standard
Synchronization	synchronisation
Exclusive Choice	standard
Simple Merge	synchronisation
Advanced Branching and Synchronization Patterns	
Multi-Choice	standard
Structured Synchronizing Merge	synchronisation
Multi-Merge	synchronisation
Structured Discriminator	synchronisation
Blocking Discriminator	synchronisation
Cancelling Discriminator	synchronisation
Structured Partial Join	synchronisation
Blocking Partial Join	synchronisation
Cancelling Partial Join	synchronisation
Generalised AND-Join	synchronisation
Local Synchronizing Merge	synchronisation
General Synchronizing Merge	synchronisation
Thread Merge	synchronisation
Thread Split	standard
Multiple Instance Patterns	
Multiple Instances without Synchronization	standard
Multiple Instances with a Priori Design-Time Knowledge	standard
Multiple Instances with a Priori Run-Time Knowledge	standard
Multiple Instances without a Priori Run-Time Knowledge	standard
Static Partial Join for Multiple Instances	synchronisation
Cancelling Partial Join for Multiple Instances	synchronisation
Dynamic Partial Join for Multiple Instances	synchronisation
State-based Patterns	
Deferred Choice	standard
Interleaved Parallel Routing	standard
Milestone	standard
Critical Section	standard
Interleaved Routing	standard
Cancellation and Force Completion Patterns	
Cancel Task	standard
Cancel Case	standard
Cancel Region	standard
Cancel Multiple Instance Activity	standard
Complete Multiple Instance Activity	standard
Iteration Patterns	
Arbitrary Cycles	standard
Structured Loop	standard
Recursion	standard
Termination Patterns	
Implicit Termination	NA
Explicit Termination	standard
Trigger Patterns	
Transient Trigger	trigger
Persistent Trigger	trigger

Table 4.1: Control Flow Patterns

We make the distinction between standard patterns, synchronisation patterns, and trigger patterns. The distinction between standard patterns and synchronisation patterns is necessary because synchronisation patterns need to define a merging strategy that specifies how the data environments must be merged (Section 4.2). The last category of patterns, namely the trigger patterns, are different from the former two as these patterns can be manipulated externally: They react upon external events.

In Table 4.1 we enumerate all 43 control flow patterns into the eight categories that were proposed by van der Aalst et al. [RtHvdAM06]. The second column in the tables is used to divide the patterns into one of our proposed categories: standard, synchronisation, or trigger pattern. In the remainder of this section we use this distinction to describe how the different categories can be composed.

4.3.1 Standard Patterns

The first category of control flow patterns we define is the so-called "standard patterns" category. This group contains control flow patterns that do not require any special precautions concerning data flow, and cannot be triggered by external events. In this section we discuss how these standard patterns can be composed and how the data flow mechanism that we described in Section 4.2 works for this group of patterns.

In order to explain the composition of (standard) patterns, we introduce an example that combines a Sequence and a Parallel Split pattern. Before describing the example, we give the definition (according to [RtHvdAM06]) of the patterns involved:

- **Sequence**: "*A task in a process in enabled after the completion of a preceding task in the same process.*"
- **Parallel Split**: "*The divergence of a branch into two or more parallel branches each of which executes concurrently.*"

Figure 4.5 depicts the composition of a Sequence and a Parallel Split pattern. This small workflow models the scenario where a plane has landed and activities must be executed in parallel. In this example the ground crew is reminded to bring the jet bridge, and in parallel the trailer that is in charge of the luggage of that flight must remove the luggage from the plane and transport it to a luggage belt.

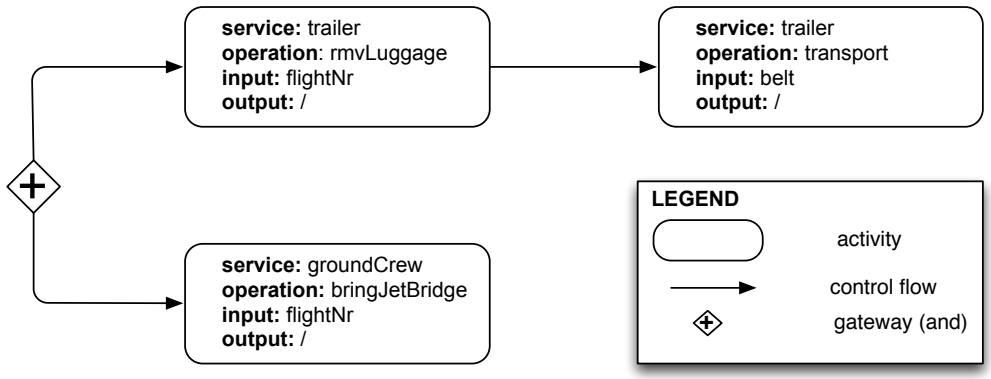


Figure 4.5: Basic control flow patterns: Parallel Split with a Sequence in its first branch.

The three activities in this small workflow example are activities that do not need to discover a service. For each activity a reference is available in the data environment (that is passed to the activity at runtime).

Figure 4.6 shows the sequence diagram of an instance of this workflow example. When the Parallel Split is started with a data environment, this data environment is copied and used to start the execution of the pattern's outgoing branches, namely a Sequence pattern, which is started with `Data Environment 1`, and an activity (called `Activity3`), which is started with `Data Environment 2`. The execution steps of an activity are the same as we mentioned in Section 4.2: the actual parameters of the invocation are looked up in the data environment, the service is invoked[1], and the output variables are bound to their variables and inserted in the data environment.

In the sequence diagram, we see that the execution of the Sequence pattern starts the execution of its first activity (`Activity1`). Only when the execution of that activity is finished, i.e., when an updated data environment is returned as a result of that execution, the execution of the second activity can be started with that updated data environment.

The execution of the Parallel Split pattern itself is finished at the moment all its outgoing branches, 2 branches in this example, have returned an updated data environment as the result of their execution. Note that the data environment that is returned as a result of the execution of the Parallel Split pattern is a new data environment. Split patterns, such as the Parallel Split pattern, frequently precede a synchronisation pattern, which merges the data environments. However, the example we present in Figure 4.5 does not combine the split pattern with a syn-

[1]Recall that in this example, no service discovery needs to be performed.

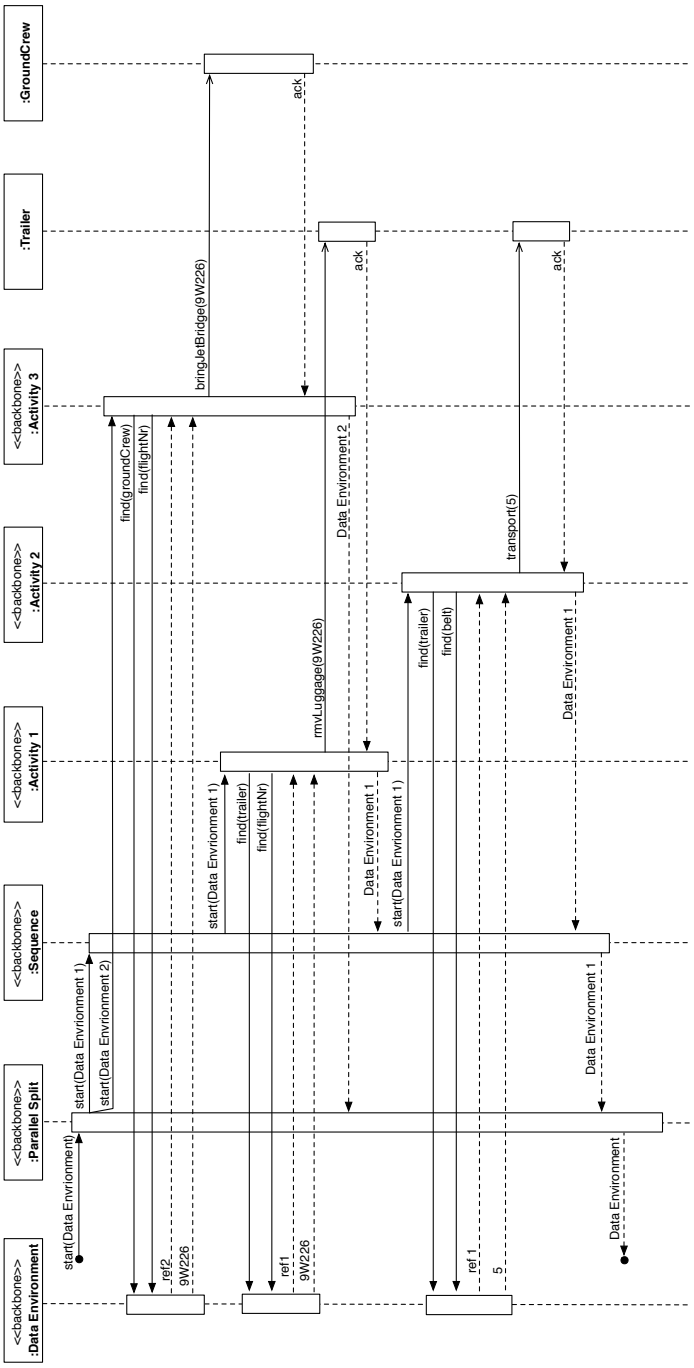


Figure 4.6: Sequence diagram of the execution of the workflow depicted in Figure 4.5.

chronisation pattern, denoting that the data environments do not need to be merged. Therefore, in this case *a* (random) data environment is returned to mark that the execution is finished.

We now discuss an interesting type of standard patterns, namely the patterns that are classified by van der Aalst et al. [RtHvdAM06] as "multiple instances patterns". Multiple instance patterns describe situations where there are multiple threads of execution active. Each of these instances relates to the same activities and patterns, and hence share the same workflow definition.

Consider the example where the boarding pass of all passengers who checked in needs to be verified before going on board. When this task is executed for all passengers, the boarding personnel is notified to close the gate. The workflow modelling this example is depicted in Figure 4.7.

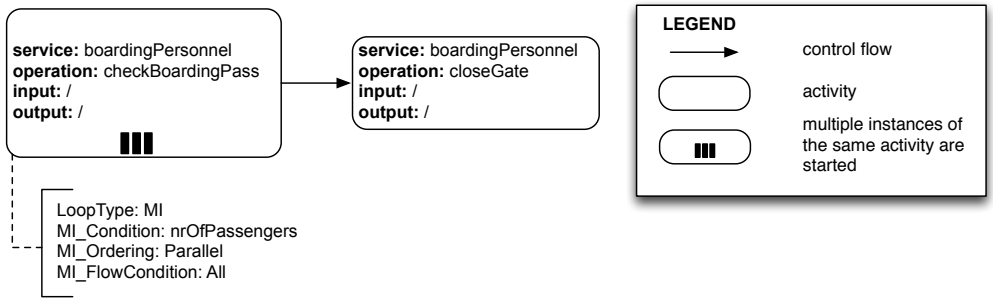


Figure 4.7: Example of a multiple instances pattern.

As we explained in Section 4.2, each time a control flow pattern is started, a new data environment is instantiated and passed to that instance. Hence, in order to support multiple instances patterns, it suffices to start each instance with a copy of the passed data environment with a new unique id.

4.3.2 Synchronisation Patterns

The second category of control flow patterns contains the so-called "synchronisation patterns". This category consists of patterns that have several incoming branches that need to be merged.

Synchronisation patterns distinguish themselves from standard patterns because they require the specification of a data merging strategy. This merging strategy is necessary to merge the data environments of the synchronisation pattern's incoming branches, as we explained in Section 4.2.

Synchronisation patterns also differ from standard patterns, because these patterns have multiple incoming branches. These incoming branches are not necessarily originating from the same preceding split pattern (such as the Parallel Split pattern). Moreover, it is also possible that the branches of a split pattern are merged by several *different* synchronisation patterns. Before presenting a scenario of such a workflow example, we give the definition of the patterns involved. We already gave the definition of the Sequence and Parallel Split pattern (see Section 4.3.1), which we now extend with the following definition (according to [RtHvdAM06]):

- **Synchronization**: "*The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.*"

Consider a scenario where several things need to happen before a plane can take off. First of all, catering must supply the plane with the necessary food and beverages, and the luggage of the passengers must be loaded. Furthermore, before passengers can board, the plane must be cleaned and a flight attendant must check whether all personnel are on board. Figure 4.8 shows a workflow diagram for this example scenario.

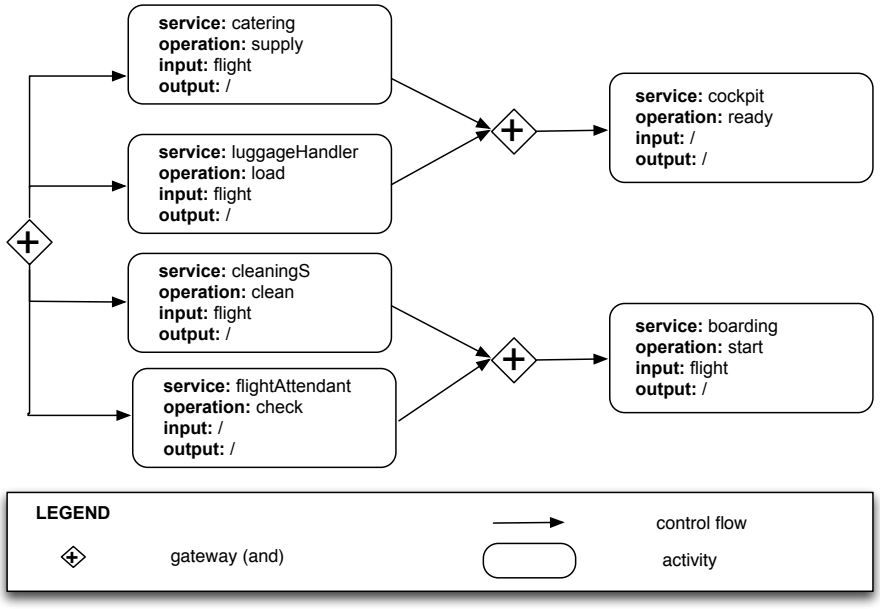


Figure 4.8: Basic control flow patterns: Parallel Split followed by multiple synchronisation patterns.

In this example scenario, the services that need to be addressed are all *registered services*. Hence, the services do not need to be discovered as a reference is available in the data environment that is used to start the workflow.

Figure 4.9 shows the sequence diagram of part of the workflow depicted in Figure 4.8. We omitted the details of the actual service invocation and only show the starting of the activity and the data environment that is returned as a result of executing an activity. Furthermore, we only show the first activity (catering) and the third activity (cleaning service) of the example. For conciseness, we also omitted the retrieval and insertion of variables in the data environment.

When the parallel split is started, the data environment is passed to its outgoing branches (only two in the sequence diagram). The outgoing branches of the Parallel Split pattern are Sequence patterns consisting of the activities of the branches followed by a Synchronization pattern. When the Sequence pattern is started, it starts its first activity, and when the execution is finished (i.e., when the data environment is returned), the Sequence pattern starts its second component, namely the Synchronization pattern.

When all incoming branches of the Synchronization pattern have been enabled (i.e., when the pattern is started as much time as it has incoming branches), the data environments are merged and the resulting data environment is returned as a result. Note in the diagram in Figure 4.9, we only show two out of four outgoing branches of the Parallel Split pattern.

When the data environment is returned as a result of the execution of the Synchronization pattern, the execution of the Sequence pattern is finished because all its components have finished their execution. As a result, the data environment is returned as a result of its execution. The Parallel Split pattern's execution is finished once it has received the data environments of all its outgoing branches (i.e., of all Sequence patterns).

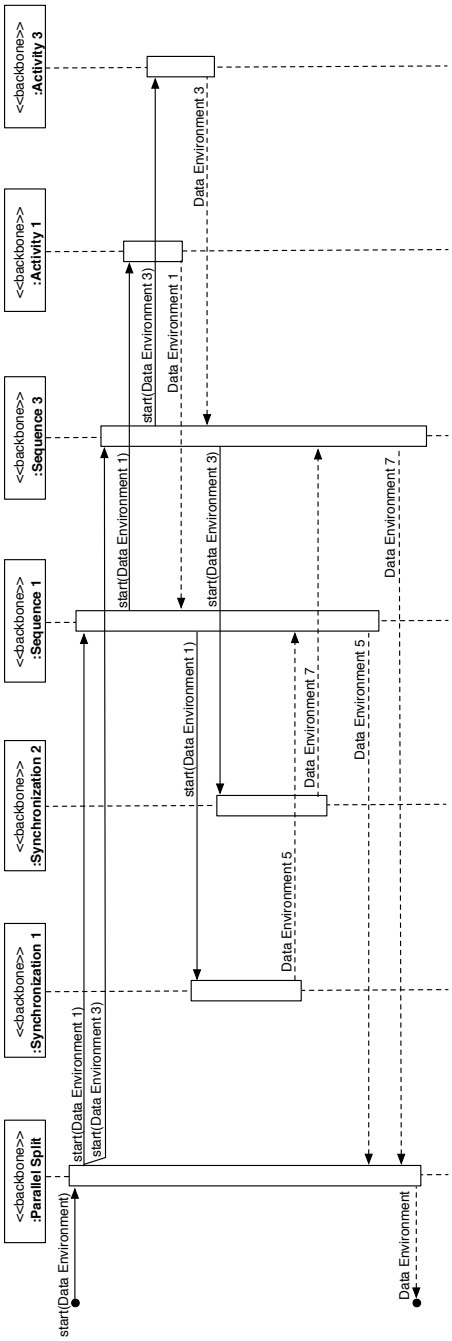


Figure 4.9: Sequence diagram of the execution of the workflow depicted in Figure 4.8.

4.3.3 Trigger Patterns

We discuss a third category of control flow patterns, namely "trigger patterns". The category of trigger patterns deals with workflow components that require an external signal in order to start their execution. van der Aalst et al. [RtHvdAM06] define two patterns for this category, namely the *Transient Trigger* and *Persistent Trigger* pattern.

These group of patterns behave similarly to standard patterns concerning how data flow is handled. These categories of patterns differ however, because trigger patterns must be able to react upon receiving external events. In this section we discuss one of those patterns, namely the Persistent Trigger, which is defined as:

- **Persistent Trigger**: "*The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task.* "

We introduce a small example where the execution is triggered by the reception of an external event. Consider the scenario where the pilot is waiting for the final signal in order to take off. When the jet bridge has been removed, the pilot is waiting from a signal of the ground crew in order to start moving the plane towards the runway. When the plane departs, the steward on board can start explaining the emergency procedures. A workflow description modelling this scenario is depicted in Figure 4.10.

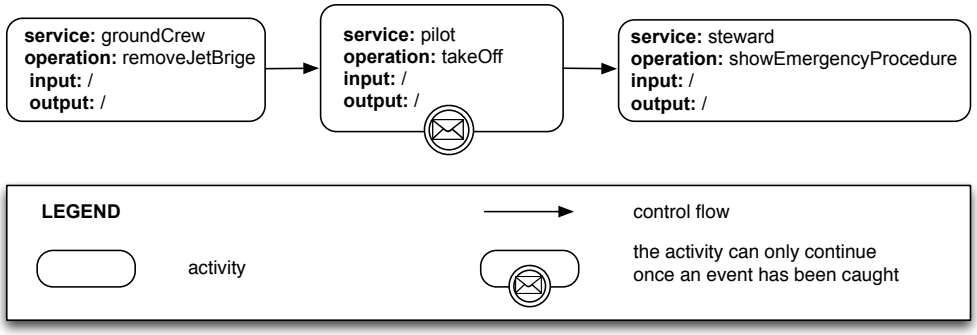


Figure 4.10: Trigger patterns: Sequence pattern using a Persistent Trigger pattern.

Figure 4.11 depicts the sequence diagram for this workflow. For conciseness we omit the actual service invocation and depict only the start of the activities and the data environment that is returned as a result of the activities' executions. For the same reason, we also do not depict the retrieval and insertion of variables in

the data environment.

The execution of this workflow is started by passing the data environment to the Sequence pattern. This pattern executes its components (activities in this example) consecutively. When the first activity has finished its execution, i.e., the data environment is returned as a result, the second activity's execution is started with that data environment. As we can see in the figure, the execution of the second activity is waiting for the reception of a *signal*. Once such a signal has been received, the activity can start its computation (perform a service invocation), and return the data environment as a result.

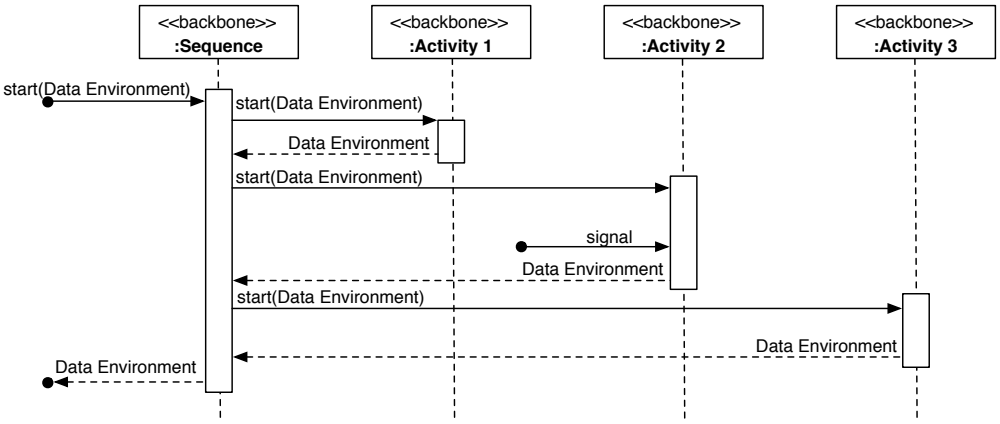


Figure 4.11: Sequence diagram of the execution of the workflow depicted in Figure 4.10.

4.4 Patterns for Group Orchestration

The increasing popularity of mobile devices fosters the omnipresence of services in mobile environments. Software systems in a mobile environment often want to manage a set of services that form a logical group and orchestrate the execution of a particular process for all its members. To orchestrate a group of services, abstractions are required which allow control over the execution in a way that transcends the individual process of a single member. In this section we present high-level abstractions for group orchestration as a new set of workflow patterns. The description of these abstractions adheres to the categories and criteria we presented in Section 2.3.4.

In the previous sections of this chapter we used examples that are inspired by the iMPASSE application we presented in Section 2.3.1. This scenario merely

focusses on the orchestration of services, which addresses both the control flow and data flow of an application. In this section we base the examples upon the more sophisticated SURA application. The scenario we described in Section 2.3.1 introduced several group-related concepts, such as the need to intensionally describe the services the process will interact with. The SURA application also exemplifies the need for addressing a group of services and synchronising the application for all these services.

4.4.1 Definition of Group Membership

1) Intensional Definition

The description of the services can be either achieved by enumerating all of them (extensional description) or by describing all properties those services must fulfil (intensional description). Deciding which services to interact with in a dynamically changing environment is hard when reasoning extensionally about it, as the set of services can vary over time. In these kinds of environments it is opportune to provide intensional descriptions for those services. Intensional descriptions abstract away the precise number of services during interaction and let services maintain anonymity during this interaction. A group should be able to specify a description such that not only services of the same type, like temperature services, but also more sophisticated characterisation of members can be achieved. In particular, there is a need for intensional descriptions of services such as "*the service I last used*", "*my favourite service*", or "*all temperature services that are nearby and have an accuracy of more than 95%*". The service types that can be used to describe a service (see Section 4.1) is one way of specifying intensional descriptions. In order to support more complex intensional descriptions, we use the logical coordination language CRIME [MSP+07].

We choose for CRIME because this logic coordination language offers a full fledged first order predicate logic. Moreover, this language is specifically sculpted towards networks where volatile connections are present. We did not chose for semantic web languages, such as OWL [BvHH+04], since these languages are based upon description logics and are not as expressive as logic query languages. Semantic web languages are however useful to define hierarchical relations of the world. Inspired by this idea, the logic coordination language CRIME has been extended with the notion of lists and records such that composition and structuring of contextual data are supported [Vas07]. Therefore, JSON syntax [Cro06] is introduced to specify the meaning of attributes and the interrelation between them.

By introducing records in CRIME context information can be defined in a generic, structured way that is flexible and adaptable to the constantly changing parameters and the context information described by the record itself.

CRIME The Fact Space Model [MSP+07] of CRIME provides a logic coordination language for reasoning about context information that is represented as facts in a *federated fact space*. Concretely, facts are locally published by applications and transparently shared between nearby devices as long as they are within communication range. The Fact Space Model equips applications with at least two fact spaces, a "private" fact space, and one or more interface fact spaces. Facts residing in the private fact space are not shared, whereas facts residing in an interface fact space are exchanged with other applications in connection range.

Applications have the ability to react upon the appearance of facts, by making use of rules. The conditions to adapt an application are described by making use of the logic coordination language CRIME of which the rules record the causal link between facts and the conclusions that may be drawn from them. These links are used in the Fact Space Model to reverse the effects a fact had on the system, when the fact is retracted. In the Fact Space Model both the assertion and the retraction of facts have consequences. Because facts are retracted when the device that published them disconnects, the Fact Space Model offers fine-grained support to deal with the effects of this disconnection.

The logic language uses the forward chaining strategy for deriving new conclusions as this data-driven technique is very suitable for the event-driven nature of CRIME. Because we describe workflow patterns that allow orchestration in nomadic networks where lots of events (connections, disconnections, etc.) occur, using CRIME is beneficial.

The federated fact space used by CRIME ensures that the view of an application on its environment is kept consistent by translating changes in the environment into the assertion or retraction of facts shared by colocated devices. As we already mentioned, applications have the ability to react on changes in their environment by specifying rules:

```
1 private<-room(plane, silent).
2 private<-room(airportBuilding, general).
3
4 :switch(?profile),
5 profile(?profile) :-
6    public<-location(myID, ?room),
7    private<-room(?room, ?profile).
8
9
```

```
10 :switch(default) :-
11    not profile(?p).
```

Listing 4.1: CRIME: facts and rules to change the profile of a mobile phone.

The rule shown in Listing 4.1 is used to change the profile of a mobile phone. Consider an air hostess who wants the profile to change automatically to "silent" when she is on a plane, and to change to "general" when she is in the airport building. The CRIME rule above implements this behaviour:

- **Definition of facts**: On line 1 and line 2 we define two facts which specify the desired profile for a certain location. As can be seen in the example, facts can be *quantified* to denote the fact space in which they should be found or asserted. Both facts are published in the `private` fact space, meaning that these facts are not interchanged with fact spaces residing on nearby devices.
- **Definition of rules**: Rules can be seen as a mapping from (a combination of) facts onto a conclusion. Conclusions may consist of the addition of new facts to the fact space or the execution of application-specific actions (preceded by a colon symbol). Application-specific actions are written in the base language implementing the Fact Space Model (i.e., JAVA).
  We define a first rule on lines 4-7. This rule has two prerequisites (lines 6-7), which state that a fact of type `location` must be published in the `public` fact space, and that a fact of type `room` must be available in the "private" fact space. Moreover, the values of the variable `room` must be identical for both facts. The rule is triggered when the mobile phone enters a location for which the user has made her preferences clear (in the example, a plane or the airport building). Two conclusions need to be made for this rule (lines 4-5): when the rule is triggered, the application-specific action `switch` is executed and a private `profile` fact is added. Note that the `location` facts are not defined in Listing 4.1. These types of facts are published by the infrastructure of the airport and shared with nearby devices.
  The second rule (defined on lines 10-11) is used to ensure that whenever no desired profile is specified for a certain location, the "default" profile is chosen.

As we already mentioned, CRIME is sculpted to deal with the characteristics of a mobile environment. One of these characteristics is that connections are volatile, therefore intermittent disconnections happen frequently. Recall that facts from a co-located device are retracted when the device is disconnected and reinserted when it reconnects at a later time. Although care has been taken to optimise the matching phase of the inference engine by making use of the Rete algorithm [For82],

such removals and reinsertions of facts remain relatively costly. Therefore, CRIME implements an optimisation, known as "scaffolding the Rete network", specifically geared towards the way the coordination language deals with volatile connections. By introducing causal links, constant time retractions are supported. Small scaled test have shown that this optimisation performs significantly better than the normal Rete algorithm [SP07]. Where the normal Rete algorithm slows down when used extensively, the optimised version does not.

Now that we introduced CRIME, we present how this logic coordination language can be used to describe a group of services.

Service Descriptions We support both types of group descriptions: extensional and intensional. First of all, it is possible to instantiate a group with a set of objects. These objects can be either references to the services (e.g., the fans of festival's headliner), or just plain objects (for instance, all ids of those fans). Additionally, we allow intensional description of the group members by either using a service type or by writing a logical expression.

Listing 4.2 shows an example of an intensional description expressed using a logical expression. This logical expression is triggered for all facts `festival_visitor` for which a fact `fan_info` with the same `id` is available. Moreover, the `band` should correspond to the band for which the fact `band_info` states that it is the `headliner`.

```
festival_visitor( ?id ),
fan_info( ?id, ?band ),
band_info( ?band, "headliner" ).
```

Listing 4.2: Intensional description of a group.

In Figure 4.12 we depict the federated fact spaces of two fans and the federated fact space residing on the festival's infrastructure who are connected in a mobile ad hoc network. Those fact spaces consist of *quantified facts* which denote the fact space the fact belongs to. The different kinds of fact spaces we support are "private", "public" and "shared". Private facts are not exchanged between colocated devices, whereas public ones are. In order to limit the exchange of facts between colocated devices, we added a third type of fact space "shared". Facts that are published in this fact space are only exchanged to fact spaces who have subscribed to that type of facts. In our example, the infrastructure of the festival is interested in a lot of information of the fans, for instance, the facts of type `festival_visitor` and `fan_info`, whereas fans are, in general, not

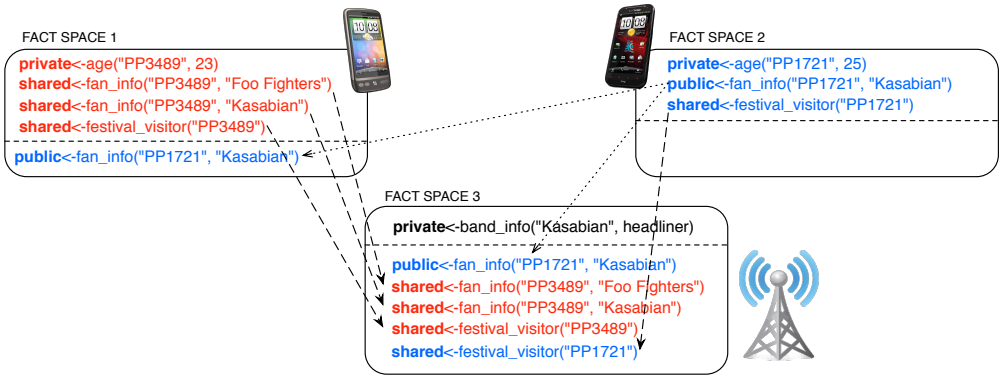


Figure 4.12: Federated fact spaces of colocated devices.

interested in those published facts of other fans. Therefore, the facts that are published in the *shared fact space* are only asserted in the federated fact space of the infrastructure (`fact space 3` in Figure 4.12).

2) Arity Decoupling

Now that we have presented how the members of a group can be described, we explain how a group can be modelled. We introduce the notion of a *Group pattern* which consists of

- **description** of the group members;
- **variable name** to refer to an individual group member;
- **sub workflow** that must be executed for all group members.

Figure 4.13 depicts a Group pattern, which has "all fans of the headliner" as its description and the symbol "fan" as its variable name. The sub workflow that is wrapped by the Group pattern consists of a sequence of several activities, of which only the first one is shown.

The notation that is used to denote a Group pattern is loosely based upon the BPMN [Obj11]. The upper part of the notation is used for the group's description and its variable name, whereas the part at the bottom is used to wrap a sub workflow. The latter part of the notation is based upon BPMN's group which is used to group graphical elements that are within the same category.

Now that we have introduced the Group pattern, we describe how the execution of this pattern takes place. When the group pattern is started, first all services satisfying the group's description need to be retrieved. Once these services are retrieved, the incoming data environment is copied for each of these services. This

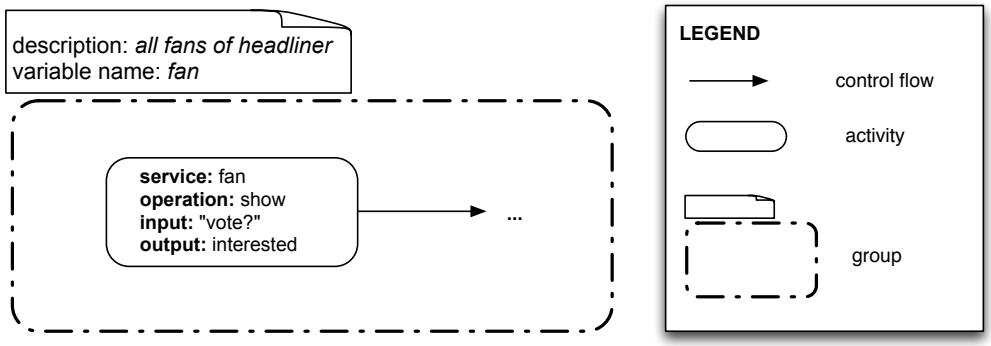


Figure 4.13: Example of a Group pattern.

way, each member of the group has its own data environment where local changes can occur. In order to access the specific service (member) for which the sub workflow is executed, a reference to the service is added in the data environment used to start each individual instance of the sub workflow. Afterwards, the sub workflow that is wrapped by the Group pattern, is started with each of these data environments. Note that each instance has its own copy of the sub workflow (for more details we refer to Chapter 6). This is depicted in Figure 4.14.

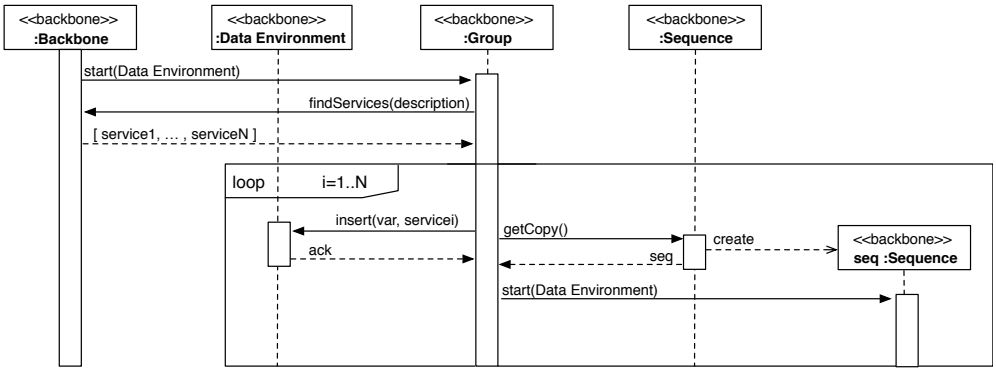


Figure 4.14: Sequence diagram of the execution of the workflow depicted in Figure 4.13.

Please note that the sequence diagram in Figure 4.14 is not complete: we do not show what happens when the execution of the sub workflow (the `Sequence` pattern) is finished.

Once the execution of the Group pattern is started, it is possible that the backbone discovers a new service that satisfies the group's description. In this case, a new member is added to the group and the sub workflow wrapped by the group

is started once more. Note that this only applies when an intensional description (type tag or logical expression) was given. However, sometimes this behaviour is not wanted. Therefore, we introduce the notion of a *snapshot group* [VC08], where the number of services communicated with is fixed. Unlike a normal group, such a snapshot group does not allow new members to join the execution once started. Note, however, that once the snapshot group is made, members of the group can still disconnect, upon which can be reacted in an appropriate way, as we explain in Section 4.5.

3) Dynamic Modification

It is possible to redefine the members of a group when its execution is ongoing. For instance, it is possible to restrict the members of a group by filtering out those members that do not satisfy a certain condition. The *Filter pattern* only allows the instances who satisfy the given condition to continue their execution.

Recall the nomadic application, called SURA, that addresses a group of fans at a festival (see Section 2.3.1). The application is initially executed for all fans of festival's headliner. However, after being asked if they are interested in participating, the members of the group are restricted to only those whom expressed their enthusiasm. This is depicted in Figure 4.15, where the condition of the filter will verify the value of the variable `interested` in the data environment.

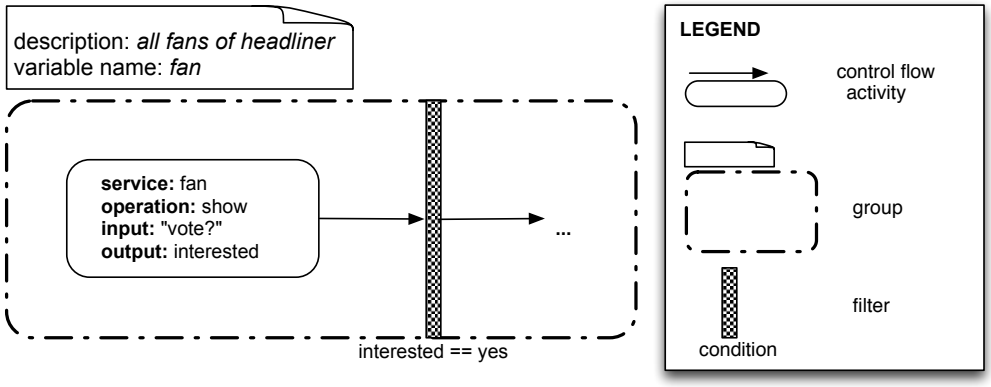


Figure 4.15: Restricting the members of the group by using a filter.

4.4.2 Synchronisation Mechanisms

The introduction of group orchestration gives rise to more advanced synchronisation patterns. The inherent volatile connections of the network cause communica-

tion partners to disconnect making full synchronisation not always possible. For instance, synchronisation should be able to succeed when only partial results are returned (after the first result, after a number of results, after some time, etc.). In this section we elaborate on the different synchronisation mechanisms needed to orchestrate a group in a nomadic network. All of these synchronisation mechanisms do not only have an influence on the execution of the sub workflow for each group member, they are also reflected on the group itself.

In this section we describe four patterns that enable group synchronisation, namely a *Barrier*, *Cancelling Barrier*, *Group Join*, and *Synchronised Task* pattern. Those patterns have the following criteria in common:

- A condition is given to specify *when* the synchronisation may succeed. All individual instances of which the execution was blocked, can continue their execution the moment the condition is satisfied.
- When the given condition is fulfilled, it is possible that instances that reach the synchronisation at a later point in time should not continue their execution. Therefore, it should be possible to state whether or not a *cancellation* should take place.
- It is possible that when the condition is fulfilled, a specific task (sub workflow) needs to be executed *once*.
- When such a one-time task is specified, a merging strategy can be defined to specify which *data* must be available during the execution of that task.

In the remainder of this section, we first elaborate on the conditions that are used to instantiate group synchronisation patterns. Thereafter we describe four specific synchronisation patterns which implement (a subset of) the criteria mentioned above.

1) Conditions used by Group Synchronisation Patterns

All group synchronisation patterns are instantiated with a certain condition, such as *"after 10 seconds"*, *"when all instances have succeeded"*, or *"when 90% of the instances have succeeded"*. Such a condition can be either classified as a *time constraint*, a *quota constraint*, or a combination of both. We define two different kinds of time constraints, namely a deadline and a duration constraint. A deadline is a condition which is fulfilled at a certain moment in time, whereas a duration constraint is fulfilled a predefined time after the synchronisation pattern is reached for the first time. We also distinguish two kinds of quota constraints: *percentage* and

amount. The amount and percentage condition take as argument a number and are satisfied when that number, or that given percentage of instances respectively, has reached the synchronisation. It should be possible to combine the above conditions using logical expressions, such that it is possible to have a constraint like *"when one instance has reached the synchronisation pattern, or after 60 seconds"*. This list of constraints is not complete: other user-defined constraints are possible.

2) Barrier and Cancelling Barrier Pattern

We now present two novel patterns that allow synchronisation of individual instances of group members by blocking their execution until a specified condition is fulfilled. When the execution of an individual instance of a group member reaches a *Barrier*, the condition of the pattern is verified. When the condition is not yet fulfilled, the execution of that instance is blocked. At the moment the condition is fulfilled, all instances that were blocked resume their execution. Figure 4.16 shows a Barrier pattern with a time constraint.

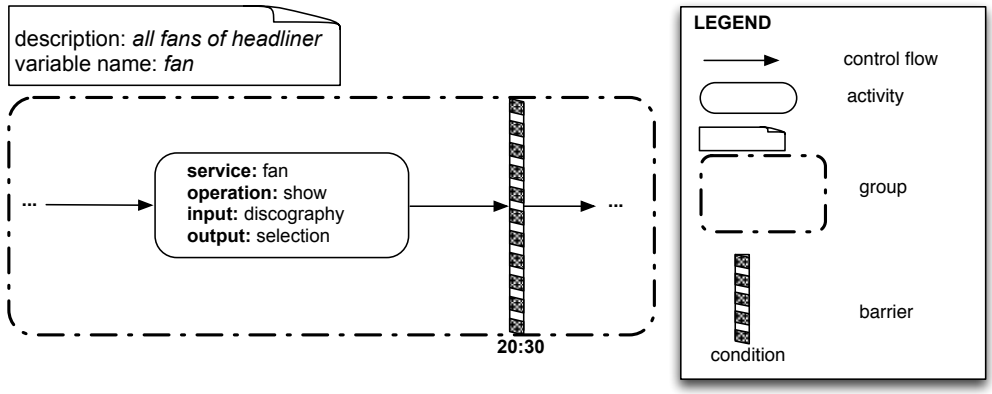


Figure 4.16: Synchronisation: the Barrier pattern.

The difference between a (normal) Barrier and a *Cancelling Barrier* pattern is explained by the way they treat instances that arrive at a barrier of which the condition is already fulfilled (i.e., the "cancellation" criteria mentioned earlier). Individual instances of members that arrive later at a normal barrier continue their execution without waiting. On the other hand, when a Cancelling Barrier pattern is used, only the blocked instances will execute the remainder of the workflow after the barrier. The execution of instances that reach the Cancelling Barrier pattern after these blocked instances' execution is restarted are cancelled.

Remark that a Cancelling Barrier pattern, by definition, has an influence on the amount of members of the group. Once the condition of the Cancelling Barrier is fulfilled, the blocked instances of the individual members continue their execution. From that moment on, the amount of members of the group is restricted to those that were able to continue their execution.

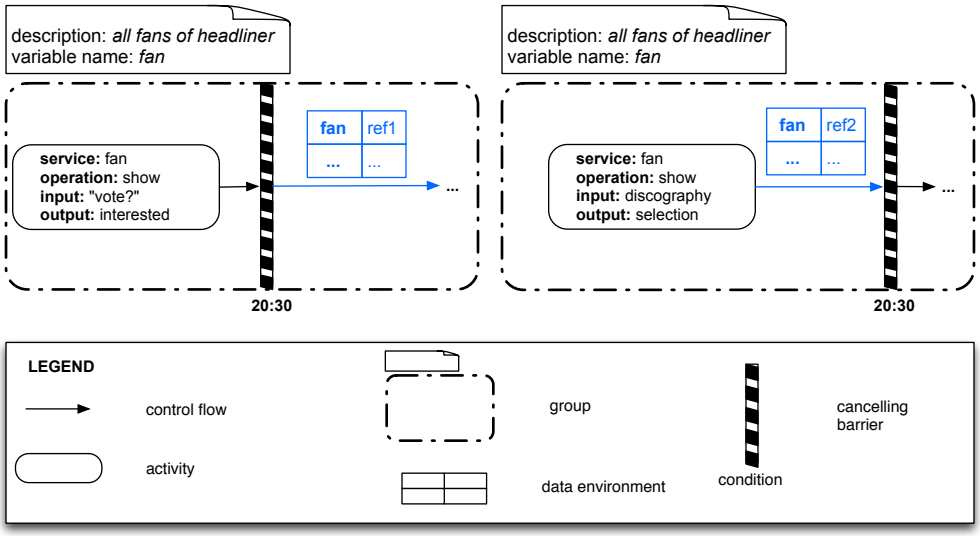
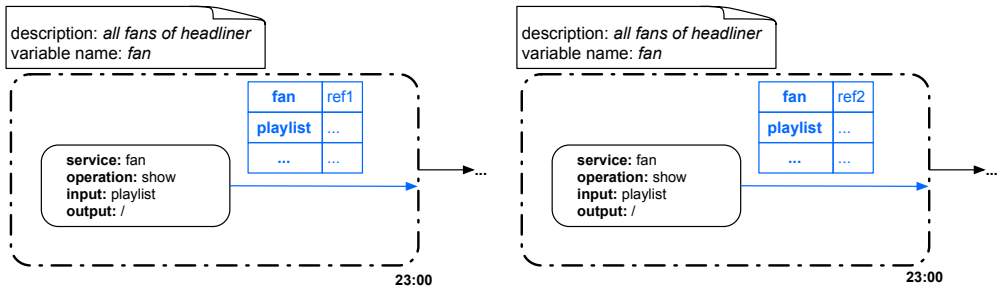


Figure 4.17: Synchronisation: execution of a Cancelling Barrier pattern.

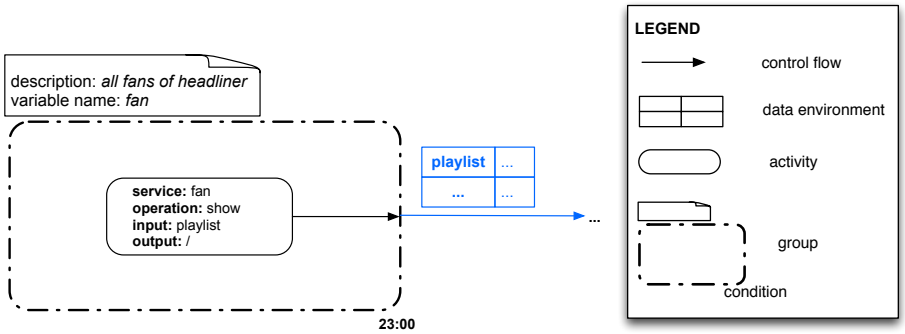
In the SURA application, fans of the headliner are allowed to vote for their favourite songs until half past eight. All votes that are received afterwards are not taken into consideration. Figure 4.17 depicts this part of the application, which is modelled using a Cancelling Barrier pattern. This figure shows the execution of this workflow for two instances (i.e., for two fans). As we can see in the picture, the first instance (corresponding the service with reference `ref1`) has already passed the Cancelling Barrier pattern, and a second instance just reaches the pattern. However, as the Cancelling Barrier's condition is already satisfied and the blocked instances (amongst other the first one, depicted on the left hand side) have already continued the execution of the sub workflow after the pattern. Therefore, the execution of the second instance is terminated.

3) Group Join Pattern

In order to terminate the execution of the group pattern both control flow and data flow must be merged. We introduce a *Group Join pattern* that allows managing how control flow and data flow are merged. The default merging strategy used



(a) Execution of two instances of individual group members that reach the Group Join pattern.



(b) After the execution of the Group Join pattern.

(c) Legend.

Figure 4.18: Synchronisation: a Group Join pattern to terminate the group.

to merge the data environments of all instances is "accumulating all values for each variable", one of the merging strategies proposed in Section 4.2. However, it is often wanted to specify another merging strategy, which can be done upon instantiation of the Group pattern.

A Group Join pattern is instantiated with both a condition and a merging strategy. Once this condition is fulfilled, the control flow of all individual instances is merged such that the remainder of the workflow pattern after the Group pattern is executed only once. Therefore, a Group Join pattern is, by definition, always cancelling, meaning that instances that reach the pattern after its condition is fulfilled, will not be able to continue their execution.

In Figure 4.18 we show how both control flow and data flow are merged to terminate the execution of a group pattern. Figure 4.18(a) depicts two running instances of individual group members where both members have reached the end of the Group pattern. The workflow depicted in Figure 4.18(b) represents the state

of the execution where the Group Join pattern's condition (in this example the deadline `23:00`) is met and the execution of the Group pattern is finished. As we can see in the figure, only one single instance continues the execution of the remainder of the workflow, with a merged data environment without the variable name, used to instantiate the Group pattern. Just as synchronisation patterns (see Section 4.3.2), a merging strategy specifies how the data environments must be merged (for instance, by accumulating all values for a single value in a collection).

Note that we do not depict the group join pattern explicitly, as it is the only place the pattern is allowed.

4) Synchronised Task pattern

In this section we present the notion of a *Synchronised Task*, a sub workflow that only needs to be executed once for the entire group at a specific moment. In essence, each member of the group executes its own instance, but there should be provisions to allow a single task to be executed *once* for several or all members of the group. Therefore, a Synchronised Task is, by definition, always cancelling, meaning that instances that reach the pattern after the condition is fulfilled, will not be able to continue their execution.

A Synchronised Task starts the execution of its wrapped sub workflow once a given condition is fulfilled. In Figure 4.19 we show the functionality of the Synchronised Task, that is depicted as a grey box wrapping a sub workflow (in this example, one single activity). This figure shows the workflow that models part of the SURA application: sending the accumulated votes of all fans to the headliner at nine o'clock. Upon receiving all collected votes, the headliner decides upon the final playlist, which is sent to all fans who voted.

Figure 4.19(a) depicts two running instances of individual group members (i.e., two fans), which have both reached the Synchronised Task pattern. Because the Synchronised Task pattern's condition is not yet fulfilled (the deadline `21:00` is not expired), the instances are blocked. Once the condition is met, the execution of the blocked instances can continue. To this end, the data environments of those instances need to be merged (i.e., all votes `selection` need to be accumulated), and the variable name that was used to instantiate the Group pattern needs to be removed.

The resulting data environment is used to start the execution of the sub workflow that is wrapped by the Synchronised Task pattern, as is depicted in Figure 4.19(b). The sub workflow, a single activity in this example, shows the accumulated votes to the headliner who decides upon the final playlist.

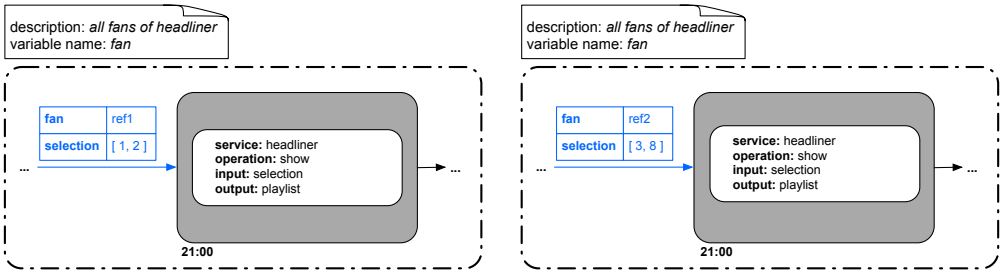
In Figure 4.19(c), we see that the Synchronised Task's sub workflow is terminated. Before continuing the remainder of the workflow after the Synchronised Task, the incoming data environments of the instances of the individual group members need to be restored, and updated with additional variable bindings. In this concrete example, a new binding is added for the variable `playlist`.

Figure 4.19(d) depicts the state of the workflow where the two instances of the individual group members execute the remainder of the workflow after the Synchronised Task pattern with their own data environment. As we can see, these data environments are the ones of Figure 4.19(a) with an extra variable binding for the variable `playlist`.

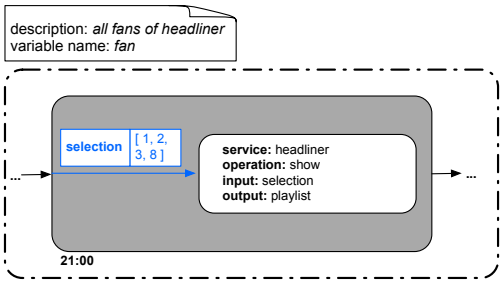
4.4.3 Relation to Existing Research

van der Aalst [RtHvdAM06] describes *multiple instances patterns* which wrap part of a process that needs to be instantiated multiple times. These patterns are supported by YAWL [vdAtH05]. In YAWL it is possible to add new instances during the execution of the "multiple instances without a priori run-time knowledge" pattern. However, the synchronisation mechanisms that are supported by YAWL are rather restricted. The only synchronisation mechanism YAWL supports, is used to terminate the execution of the multiple instances pattern. There are no mechanisms like the barriers we propose provided to synchronise the execution of all instances inside the multiple instances pattern. Moreover, there is no support for the synchronised task abstraction we presented. This behaviour can be modelled in YAWL by defining a multiple instances pattern, followed by an activity or sub workflow preceding a second multiple instances pattern. The disadvantage of this approach is that contextual information, i.e., the data for which the first multiple instances pattern was started, is discarded. By introducing a Group pattern and more advanced synchronisation patterns, specifically sculpted towards group orchestration, modelling workflows for a group of services is more straightforward.

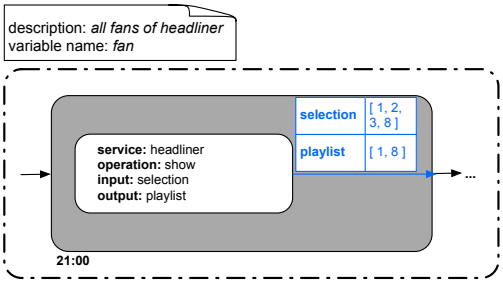
In the web services community, the notion of a *service group* [MSB12] is introduced to denote a heterogeneous collection of web services that satisfy a given constraint. Such a service group only consists of fixed web services that are known beforehand (by means of a URL). Arity decoupling is supported as services can be added or removed from the service group, causing the service group registration to notify requestors of modifications to that service group. This abstraction is similar to the Group pattern we described in this chapter, but is not suited to function in dynamic networks where services are not necessarily known a priori. Moreover, no synchronisation mechanisms are provided to control the execution of such a service group.



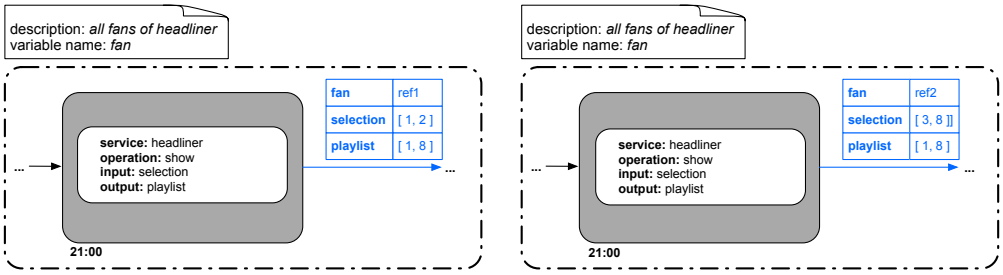
(a) Execution of two instances of individual group members that reach the Synchronised Task pattern.



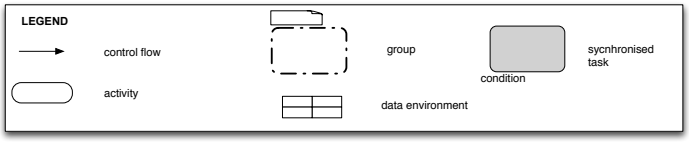
(b) Execution of the Synchronised Task pattern (part I)



(c) Execution of the Synchronised Task pattern (part II)



(d) Execution of two instances of individual group members after the execution of the Synchronised Task pattern.



(e) Legend.

Figure 4.19: Synchronisation: a Synchronised Task pattern executing a task once for all instances.

Ambient References [VDM+06] enable communication with a volatile group of proximate objects by means of asynchronous message sends. Ambient references are developed as a programming language abstraction for AMBIENTTALK [VMG+07]. This language construct satisfies two of our criteria related to group orchestration, namely intensional definitions (criterion 5) and arity decoupling (criterion 6). Ambient references provide synchronisation mechanisms by providing observers that are triggered either when the first service has answered or when all services have responded. However, as communication with the set of services is expressed by means of a single message send, redefinition of group members and synchronised task abstractions do not make sense. Moreover, there are no mechanisms provided to express failure handling on an ambient reference, for instance, expressing the action that must be performed when a service disconnects.

4.5 Patterns for Failure Handling

In a dynamically changing environment, the challenge is to make the large heterogeneity of services co-operate and deal with their transient and permanent failures. Before introducing the patterns for failure handling we ought necessary for orchestration in nomadic networks, we describe the possible types of failures and the necessity for the automatic handling of failures.

4.5.1 Automatic Failure Handling

Orchestrating services can always cause failures, such as exceptions during a service invocation or an error of the service. However, when dealing with services in nomadic networks, the high dynamicity of the network gives rise to network failures as well. Therefore, the following failures can happen frequently: a service is unavailable, a service is unresponsive, or a service does not respond in time.

In Figure 4.20 we show an updated version of the lifecycle diagram of an activity (that we presented in Figure 4.1). As we can see in Figure 4.20, three new states are added:

- **Service unresponsive**: when a service is not online or disconnects during interaction with it;
- **Service timeout**: when the result of the service invocation is not received within a limited time period;
- **Service exception**: during the service invocation, the service invocation can throw an exception.

The transitions between the different states that are coloured in blue are the default compensating actions to recover from these types of failures.

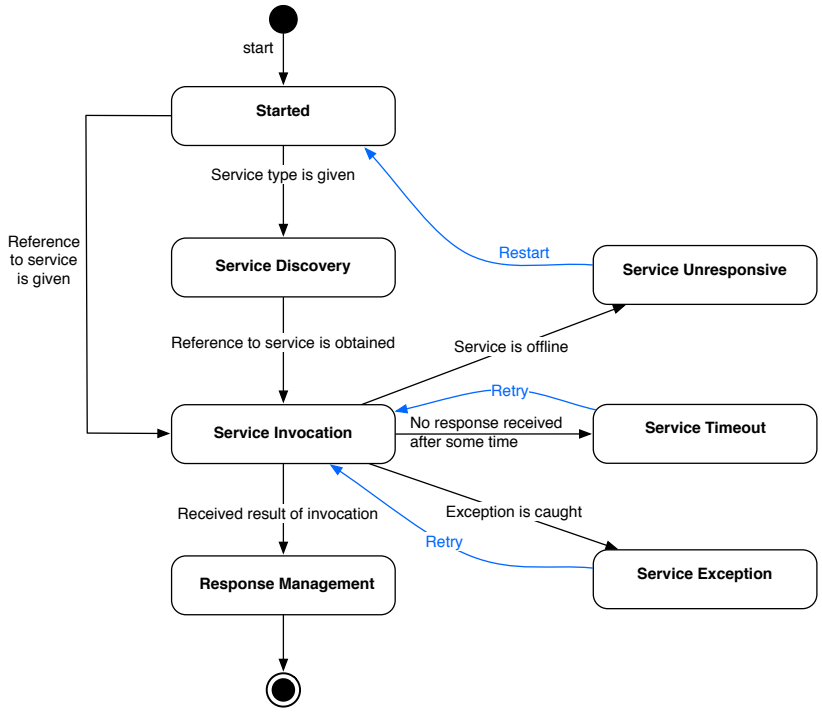


Figure 4.20: Lifecycle of an activity (version II).

- **Restart**: This compensating action will restart the execution of the activity.
- **Retry**: This compensation will retry to invoke the same service again.

In order to use these compensations for failures, the services must be stateless. Note that a compensating action is not always successful, which can lead to indefinitely trying to compensate for a certain type of failure. Therefore, we may want to limit the number of times a compensating action is tried. Moreover, sometimes it is opportune to provide other (more drastic) compensating actions than the ones that are executed by default. To this end, we introduce patterns that allow to detect specific kinds of failures and the specification of (a chain of) compensating actions to overcome those failures.

4.5.2 Specification of Compensating Actions as a Failure Handling Mechanism

We introduce a *Failure* pattern that wraps part of a workflow and imposes compensating actions and strategies. Since we want to handle (transient) failures at different levels of granularity, the failure pattern can be used on one specific

activity or wrap an entire sub workflow. Russell et al. [RtEv05a] already classified workflow exception patterns used by workflow systems. The exceptions he discusses are, for instance, *constraint violation*, *deadline expiry* and *work item failure*. The failures we support are specific to the (temporary) network failures that can arise, although some basic exception handling can be achieved by using the *exception* failure. Examples of events we capture in a failure pattern are disconnections, reconnections, timeouts and possibly service exceptions/errors, as explained in Section 4.5.1.

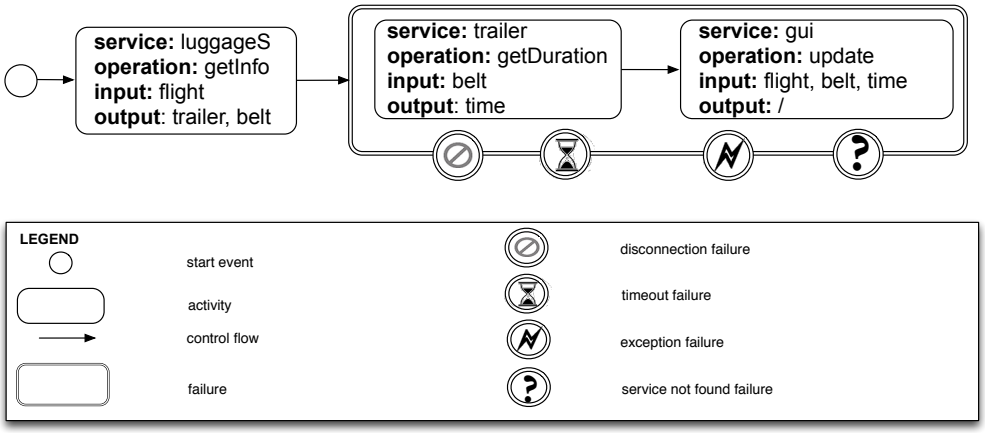


Figure 4.21: Failure pattern.

Figure 4.21 shows a workflow example where a Failure pattern is used to wrap a sub workflow (the second and third activity of the Sequence pattern). In order to avoid any confusion, remark that the notation that is used to depict a Failure pattern resembles the BMPN [Obj11] notation for a *transaction*. However, the Failure pattern we define, does not share the definition of such a BPMN transaction.

Using this Failure pattern, specific types of failure events (disconnection, timeout, exception, service not found) can be captured. For each of these failure events, compensating actions can be specified to override the default failure handling behaviour. In Section 4.5.1 we already explained three possible compensating actions, namely rediscover, restart and retry. Further possible compensations include:

- **RestartAll**: This compensating action restarts the *entire* wrapped sub workflow.
- **Skip**: This compensation just skips the failed activity.
- **SkipAll**: This compensation skips the *entire* wrapped sub workflow.
- **Replace**: This action replaces a failed activity by executing a sub workflow.

- **Alternative**: This compensation replaces upon failure of one activity the *entire* wrapped sub workflow by executing another sub workflow.
- **Wait**: This compensating action is used in combination with one of the other compensations, and will simply wait for a specified time before trying the next compensating action.

These compensating actions can be divided in two categories, namely the compensations that only have an effect on the failed activity and the ones that have an effect on the entire wrapped sub workflow. The compensations that only affect the failed activity are: Rediscover, Restart, Retry, Skip, Replace. The compensating actions RestartAll, SkipAll, and Alternative have an effect on the entire sub workflow that is wrapped by the Failure pattern.

Chaining Actions to Handle Failures Application-Specifically

Because compensating actions are not always successful, we provide a way of limiting the number of times each compensating action is tried. When a compensating action has reached its maximum attempts, another (more drastic) one can be tried. Consider the example of updating a screen at the airport with the necessary information (flight number, belt number, estimated duration before luggage is on the belt) so that passengers can retrieve their luggage. By using the Failure pattern, application-specific compensating actions (instead of the default actions Rediscover and Retry) can be specified, as is shown in Figure 4.22.

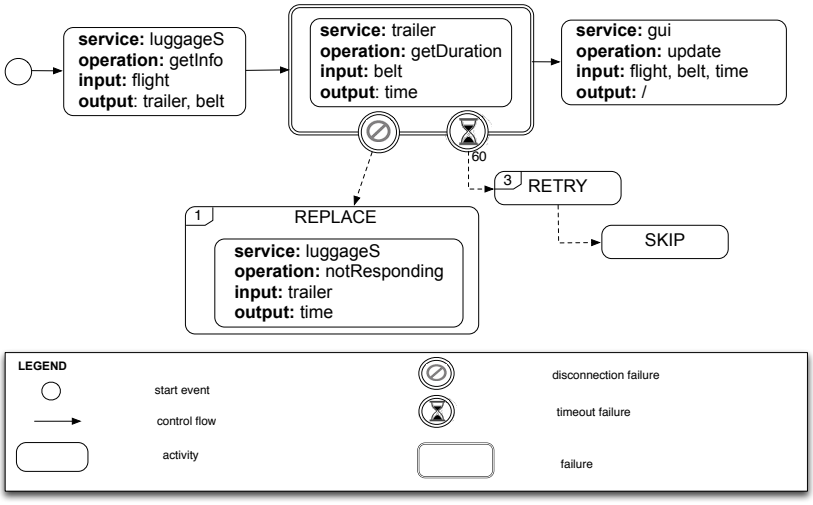


Figure 4.22: Failure pattern: overriding default failure handling strategies with more specific compensating actions.

The second activity of the Sequence pattern that is depicted in Figure 4.22 is wrapped with a Failure pattern. This Failure pattern is used to specify more specific compensations in case a disconnection or timeout failure occurs. When the second activity has a timeout (after 60 seconds no reply is returned), we try to resend the message three times. If this is still unsuccessful we move on to the next compensating action, which just skips this activity (so no time gets displayed on the screen). In case of a disconnection, however, the failed activity is replaced with another sub workflow (in this example, a single activity). The activity that replaces the failed activity will contact the luggage service and inform them that the trailer is not responding. This luggage service will give an estimated duration (calculated based on the location of the trailer), such that the third activity of the Sequence pattern is able to show an estimated time on the screen.

Please note that in this example compensating actions are only specified for two types of failures, namely for a disconnection and a timeout. However, the default compensations still apply: for example, when an exception occurs during interaction with the trailer, the service will be invoked again (the compensation Retry is executed). Moreover, these default compensating actions apply for all the activities in the sub workflow: not only the three activities of the Sequence pattern, but also the activities used by compensating actions (for instance, the activity of the Replace compensation).

Nesting of Failure Patterns to Enable More Specialised Failure Handling

Failure patterns can be nested, so different strategies can be formulated on different levels of granularity. A whole workflow can be surrounded by a failure pattern specifying “after a disconnection, wait 20 seconds and then try to rediscover" and smaller parts of this workflow can be wrapped with more specific failure patterns, which possibly override (shadow) the behaviour imposed by the outermost failure pattern. An inside-out policy is used to determine the compensations that must be executed for an occurred failure event.

Figure 4.23 shows the nesting of Failure patterns. In this example, more accurate behaviour is required in case an exception occurs. When an exception is caught during the execution of the first activity of the Sequence pattern, the compensating action that is executed is the Alternative compensation. This action will put the message “Technical problem” on the screen next to the luggage belt. However, when an exception is caught during the execution of the second or third activity of the Failure pattern, another compensation is executed. In that case, the invocation of the service is retried after waiting for 60 seconds. Remark that in this particular example, no number is used to indicate the number of attempts that should be

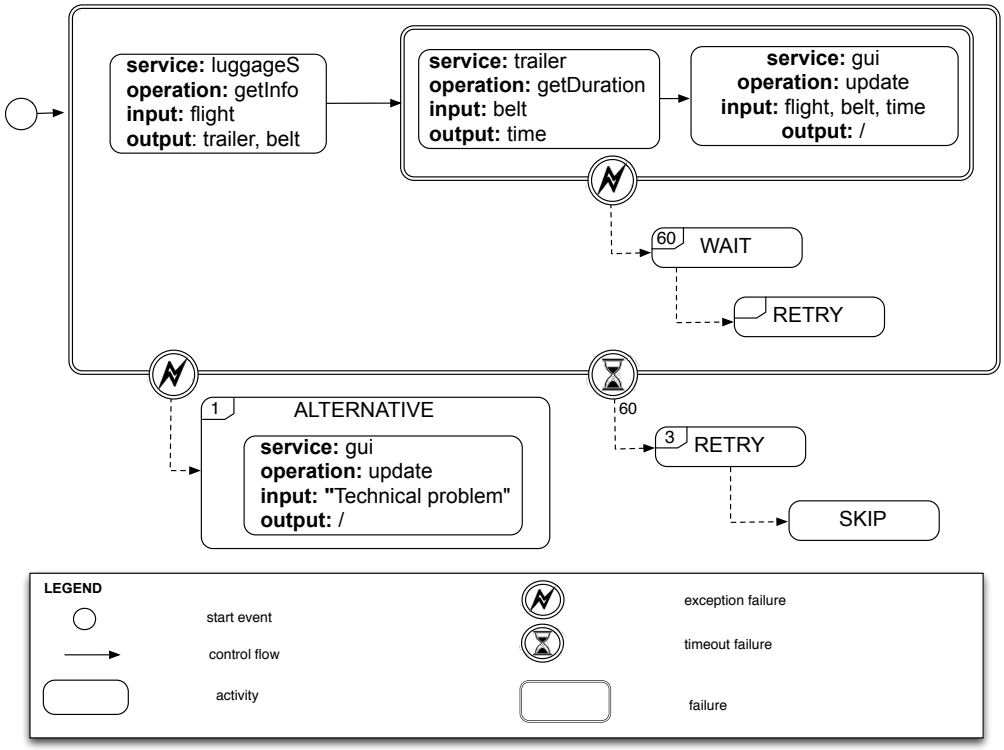
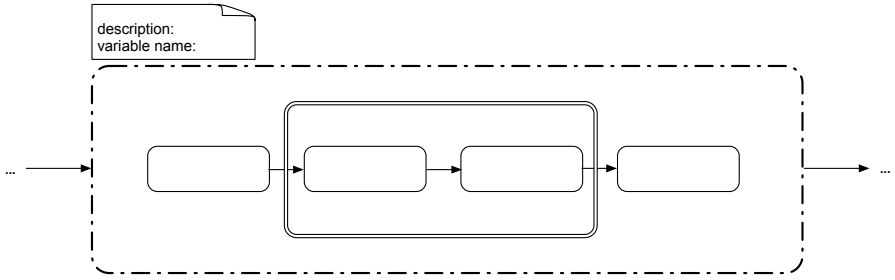


Figure 4.23: Nesting of Failure patterns to acquire more accurate failure handling strategies.

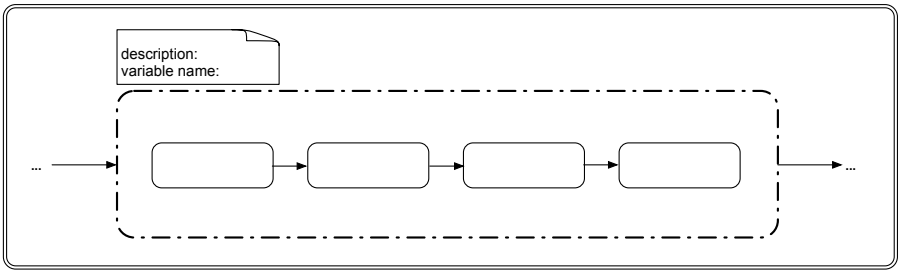
executed, which will result in indefinitely retrying to invoke the service that threw the exception.

4.5.3 Failure Handling for Group Orchestration

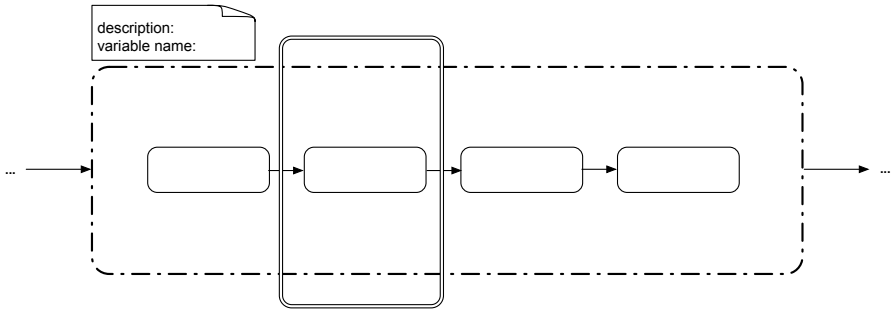
As mobile environments are liable to volatile connections, ways to detect and handle failures must be available. In this section we present the influence of this failure handling mechanism on the novel abstractions for group orchestration we presented in Section 4.4. We distinguish three different compositions for failure handling with respect to group orchestration, which we depict in Figure 4.24.



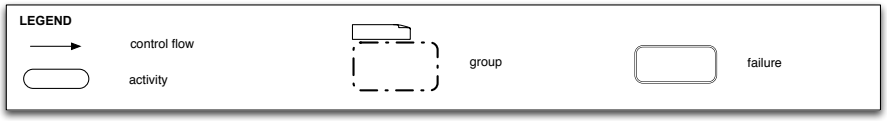
(a) Composition 1: Failure pattern inside the Group pattern and wrapping (part of) the Group pattern's sub workflow.



(b) Composition 2: Failure pattern wrapping the Group pattern.



(c) Composition 3: Failure pattern outside the Group pattern and wrapping (part of) the Group pattern's sub workflow.



(d) Legend.

Figure 4.24: Different compositions of failure handling for group orchestration.

We summarise the effects of compensating actions, depending on the composition of failure handling for group orchestration that is used. Recall that compensating actions can be divided into two categories:

1. **Compensations - Category 1**: This category contains the compensating actions that only affect the execution of *the failed activity*. This category consists of the compensations Rediscover, Restart, Retry, Skip, and Replace.
2. **Compensations - Category 2**: This category of compensating actions contain those compensations that affect *the entire wrapped sub workflow*. The compensating actions that are included in this category are RestartAll, SkipAll, and Alternative.

In Table 4.2 we describe the effect these compensating actions have, depending on the composition that is used. For each of these compositions, and each of the categories for compensation, we describe what happens when a failure is detected during the execution of an activity that is part of the group's sub workflow.

Composition	**Compensations Category 1**	**Compensations Category 2**
Composition 1 Figure 4.24(a)	The compensation only affects *the failed activity* (and hence, only *the execution of a single group instance*).	The compensation only has an effect on the execution of *an individual group instance*.
Composition 2 Figure 4.24(b)	The compensation only affects *the failed activity* (and hence, only *the execution of a single group instance*).	The compensation has an effect on *the execution of the entire group* (i.e., on the execution of *all group instances*).
Composition 3 Figure 4.24(c)	The compensation only affects *the failed activity* (and hence, only *the execution of a single group instance*).	The compensation has an effect on *the execution of the entire group* (i.e., on the execution of *all group instances*).

Table 4.2: Effect of a compensating action depending on the composition that is used for failure handling for group orchestration.

Before elaborating on these three different compositions, we introduce new types of failure events and new compensating actions.

1) Dedicated Failure Handling for Groups

We extended the failure handling support in order to distinguish between a failure that occurred during the execution of a group member (service), or another service, such that all failures have a *participant* variant. For instance, we make the distinction between a normal disconnection and a participant-disconnection failure.

Consider the following example scenario: the festival committee wants to organise a poll to inquire the festival visitors about the organisation of the festival. To this end, during the last concert each festival visitor receives this poll on his/her smartphone. The festival committee receives the results of this inquiry once the last band has finished their performance. Therefore, the results of all festival visitors are accumulated before sending. In order to stimulate the festival visitors to fill in the questions, a reward (free drink tickets) is given after the results have been received. This reward is given by sending a special code to the smartphones of the visitors who participated. Figure 4.25 depicts the workflow modelling this scenario.

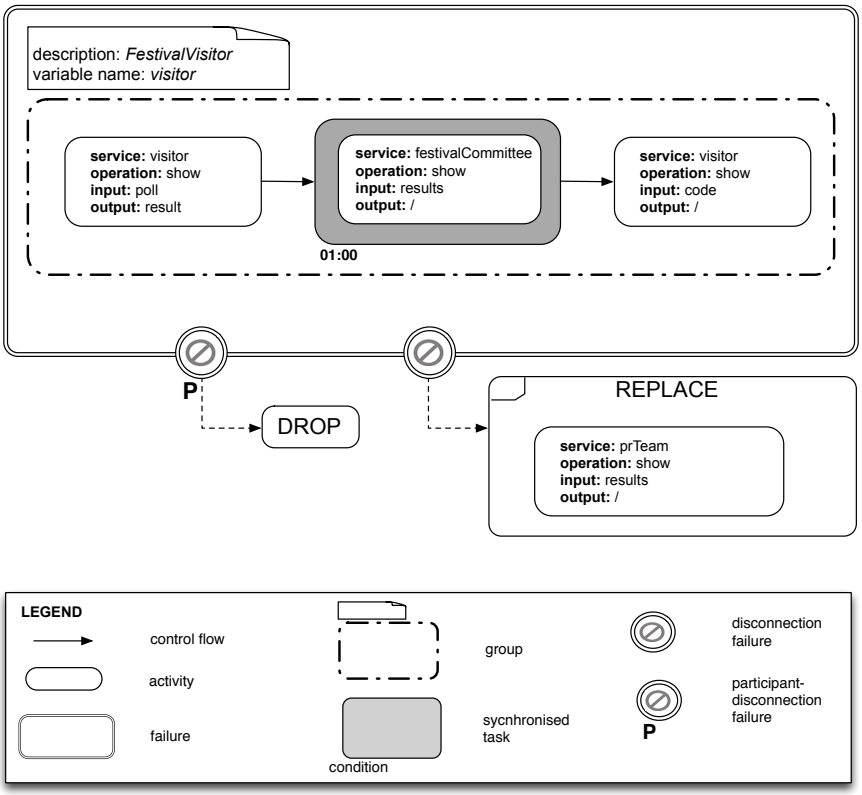


Figure 4.25: Failure detection: normal failures versus participant failures.

In this example scenario, we would like to specify a different compensating action when the mobile device of single visitor disconnects, in contrast to a disconnection of the server of the festival committee.

Figure 4.25 shows the difference between a (normal) disconnection failure and a participant-disconnection failure. When a disconnection happens during the interaction with the mobile device of an individual festival visitor, the compensating action will just drop that festival visitor from the members of the group. On the other hand, when a disconnection occurs when communicating with the server of the festival committee, the compensation will send a message to the members of the festival's PR team.

Besides the participant-failures we also introduce two additional group-specific compensating actions, namely *Drop* and *Wait-and-Resume*. Both compensating actions can *only* be used in combination with a participant-failure (a failure that affects a group member or the communication with that member). The Drop compensating action drops the member from the group. The second compensating action, Wait-and-Resume, can be used in combination with a participant-disconnection or a participant-not-found failure. When such a failure occurs, the activity where that failure occurs is stored, such that the execution can be resumed when that specific group member (re)connects.

These two group-specific compensating actions belong to a dedicated category of compensating actions that affect the execution of the group. In case of the Drop compensation, the number of participants is reduced, and the activities following the failed activity are all cancelled. Hence, the execution of the group is affected, and the Drop compensation also has an influence on the execution of the individual group instance by cancelling the remainder of its execution. The Wait-and-Resume compensating action has an influence on the execution of an individual group instance.

2) Failure Handling for Group Orchestration - Composition 1

In this section we describe the first composition where a Failure pattern is defined inside a Group pattern and wraps (part of) the Group pattern's sub workflow (cf., Figure 4.24(a)). This composition is used when compensating actions must only affect the execution of an individual group instance.

Figure 4.26 depicts the situation where a sub workflow is wrapped by a Failure pattern. In case a disconnection occurs, the compensation RestartAll is executed, and when a timeout is detected, Retry is executed as the compensation. As we

already mentioned, RestartAll is a compensation of category 2 (a compensation which influences the sub workflow that is wrapped by the Failure pattern), whereas Retry is classified as a compensation of category 1 (a compensation which only affects the failed activity).

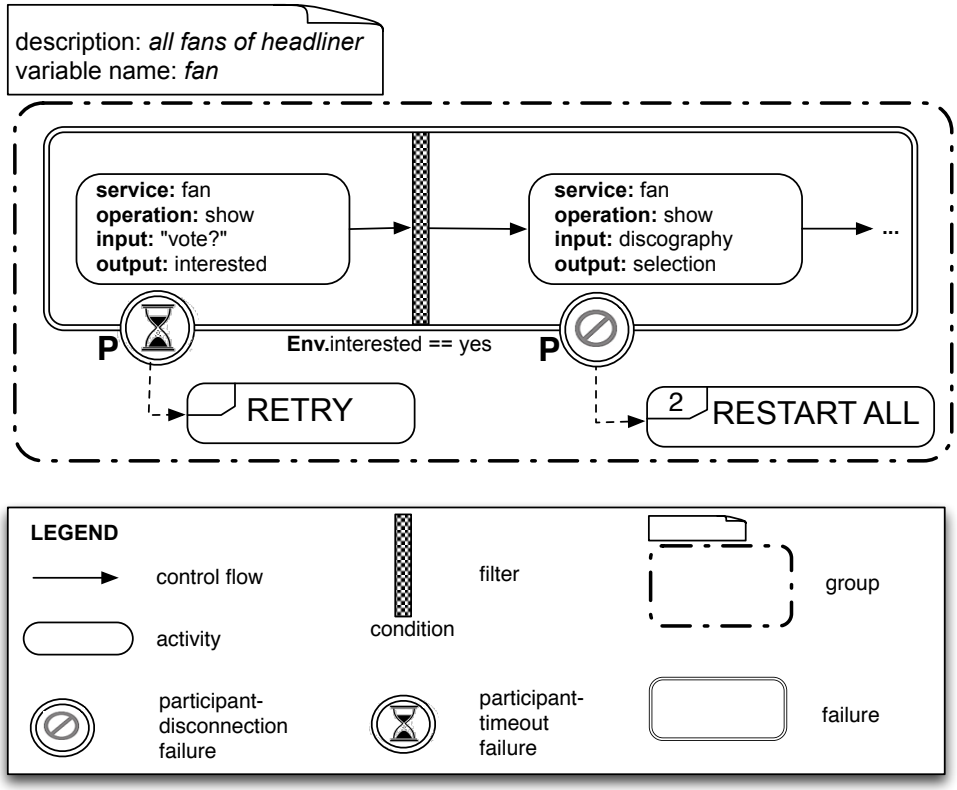


Figure 4.26: Failure Handling for Group Orchestration - Composition 1.

In the example, when a timeout happens during communication with a service, the failed activity is retried, only for the individual group instance for which the failure occurred. In case a disconnection failure occurs, the RestartAll compensating action is executed. Because the Failure pattern is defined inside the Group pattern, this compensation only affects the execution of an individual group instance, i.e., the execution of the sub workflow wrapped by the Failure pattern is only restarted for one instance. This is in contrast with the behaviour that is accomplished when this Failure pattern is defined on the outside of the Group pattern, as we explain in the following section. When using the failure pattern outside the group pattern, compensating actions might (depending on the specific category) affect the group as a whole, as we explain later.

3) Failure Handling for Group Orchestration - Composition 2

It is also possible to wrap the Group pattern itself with a failure pattern (cf., Figure 4.24(b)). When the Failure pattern is defined outside the Group pattern, the compensating actions have either an effect on the execution of an individual instance or on the execution of the entire group, depending on the type of compensation that is specified.

Suppose we adapt the example we used earlier (Figure 4.26) such that the Failure pattern is now placed outside the Group pattern instead of inside (see Figure 4.27). When a disconnection failure occurs, the compensating action restarts the wrapped sub workflow, in this case the Group pattern. So now, all instances of the group would be cancelled and the execution of the entire group would be restarted. So in this case, all fans would be contacted again and asked whether they want to vote.

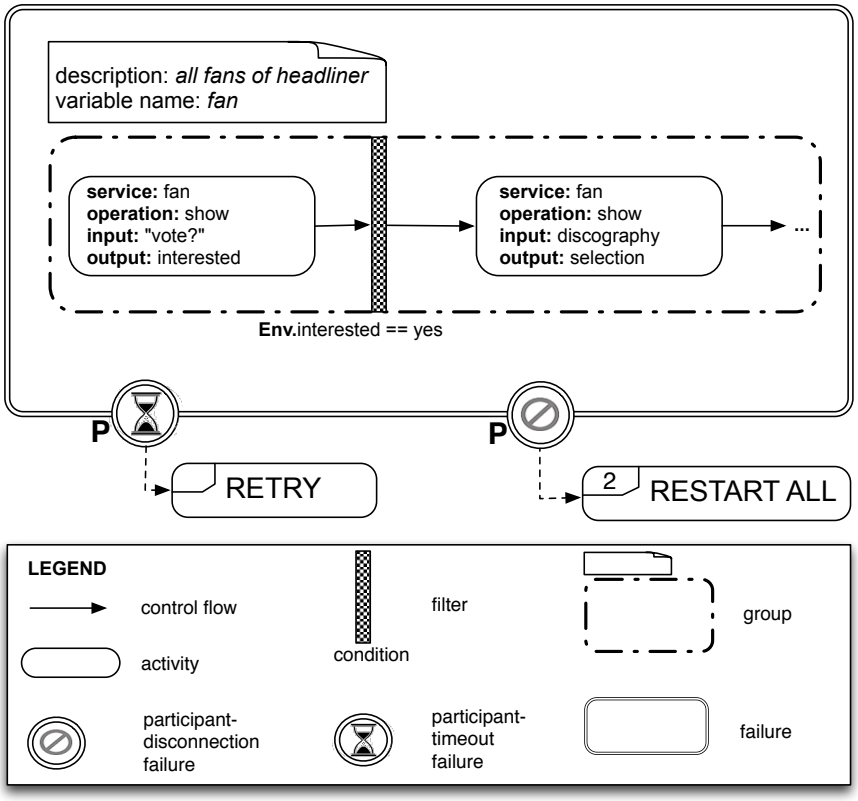


Figure 4.27: Failure Handling for Group Orchestration - Composition 2.

4) Failure Handling for Group Orchestration - Composition 3

This type of composition is a special case of the previous one, and is used when different compensating actions need to be specified for different activities in the Group pattern's sub workflow (cf., Figure 4.24(c)).

Figure 4.28 depicts an example where a Failure pattern wraps a part of the Group pattern's sub workflow. In this example different compensating actions are specified for different activities that are part of the Group pattern's sub workflow. In case a disconnection occurs for the Sequence's first activity, the compensating action Retry is executed. Because Retry is classified as a compensation of category 1, only the execution of a single group instance is influenced.

When a disconnection is detected during the execution of the Sequence's second activity, the compensation that is performed is RestartAll. This compensating action is classified as a category 2 compensating action, and hence it influences the execution of the entire group.

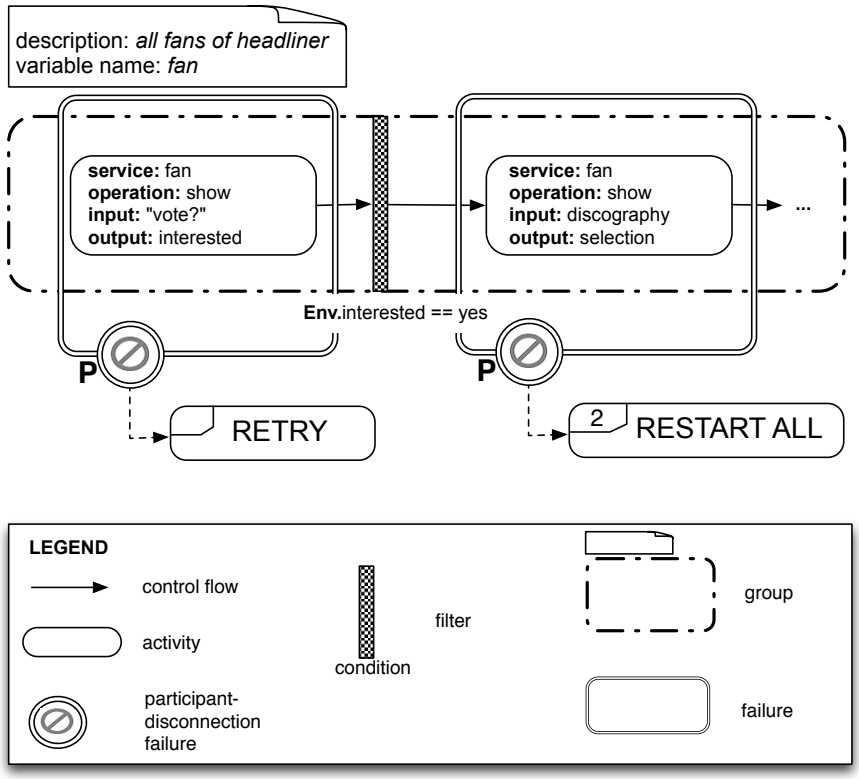


Figure 4.28: Failure Handling for Group Orchestration - Composition 3.

4.5.4 Relation to Existing Research

There exists related work concerning the handling of both expected and unexpected exceptions in workflow systems. The exception patterns proposed by Russell [RtEv05a] (work item failure, work item deadline, external trigger and constraint violation) are already available in workflow systems and business process modelling languages, such as BPMN [Obj11]. Russell et al. [RtEv05a] introduce several primitives for failure handling, both on the level of a single *work item*[2] and on the level of a *case* (i.e., a workflow instance). Possible primitives for handling failures at the work item level are restart, continue-execution, reallocate, force-complete. Figure 4.29 depicts these primitives as transitions between the different states of a work item (i.e., the dotted arcs connecting the states in the lifecycle diagram).

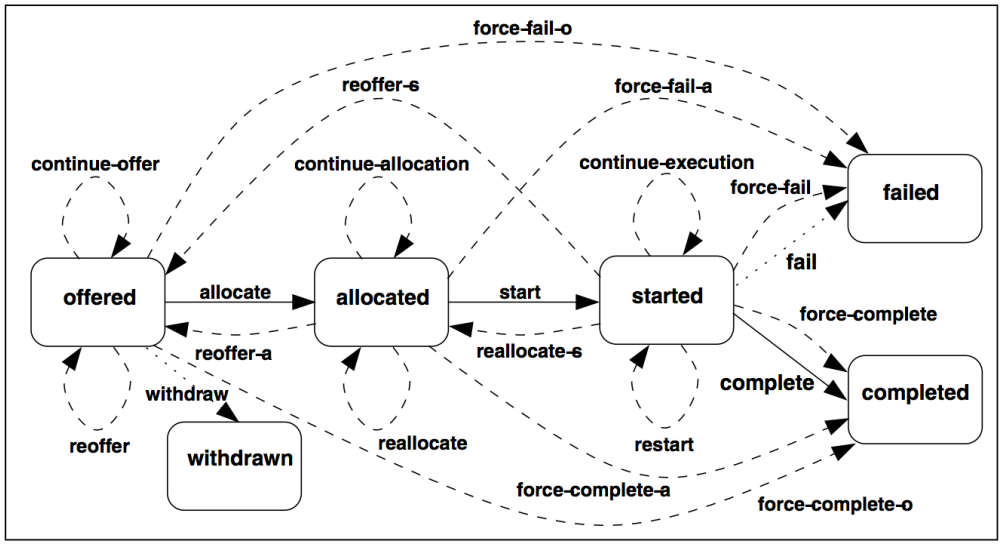


Figure 4.29: Failure handling at the level of a work item [RtEv05a].

The lifecycle diagram presented in Figure 4.29 presents the different states of a work item. We consider a task (activity) as a service invocation, and, therefore, the diagram presented here differs from the lifecycle diagram we showed in Figure 4.20. For instance, the states "offered", "allocated", and "started" of the work item lifecycle correspond to the "service invocation" state of the activity's lifecycle. The lifecycle diagram of an activity we showed in Figure 4.20 is also more centred around the discovery of services, and failures that can occur due to the volatile connections that dominate a nomadic network. These differences explain the presence

[2] A *work item* is a runtime instance of an atomic task definition.

of other failure handling primitives (such as rediscover) in the activity's lifecycle diagram.

Besides these failure handling primitives, Russell [RtEv05a] proposes three alternatives for failure handling at the level of a case (i.e., workflow instance), namely continuing the workflow case, removing the current case, and removing all cases. The first alternative can be accomplished by skipping the activity that failed whereas the second one can be modelled by wrapping the entire workflow with a Failure pattern and specifying "skip all" as the compensating action that must be executed in case of a failure. The third alternative cannot be modelled using our failure handling mechanism, because a Failure pattern only affects the execution of a single workflow instance.

As possible recovery actions, Russell identifies three possible actions: do nothing, rollback to a specific point in the process, and compensate by performing specified compensation actions. Our failure handling mechanism does not allow rollback as a compensating action. In nomadic networks, where disconnections are warp and weft, rollback of a process/task is not always possible, because the service that performed a task might not be in range any more.

WS-BPEL [JE+07] offers the concept of a *fault handler* for handling exceptions. A fault handler can be attached to a process, a scope or as a shorthand notation on an invoke-activity and it is installed when its associated scope is started.

YAWL [vdAtH05] only has built-in support for failure handling for atomic tasks (i.e., a single activity and not a sub workflow). In YAWL it is not trivial to express failure handling strategies over a sub workflow. Introducing a Failure pattern that can specify compensations for failure events, enables a strong mechanism for failure handling. These failure patterns can be nested to acquire more drastic compensations and compensating actions can even be chained together, such that more complex failure handling can be achieved.

Research to handling of expected failures has led to the development of transactional workflows [SMA+97, SR93, AAA+96]. ConTracts [RS95] provides a failure model for workflow applications by introducing so-called *contracts*. A contract is a long-lived transaction composed of steps, where a script describes the order in which these steps need to be executed. Each of these steps is associated with a compensation step that undoes the effects of the execution step. Transactional steps are rolled back, whereas compensation of a non-transactional step requires human intervention. ConTracts proposes a coordinated transaction model that allows both forward, backward and partial recovery when a failure event is signalled. Backward recovery compensates steps of which the execution is completed, typically

in reverse order of their execution. Recovery can be executed up to the point that forward execution can be resumed (typically along a path, different from the one that caused the failure), in which case the recovery is only partial. The failure handling we propose differs from the transaction mechanism that is proposed by the ConTracts project. First of all, we allow the programmer to define application-specific compensations, which do not require any human intervention. Furthermore, we argue that rolling back execution steps is not always desirable and/or possible in a nomadic network. In these networks, services can reside on mobile devices that can go out of range at any moment in time. Hence, it is possible that the when a task (i.e., a service invocation) fails, the roll back cannot be performed because the service that was invoked is no longer in communication range.

OPERA [HA98] incorporates language primitives for exception handling and their strategies into a workflow system. OPERA introduces the notion of *spheres* that are used to categorise operations as units with transactional properties. When a task fails during the execution of an OPERA workflow, the execution is stopped and the execution is transferred to the installed exception handler. Exception handling is specified by defining tasks and the control and data flow between these tasks. In OPERA, each event must be managed separately and for every case affected by the exception, a different exception handler must be triggered. The approach we propose is very similar to the exception handling mechanism employed by OPERA. For each type of failure event a (chain of) compensating action(s) can be specified.

Another way to specify exception handling strategies is by using event-condition-action (ECA) rules, as seen in, for example, the exception language Chimera-Exc [CCPP99]. The types of events that are defined by Chimera-Exc are data manipulation events, temporal events, external events, and workflow events. The actions supported by this exception language are either data modification primitives (for instance, to modify the value of an object's attribute) or workflow management primitives (which can, for example, start a new task). Since maintaining transactions in a mobile network where participants can disconnect indefinitely, rolling back transactions, etc. can be problematic and is not ideally suited for the environment we are targeting.

Adaptive workflow research and workflow evolution research focus on the detection and handling of unexpected failures introduced due to the dynamic changing of the workflow. ADOME-WFMS [CLK01] is an example of a project that provides both support for expected and unexpected failures where exception handlers are specified using ECA rules. The compensating actions (like skipping, or repeating a task) that are supported by ADOME-WFMS can also be achieved using the failure handling mechanism we describe. Moreover, we allow compensations to be

chained, such that a cascade of compensating actions is executed in case a failure occurs.

Failure handling or recovery concepts are proposed by Eder [EL96]. They categorise two recovery techniques within workflow execution, namely automatic and manual tasks. Automatic tasks are further divided into restarting the same task, starting an alternative task and manual intervention. Recovery for a manual task must be performed by the workflow participant responsible for that particular task. The failure handling mechanism we introduce by specifying compensations for specific failure events allows the same recovery concepts that Eder proposes. The several compensations we introduce cover the automatic tasks proposed by Eder. Recovery via manual tasks can be achieved by specifying a component (sub workflow or activity) as the compensating action of a failure, where that component implements the human intervention that is needed, for instance, by implementing a service that asks for user input.

4.6 Conclusion

In this chapter we introduced a programming model for orchestration of services in nomadic networks. This programming model consists of workflow patterns that allow the orchestration of services in nomadic networks. First, we introduced the functionality that must be provided by activities in nomadic networks, and explained a data flow mechanism that can be used in these networks. Afterwards, we explained how existing control flow patterns can be used in nomadic networks. We presented a categorisation for these patterns based on the way the control flow patterns can be composed and how they deal with data. The combination of activities, data flow and control flow patterns allow the orchestration of services in a nomadic network.

In order to support orchestration of groups of services, novel patterns were introduced. We showed how intensional definitions can be used to describe the members of the group. We proposed a pattern that allows a process to be executed for these group members and presented a pattern to restrict the members during the execution of that process. We also put forward several patterns that allow the synchronisation of the execution of the process that transcends the execution of the individual instance of a group member.

The combination of the above patterns for service orchestration and group orchestration almost adhere all criteria we postulated for orchestration in nomadic networks. In order to fulfil all criteria, some failure handling mechanism must be available. We propose the use of a failure pattern that can wrap a sub workflow and

where compensating actions for specific types of failures can be specified. Moreover, we argue that a default compensation strategy must be available to overcome the frequent network failures that dominate a nomadic network. We also showed how this failure pattern can be used in combination with the proposed patterns for group orchestration.

To summarise, we enumerate the criteria for orchestration in nomadic networks (defined in Section 2.3) and show which abstractions we introduced to fulfil these postulated criteria.

Service Orchestration	
1. Time decoupling	During the "service invocation" execution step, it is possible that the service becomes temporarily unavailable, while the activity is waiting for the result of the invocation. This ensures decoupling in time, because the workflow and the service do not need to be online at the same time. (Section 4.1)
2. Space decoupling	The activity of a workflow does not need to know a priori the service it must invoke. It is possible to define an activity using a service type. During the "service discovery" execution step, the activity will discover a service of the correct type. (Section 4.1)
3. Synchronisation decoupling	Invocation of a service is achieved by sending an asynchronous message to the service, ensuring that the execution of the workflow and the service are not blocked upon sending or receiving messages. (Section 4.1)
4. Explicit control flow	Introducing the control flow patterns, proposed by van der Aalst et al. [RtHvdAM06], ensures that the fine-grained application logic is separated from the control flow of an application. (Section 4.3)

Group Orchestration	
5. Intensional definition	Using service types and logical expressions, the services that form a logical group can be described intensionally. (Section 4.4.1)
6. Arity decoupling	A Group pattern allows a group of services to be treated as a single unit. The definition of this pattern specifies that the number of participants can fluctuate over time, i.e., new services satisfying the group's description can join at any moment in time. (Section 4.4.1) Moreover, services can also disjoin the group (for example, by using the Drop compensating action to recover from failures). (Section 4.5.3)
7. Dynamic modification	The participants of the group can be restricted during the execution. A Filter pattern allows to filter out group members that do not satisfy a given condition. (Section 4.4.1)
8. Synchronisation mechanisms	Novel patterns (Barrier, Cancelling Barrier, Group Join, Synchronised Task) allow the execution of a group to be controlled. (Section 4.4.2)
Failure Handling	
9. Automatic failure handling	Automatic handling of failures that occur during the execution of an activity by executing predefined compensations, such as retrying to invoke the service. (Section 4.5.1)
10. Explicit failure handling	The ability to formulate different compensation strategies on different levels of granularity by overriding and/or extending the default failure handling behaviour. These compensations can be chained together such that a cascade of compensating actions is performed when a failure is detected. (Section 4.5.2)
11. Individual failure handling	Reacting upon a failure that occurs during the individual process execution of a single member of the group can be realised by composing the Failure pattern and the Group pattern such that the Failure pattern is defined inside the Group pattern. (Section 4.5.3)
12. Failure handling for groups	Detecting and handling failures at the group level is realised by composing a Failure pattern and a Group pattern such that the Group pattern is (partly) wrapped by a Failure pattern. Depending on the category of compensating action that is specified for the detected failure, the execution of an individual group instance (compensation of category 1) or the entire group (compensation of category 2) is affected. (Section 4.5.3)

5

A WORKFLOW LANGUAGE FOR ORCHESTRATION IN NOMADIC NETWORKS

Now that we have introduced patterns that allow orchestration in nomadic networks, we can present a workflow language that has support for the proposed orchestration patterns. The workflow language for nomadic networks we present, called NOW, is built as a library on top of the ambient-oriented programming language AMBIENTTALK. Throughout this chapter we explain the syntax and semantics of this workflow language. We start this chapter with a motivation for using AMBIENTTALK as the platform to build our nomadic workflow language on (Section 5.1). Subsequently, we introduce the notion of activities in NOW and present the implementation of a service in NOW (Section 5.2). Thereafter, we explain the data flow mechanism employed by NOW (Section 5.3). In this chapter we also present the patterns supported by the workflow language to allow orchestration in nomadic networks. First of all, we describe the control flow patterns that are provided by NOW (Section 5.4), and show how these patterns can be used to build workflows. Secondly, we present the patterns for group orchestration and implement the SURA application using these patterns (Section 5.5). Thirdly, we introduce the patterns for failure handling in NOW, and implement the SWOOP application using these patterns (Section 5.6). Before concluding this chapter, we present a survey of existing workflow languages and coordination languages and evaluate these languages using the criteria we postulated in Chapter 2.

5.1 Motivation

In this section we motivate why we develop our nomadic workflow language NOW on top of AMBIENTTALK. AMBIENTTALK is a distributed programming language targeted towards mobile ad hoc networks which deals with volatile connections at the heart of its programming model. As nomadic networks are a special case of MANETs, a lot of AMBIENTTALK's features are useful for our work as well. AMBIENTTALK allows decoupled communication because it enables decoupling in time, decoupling in space, synchronisation decoupling, and arity decoupling.

However, as we discussed in Section 3.8, AMBIENTTALK has three limitations with respect to orchestrating services in a nomadic network.

First, the application logic in AMBIENTTALK is divided amongst several event handlers which can be triggered independently of one another [CM06]. For small examples it is still manageable to understand the control flow. However, the control flow of large-scale event-driven applications can become very hard to understand. Applications implemented in AMBIENTTALK are not easily manageable as the control flow and the fine-grained application logic are interwoven. This makes AMBIENTTALK applications hard to maintain and difficult to reuse. Recall that the nomadic applications we envision, need to control the different types of services in the network. The emphasis of these application is on the orchestration of the services. Therefore, there needs to be a focus on (the control flow of) the process, i.e., the activities a process constitutes of, the order in which these activities need to be executed, etc.

Secondly, AMBIENTTALK only has limited support for groups. The programming language introduces ambient references to enable communication with a volatile group of nearby objects. This communication is implemented by a single asynchronous message send. Therefore, it is not possible to redefine the members of the group during communication. Moreover, there are no mechanisms provided to express failure handling on these ambient references.

Thirdly, AMBIENTTALK provides only one built-in strategy which hides failures by automatically reconnecting. For MANETs, where devices can join and disjoin at any moment in time, (network) failures must be considered the rule rather than the exception. Therefore, mechanisms that allow the automatic recovery of (or compensating for) these failures should be available. For example, when a service that is being invoked goes offline, the task that service is executing cannot be finished properly, and appropriate actions should be taken to ensure the correct functioning of the application.

In the following sections we discuss how to add a layer of abstraction on top of AMBIENTTALK (which uses messages/events as the level of abstraction) such that the asynchronously executing processes can be orchestrated by means of workflow abstractions. The patterns for service orchestration, group orchestration, and failure handling NOW introduces are available as a library for AMBIENTTALK[1]. By introducing this extra layer of abstraction we present a solution for AMBIENTTALK's limitations with respect to orchestration in nomadic networks.

5.2 Activities

Activities are used to describe a piece of work (a task) that needs to be performed. The activities we described in Section 4.1 are used to invoke services residing in the nomadic network. Such an activity typically consists of a service, an operation, input parameters and output parameters. The abstract grammar of such an activity in NOW is presented in Listing 5.1.

```
<data variable>       := "Env." <symbol>
<service description> := <type_tag> | <data variable>
<argument>            := <expression> | <data variable>
<activity>            := <service description> "." <operation> "("
                         <argument>* ")" [ "@Output(" <data variable>* ")" ]
```

Listing 5.1: Abstract grammar of activities in NOW.

Please note that we use the following convention for the abstract grammars presented in this chapter: nonterminals are put in italic when they are part of AMBIENTTALK's syntax. For instance, the nonterminal *<operation>* is an AMBIENTTALK message that can be sent to an actor.

As we can derive from this abstract grammar, activities in NOW consist of a service description, an operation, arguments, and possibly output parameters.

- The **service description** can be a service type (`type_tag`) or a reference to a particular service (`"Env."<symbol>`). We use the `Env` keyword to specify that the reference to the service needs to be looked up in the data environment that is passed to the activity when started (see Section 5.3). Depending on the type of service description that is used, an activity needs to perform the "Service Discovery" execution step when started (see Section 4.1).
- The **operation** is the name of the method that must be invoked on the preferred service.

[1] http://soft.vub.ac.be/~ephilips/NOW

- The **arguments** are either AMBIENTTALK expressions, or references to variables that are stored in the data environment. When an activity is started, these latter types of arguments are looked up in the data environment that is passed to the activity.
- The **output parameters** are either references to variables in the data environment which need to be updated or names of variables for which a new binding must be added in the data environment.

As we can derive from this grammar, output parameters only need to be specified when required.

```
def type := createServiceType('LuggageHandler);
def act := type.getInfo(Env.flightNr)@Output(Env.trailer, Env.belt);
```

Listing 5.2: Definition of an activity in NOW.

Consider the code in Listing 5.2 where we create an activity (`act`). The definition of an activity returns an object which can be started with a data environment (`act.start(env)`). In this particular example, the activity is instantiated with a service type `LuggageHandler`.

NOW Services

Nomadic applications interact with different categories of services (stationary, registered, and user services). NOW is oblivious to these types of services, which are implemented in the same way, and interacting with a service is done identically. The only part of the application where the difference between the kinds of services is made explicit, is in the definition of an activity. As we can derive from the abstract grammar (see Listing 5.1), the `<service description>` can be either a service type or a reference to a service that is stored in the data environment. Activities that want to invoke services that are part of the infrastructure and that are known a priori (i.e., stationary services and registered services) typically use a reference to that service, whereas activities that interact with user services utilise service types.

Services in NOW can be implemented as distributed objects in AMBIENTTALK that are invoked by sending asynchronous messages (as explained in Section 3.4.1). Listing 5.3 shows how a NOW service can be created, based upon AMBIENTTALK's facilities for defining and exporting objects. The code excerpt below is AMBIENTTALK code which is executed on a (mobile) device and not on the fixed infrastructure of the nomadic network.

```
1  deftype LuggageHandler <: Service;
2
3  def service := isolate: {
4      def companyName := "Aviapartner";
5      /* ... other fields */
6
7      def init(cn) {
8          self.companyName := cn;
9          /* assignment of other fields */
10     };
11
12     def removeLuggage(flight, name, id) {
13         /* Remove luggage from flight,
14            and return the location where it is stored */
15     };
16
17     /* ... other methods */
18 };
19
20 registerService(LuggageHandler, service);
```

Listing 5.3: Implementation of a service.

On the last line of code (line 20), the `applicationService` function, which is provided by NOW, is called. This function is, amongst others, responsible for publishing a service on the network, given its type tag, which is defined on the first line of code. All of these tags need to be subtypes of the `Service` type tag that is provided by NOW. This is required to allow group orchestration, as we explain in Section 6.4.1, where we discuss the implementation of group orchestration.

The service itself is defined as an AMBIENTTALK object, as we can see on lines 3-18. As we have seen in Chapter 3, such an object can be instantiated using the `object:` keyword. Recall that the arguments of an activity can be references to variables in the data environment, and, hence, can be references to services. Therefore, the fixed infrastructure needs to send such a service object to a (possibly mobile) device, that runs the service that is being invoked. During this service invocation, the reference to that service must be synchronously accessible. Synchronous access ensures that during the service invocation an immediate response can be retrieved, and that no event handlers need to be installed to await the result of an asynchronous message send. Therefore, the service object must be passed by copy instead of by far reference, and hence, we must use the `isolate:` keyword (on line 3).

Recall that a symbiosis between AMBIENTTALK and JAVA exists. Because of this symbiosis, it is also possible to implement services that are orchestrated by NOW in JAVA, instead of AMBIENTTALK. Listing 5.4 shows the implementation of the same service we defined in Listing 5.3, but is now implemented in JAVA.

```
1  package airport;
2
3  public class LuggageHandler implements Serializable {
4
5      private String companyName;
6
7      public LuggageHandler(String cn) {
8          this.companyName = cn;
9      }
10
11     public String[] removeLuggage(String flight, String name, int id) {
12         /* Remove luggage from flight,  and return the location
13            where it is stored */
14     }
15 }
```

Listing 5.4: Implementation of a service in JAVA.

In order to discover a service that is implemented in JAVA, an AMBIENTTALK actor that publishes the service needs to be defined. Such an actor is defined in Listing 5.5.

```
1  def ATService := actor: {
2
3     def service := jlobby.airport.LuggageHandler.new("Aviapartner");
4     deftype LuggageHandler <: Service;
5     export: service as: LuggageHandler;
6
7  };
```

Listing 5.5: Implementation of a service in JAVA: publishing the service.

5.3 Data Flow

In Section 4.2 we introduced an *environment passing style* which ties the data flow to the control flow of the application. The data environment that is passed between the activities is implemented as an object which contains a unique identifier and a dictionary mapping values to variables.

In Section 4.2 we have shown how data can be used to invoke a service, and how response management (updating the data environment after the service invocation) is taken care of by the activity. Before invoking the service, the input parameters are looked up in the data environment, and when the result of the invocation is received, these values are bound to the variables that are specified using `@Output` (see Section 5.2).

In this section, we first show the implementation in NOW of the activities depicted in Figure 4.2. The first activity addresses a specific luggage handler and asks which trailer will transport the luggage for a particular flight, and also wants to know to which belt this luggage is transported. This can be implemented in NOW as follows

```
1 def env := DataEnvironment.new();
2 env.insert('luggageH, <far reference>);
3 env.insert('flightNr, "9W226");
4 def act := Env.luggageH.getInfo(Env.flightNr)@Output(Env.trailer,Env.belt);
5 act.start(env);
```

On the first line in the above code snippet, a new data environment is created, and two new variable bindings are inserted (lines 2-3). On line 4 a new activity is created, with the operation `getInfo`, a single argument (`Env.flightNr`), and with two output parameters (`Env.trailer` and `Env.belt`). The activity's execution is started on line 5, by passing it the data environment that is created on line 1.

The second example activity in Figure 4.2 is used to contact an assistance personnel member to notify him/her that a certain person is missing. In this example, any assistance personnel member that is in the neighbourhood suffices for this task. Therefore, this activity is instantiated with a service type, instead of a reference to a particular service. The implementation of this activity in NOW is given in the following code snippet.

```
1 createServiceType('Assistance);
2 def env := DataEnvironment.new();
3 env.insert('person, <far reference>);
4 def act := Assistance.missing(Env.person)@Output(Env.found);
5 act.start(env);
```

On line 1, a new service type (`Assistance`) is created. This service type serves as the service description of the activity that is created on line 4. The activity is created with the operation `missing`, the argument `Env.person`, and has a single output parameter (`Env.found`). Similar to the previous activity, this newly created activity's execution is started on line 5 by passing it the data environment, defined on line 2.

The data flow mechanism becomes more complex when synchronisation patterns are involved. As we already explained in Section 4.2, synchronisation patterns must specify a merging strategy that resolves conflicts in the incoming data environments. Such a merging strategy can be implemented by using an AMBIENTTALK block closure that has a table of data environments as its single argument.

```
{ | envs | ... };
```

Consider the example we showed in Figure 4.4. In that small workflow example a booking agent looks for a free seat on a flight to Italy. To this end, the services of two airline companies are invoked. Afterwards, their data environments need to be merged. A possible merging strategy could be to use the data environment that has the lowest price for that flight. This strategy can be implemented by the following function:

```
def myStrategy := { | envs| def env := envs[1];
                     def tuple := [`flight, `price];
                     def [flight, price] := envs[1].find(tuple);
                     envs.each: { |e| def [_, p] := e.find(tuple);
                                      if: ( p < price ) then: {
                                           env := e;
                                           price := p }; };
                     env; };
```

This block closure iterates over all data environments (`envs.each:`) and returns the data environment that has the smallest value for the variable `price`.

NOW has support for the four merging strategies that we have identified in Section 4.2, namely "prioritise", "random", "merge", and "restore". Merging strategies are used to instantiate a synchronisation pattern, as we show in the next section, where we introduce NOW's control flow patterns.

5.4 Service Orchestration

Now that we have explained how activities and data flow are introduced by NOW, we describe how these activities can be linked together in order to form a workflow description. The *control flow perspective* defined by van der Aalst et al. [RtHvdAM06] categorises 43 patterns which describe the control flow dependencies between several activities. NOW has built-in support for the most common control flow patterns. The current implementation consists of 19 control flow patterns of van der Aalst that are used by standard workflow languages, such as BPEL [RtHvdAM06]. Of the 43 patterns defined by van der Aalst the workflow language BPEL has support for a similar set of 21 control flow patterns. NOW's patterns range from very basic ones, like Sequence, to more advanced patterns such as the Structured Loop pattern and multiple instances patterns. NOW does not implement control flow patterns of the category "cancellation and force completion patterns", but cancelling tasks can be achieved using the language's support for failure handling, which is discussed in Section 5.6. In Table 5.1 we show an inventory of van der Aalst's control flow patterns and indicate whether the pattern is supported by NOW (in the table's third column).

Basic Control Flow Patterns		
Sequence	standard	Yes
Parallel Split	standard	Yes
Synchronization	synchronisation	Yes
Exclusive Choice	standard	Yes
Simple Merge	synchronisation	Yes
Advanced Branching and Synchronization Patterns		
Multi-Choice	standard	Yes
Structured Synchronizing Merge	synchronisation	Yes
Multi-Merge	synchronisation	Yes
Structured Discriminator	synchronisation	*No*
Blocking Discriminator	synchronisation	*No*
Cancelling Discriminator	synchronisation	Yes
Structured Partial Join	synchronisation	Yes
Blocking Partial Join	synchronisation	*No*
Cancelling Partial Join	synchronisation	*No*
Generalised AND-Join	synchronisation	*No*
Local Synchronizing Merge	synchronisation	*No*
General Synchronizing Merge	synchronisation	*No*
Thread Merge	synchronisation	*No*
Thread Split	standard	*No*
Multiple Instance Patterns		
Multiple Instances without Synchronization	standard	Yes
Multiple Instances with a Priori Design-Time Knowledge	standard	Yes
Multiple Instances with a Priori Run-Time Knowledge	standard	Yes
Multiple Instances without a Priori Run-Time Knowledge	standard	*No*
Static Partial Join for Multiple Instances	synchronisation	Yes
Cancelling Partial Join for Multiple Instances	synchronisation	*No*
Dynamic Partial Join for Multiple Instances	synchronisation	*No*
State-based Patterns		
Deferred Choice	standard	*No*
Interleaved Parallel Routing	standard	Yes
Milestone	standard	*No*
Critical Section	standard	*No*
Interleaved Routing	standard	*No*
Cancellation and Force Completion Patterns		
Cancel Task	standard	*No*
Cancel Case	standard	*No*
Cancel Region	standard	*No*
Cancel Multiple Instance Activity	standard	*No*
Complete Multiple Instance Activity	standard	*No*
Iteration Patterns		
Arbitrary Cycles	standard	*No*
Structured Loop	standard	Yes
Recursion	standard	*No*
Termination Patterns		
Implicit Termination	NA	Yes
Explicit Termination	standard	*No*
Trigger Patterns		
Transient Trigger	trigger	Yes
Persistent Trigger	trigger	Yes

Table 5.1: Control Flow Patterns supported by NOW.

The grammar of the control flow patterns in NOW is shown in Backus-Naur form in Listing 5.6. This grammar is an extension of the abstract grammar we presented in Listing 5.1.

```
<component>         := <activity> | <pattern>
<closure>           := "{ |" <parlist> "|" <body> "}"
<condition_action>  := "[" <closure> "," <component> "]"
<strategy>          := "merge" | "restore" | "prioritise" | "random" |
                       "{ | envs |" <body> "}"
<sync_cmp>          := <component> [ "," <strategy> [ "," <integer> ] ]
<pattern>           := <std_pattern> | <sync_pattern> | <trigger_pattern>
<trigger_pattern>   := "TransientTrigger(" <component> ")" |
                       "PersistentTrigger(" <component> ")"
<sync_pattern>      := "Synchronization(" <sync_cmp> ")" |
                       "SimpleMerge(" <sync_cmp> ")" |
                       "StructuredSynchronizingMerge(" <sync_cmp> ")" |
                       "MultiMerge(" <sync_cmp> ")" |
                       "CancellingDiscriminator(" <sync_cmp> ")" |
                       "StructuredPartialJoin(" <sync_cmp> ")"
<std_pattern>       := "Sequence(" <component>+ ")" |
                       "ParallelSplit(" <component>+ ")"  |
                       "Connection(" <sync_pattern> ")" |
                       "ExclusiveChoice(" <closure> "," <component> ","
                                          <component> ")"  |
                       "MultiChoice(" <condition_action>+ ")" |
                       "MIWithoutSynchronization(" <component> ")" |
                       "StaticPartialJoinMI(" <component> ")" |
                       "MIWithPrioriDTKnow(" <component> "," <integer> ")" |
                       "MIWithPrioriRTKnow(" <component> "," <closure> ")" |
                       "InterleavedRouting(" <component>+ ")"
                       "StructuredLoop(" <component>+ ")"
```

Listing 5.6: Abstract grammar of control flow patterns in NOW.

The abstract grammar makes the distinction between standard patterns, synchronisation patterns, and trigger patterns, the three categories of control flow patterns we proposed in Section 4.3. From the abstract grammar in Listing 5.6 we can derive that each pattern is implemented as a function which takes a (collection of) component(s) as its argument. Such a component can be either an activity or a pattern.

Before showing how these patterns can be used to build a workflow, we need to discuss the composition of synchronisation patterns. In Section 4.3 we introduced three categories of control flow patterns, namely standard patterns, synchronisation patterns, and trigger patterns. Synchronisation patterns differ from the other two categories in the way data flow needs to be handled, i.e., a data merging strategy needs to be specified such that the data environments of the incoming branches can be merged.

5.4.1 Composition of Synchronisation Patterns

Composition of patterns becomes more complex when synchronisation patterns are involved. The distinction between standard and synchronisation patterns is made because the latter category needs to support multiple incoming branches. These incoming branches need to be linked to the synchronisation patterns using so-called *connections*.

Connections are necessary for complex compositions where not all outgoing branches, for instance, of a Parallel Split pattern, are connected to the same Synchronization pattern. When all branches of a split pattern are joined by one synchronisation pattern, it would be possible to write the following code

```
Sequence( ParallelSplit( branch1, branch2, ...), Synchronization( ... ) )
```

This would imply that first the split pattern (for example, Parallel Split) is executed and afterwards all its branches are merged by the *same* Synchronization pattern. An example of this composition scenario is depicted by the workflow in Figure 5.1.

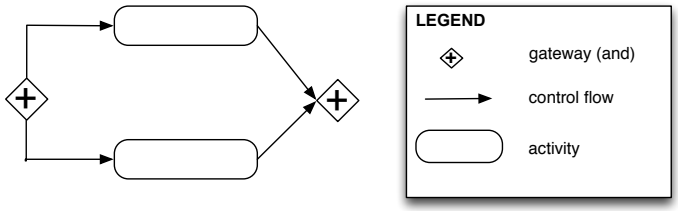


Figure 5.1: Composition of synchronisation patterns: one Synchronization pattern.

However, it is also possible that the branches of the Parallel Split are merged by *different* synchronisation patterns. Hence, for each branch it must be specified which synchronisation pattern is responsible for its merging process. Such an example is depicted in Figure 5.2.

An outline implementation of the workflow depicted in Figure 5.2 is:

```
1  def cancDiscr := CancellingDiscriminator();
2  def sync := Synchronization();
3
4  ParallelSplit(
5      Sequence( ..., Connection(cancDiscr) ),
6      Sequence( ..., ParallelSplit( Sequence( ..., Connection(cancDiscr) ),
7                                    Sequence( ..., Connection(sync) ) ) ),
8       Sequence( ..., Connection(sync) ),
9       Sequence( ..., Connection(sync) ) );
```

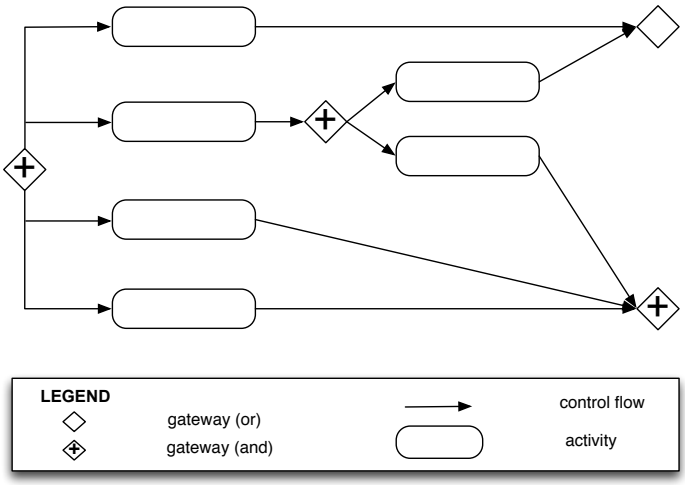


Figure 5.2: Composition of synchronisation patterns: several Synchronization patterns.

As we can see, in this example there are two synchronisation patterns used, namely a Cancelling Discriminator and a Synchronization. These patterns are defined upfront. Synchronisation patterns cannot be created inline, because every time the function is called (for instance `Synchronization(...)`) a new pattern is instantiated. Because a single synchronisation pattern has multiple incoming branches, the patterns need to be defined upfront, and a reference to that pattern needs to be used when building the workflow. As we can see, the outgoing branches of a Parallel Split pattern are sequences of the components in that split pattern, followed by a `Connection` pattern that wraps the synchronisation pattern.

5.4.2 Implementing the iMPASSE Application in NOW

In Section 2.3.1 we introduced three scenarios of nomadic applications. In this section we present the implementation of the scenario of the iMPASSE application, which focusses on service orchestration, in NOW. This application is used to assist the personnel at the airport, by implementing, for instance, the necessary actions that must be performed in case a passenger is too late for boarding.

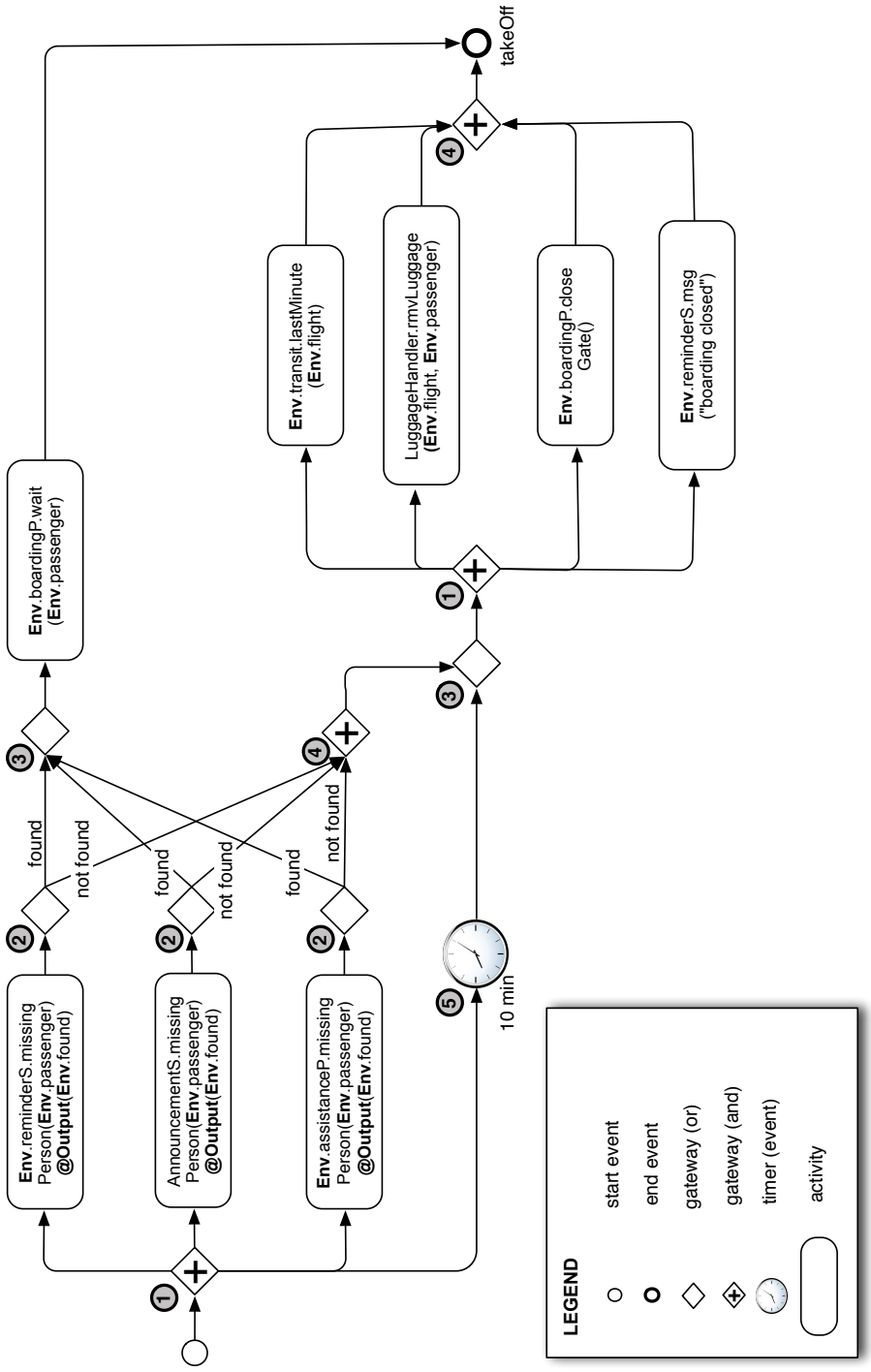


Figure 5.3: Workflow implementing the iMPASSE application.

Figure 5.3 depicts the workflow implementing this application. The workflow uses the following control flow patterns (annotated using grey circles) [RtHvdAM06]:

1. **Parallel Split**: The example uses this pattern twice. The first Parallel Split pattern is introduced because several tasks need to be executed in parallel at the moment a passenger is missing. In that case, three tasks need to be executed: a reminder must be sent to the passenger, the passenger must be announced, and the assistant personnel needs to start looking for the missing passenger.
   The second Parallel Split pattern is needed to implement the part of the scenario where four tasks need to be executed when the passenger cannot be found, such that the plane can take off after those tasks have been performed. The tasks that must be executed before the plane can take off are: informing the transit airport of the free seat on the plane, removing the passenger's luggage, closing the gate, and contacting the passenger to inform him that he missed his flight.
2. **Exclusive Choice**: The workflow has three occurrences of this pattern. The pattern is used to select between the appropriate actions that must be executed, depending on whether the passenger is found or not.
3. **Cancelling Discriminator**: This pattern is used twice in the workflow implementing the application. The first Cancelling Discriminator is used to ensure that as soon as a signal (more specifically, the first signal) is received that the passenger is found, the boarding personnel is informed to wait for him. The second Cancelling Discriminator pattern is used to implement the part where actions need to be performed when a passenger is not found. We can conclude that a passenger cannot be found, either after 10 minutes, or when all three services (the reminder service, the announcement service, and the assistance personnel) reply that the passenger is not at the airport. When one of these two conclusions can be drawn, the appropriate actions are taken.
4. **Synchronization**: Two Synchronization patterns are used to implement this scenario. The first Synchronization pattern is used to ensure that only when all three services (the reminder service, the announcement service, and the assistance personnel) have replied that the person cannot be found, appropriate actions are taken.
   The second Synchronization is used to ensure that the flight can only take off when all four actions (informing transit airport, removing luggage, closing gate, and sending message to passenger) are taken.

5. **Persistent Trigger**: The Persistent Trigger pattern is used to ensure that after ten minutes, a signal is received. The execution of the workflow acts upon this signal by ensuring that the appropriate actions are taken (for instance, the luggage of the passenger needs to be removed).
6. **Sequence**: This pattern is not annotated in Figure 5.3, but is, amongst others, used to connect the activities and the Exclusive Choice patterns in the outgoing branches of the first Parallel Split pattern.

This workflow can be implemented in NOW as is shown in Listing 5.7.

```
1  def sync1 := Synchronization(Env.tower.takeOff(Env.flight), restore);
2
3  def cancDiscr1 :=
4       CancellingDiscriminator(
5            ParallelSplit( Sequence( Env.transit.lastMinute(Env.flight),
6                                     Connection(sync1) ),
7                           Sequence( LuggageHandler.rmvLuggage(Env.flight,
8                                     Env.pass), Connection(sync1) ),
9                           Sequence( Env.boardingP.closeGate(),
10                                    Connection(sync1) ),
11                          Sequence( Env.reminderS.msg("boarding closed"),
12                                    Connection(sync1) ) ) );
13
14 def sync2 := Synchronization(Connection(cancDiscr1), restore);
15
16 def cancDiscr2 := CancellingDiscriminator(
17                      Sequence( Env.boardingP.wait(Env.pass),
18                                Env.tower.takeOff(Env.flight) ) );
19
20 def wf := ParallelSplit(
21       Sequence( Env.reminderS.missingPerson(Env.pass)@Output(Env.found),
22                 ExclusiveChoice( {|found| found}, Connection(cancDiscr2),
23                                  Connection(sync2) ) ),
24       Sequence( Env.announcementS.missingPerson(Env.pass)@Output(Env.found),
25                 ExclusiveChoice( {|found| found}, Connection(cancDiscr2),
26                                  Connection(sync2) ) ),
27       Sequence( Env.assistanceP.missingPerson(Env.pass)@Output(Env.found),
28                 ExclusiveChoice( {|found| found}, Connection(cancDiscr2),
29                                  Connection(sync2) ) ),
30       PersistentTrigger( Connection(cancDiscr1), after(minutes(10)) ) );
```

Listing 5.7: Implementation of the iMPASSE application in NOW.

As we can see in Listing 5.7, we define four synchronisation patterns upfront. Recall that defining synchronisation patterns inline is not possible, because several outgoing branches of a split pattern possibly refer to the same synchronisation pattern. On line 1 and line 14 we create the two Synchronization patterns that are used in this workflow. The first Synchronization (`sync1`) is used to synchronise the four activities, such that the plane only takes off when those four activities

have finished their execution. The second Synchronization (`sync2`) is used to ensure that only when all three services have replied that the passenger cannot be found, the necessary actions are performed. Note that both Synchronization patterns are instantiated with the "restore" data merging strategy. During the execution of the Synchronization's incoming branches, no additional variables are added to the data environment, and therefore, this merging strategy can be used. The data environment that is used to start the execution of the remainder of the workflow after the synchronisation pattern, is the same data environment that is used to start each incoming branch individually.

We also define two Cancelling Discriminator patterns upfront. The first Cancelling Discriminator is created on lines 3-12. At the moment this pattern is started for the first time (i.e., when the first incoming branch is enabled), the remainder of the workflow is executed. In this scenario, the execution of a Parallel Split pattern with four outgoing branches is started. Each of these outgoing branches is connected (using a `Connection` pattern) to the same Synchronization (`sync1`). The second Cancelling Discriminator pattern (`cancDiscr2`) is created on lines 16-18 of Listing 5.7. The first enablement of an incoming branch of this pattern results in the execution of the remainder of the workflow after this pattern. In this example, the execution of two activities is executed in sequence, namely informing the boarding personnel to wait for the passenger and informing the control tower that the plane can take off (lines 17-18).

The other patterns can be created inline, because they are classified either as standard or trigger patterns. The workflow starts with a Parallel Split (line 20) that has four outgoing branches. The first three outgoing branches (created on lines 21-23, 24-26, 27-29 respectively) have a similar implementation. Such an outgoing branch consists of a Sequence pattern, which has an activity as its first component, followed by an Exclusive Choice pattern. The Exclusive Choice pattern is either connected to the Cancelling Discriminator (`cancDiscr1`) or the Synchronization (`sync1`). The first argument of the `ExclusiveChoice` function is an AMBIENTTALK block closure which returns a boolean. In this example, when the passenger has been found (i.e., when the value of the variable `found`, which is stored in the data environment, is `true`), the second argument of the function call is executed (`Connection(cancDiscr1)`), otherwise the last argument (`Connection(sync1)`) is executed. Recall that Connection patterns are used to wrap the synchronisation patterns, such that these synchronisation patterns can be notified of their number of incoming branches.

The fourth branch of the Parallel Split pattern (created on line 30) consists of a Persistent Trigger pattern. This pattern is used to wrap part of the workflow that

can only be executed once an external event is received. In our example, only after 10 minutes we can conclude that the passenger is not found and only then the appropriate actions can be performed.

5.5 Group Orchestration

In the previous section, we presented the control flow patterns that are provided by NOW and showed how these patterns can be composed to implement workflows. In this section we extend the workflow language with the necessary abstractions and patterns to allow group orchestration in a nomadic network. We first introduce the abstract grammar of these novel patterns. Thereafter, we describe how intensional descriptions of services can be defined in NOW. We also present the implementation of the SURA application to illustrate how the patterns for group orchestration can be composed.

The grammar of the patterns for group orchestration in NOW is shown in Backus-Naur form in Listing 5.8. This grammar is an extension of the abstract grammars we presented in Listing 5.1 and Listing 5.6.

```
<component>           := <activity> | <pattern>
<strategy>            := "merge" | "restore" | "prioritise" | "random" |
                         "{ | envs |" <body> "}"
<time>                := "time(" <integer> "," <integer> "," <integer> ")"
<duration>            := "hours(" <integer> ")" | "minutes(" <integer> ")" |
                         "seconds(" <integer> ")"
<condition>           := <time_constraint> | <quota_constraint>
<time_constraint>     := "at(" <time> ")" | "after(" <duration> ")"
<quota_constraint>    := "percentage(" <integer> ")" |
                         "amount(" <integer> ")"
<description>         := <intensional_d> | <extensional_d>
<intensional_d>       := <type_tag> | <CRIME_expression> |
                         "union(" <intensional_d> "," <intensional_d> ")" |
                         "diff(" <intensional_d> "," <intensional_d> ")"
<extensional_d>       := "[" <far_reference>* "]"
<pattern>             := <std_pattern> | <sync_pattern> | <group_pattern> |
                         <trigger_pattern> | <group_sync_pattern>
<std_pattern>         := "Filter( { | env |" <body> "} )" |
<group_sync_pattern>  := "Barrier(" <condition> ")" |
                         "CancellingBarrier(" <condition> ")" |
                         "GroupJoin(" <condition> ")"
                         "SynchronisedTask(" <component>, <condition>
                                             ["," <strategy>] ")"
<group_pattern>       := "Group(" <description>, <symbol>,
                                  <component>
                                  ["," <condition> ] ["," <strategy>] ")"
                         "SnapshotGroup(" <description>, <symbol>,
                                          <component> ["," <condition> ]
                                          ["," <strategy>]")"
```

Listing 5.8: Abstract grammar of group orchestration patterns in NOW.

The abstract grammar we present here, extends the grammar shown in Listing 5.6. As we can see in Listing 5.8, two new categories of patterns are added, namely *group patterns* and *group synchronisation patterns*. The group patterns category consists of two patterns, namely the Group and Snapshot Group pattern we introduced in Section 4.4. The category of group synchronisation patterns contains the other group orchestration patterns, such as the Barrier pattern. Group synchronisation patterns differ from synchronisation patterns because a group synchronisation pattern must be the same for *all* running instances of the group. The difference between these categories is explained in more detail in Section 6.4, where we discuss the implementation of the patterns for group orchestration.

We first explain how intensional definitions of services are implemented in NOW. Afterwards, we give the implementation of the SURA application in our nomadic workflow language.

5.5.1 Definition of Group Membership

Members of a group can be defined either extensionally or intensionally. NOW supports both kinds of group descriptions: a group can be either instantiated with a group of far references to the services, or with a type tag or logical expression. In this section we elaborate more on the last types of descriptions, namely the intensional definition of services through a logical expression.

Recall that CRIME's fact space model provides a logic coordination language for reasoning about context information that is represented as facts in a federated fact space. Those facts are locally published by applications and transparently shared between fact spaces residing on nearby devices as long as they are within communication range.

In Figure 4.12 we showed such federated facts spaces, namely the federated fact spaces of two fans and the federated fact space residing on the festival's infrastructure which are connected in a mobile ad hoc network. In this section we use the same intensional description, namely "all the fans of the headliner who are at the festival", which can be expressed as a logical rule with the following prerequisites:

```
shared<-festival_visitor( ?festivalNbr ),
shared<-fan_info( ?festivalNbr, ?band ),
private<-band_info( ?band, "headliner" ).
```

In this example, the intensional description of all participants is a description that consists of three prerequisites:

- the participant must be a festival visitor;
- and that participant must be a fan of a band;
- and that band must be listed as the headliner of the festival.

The intensional description, using these three prerequisites, in NOW is given in the following code snippet:

```
1 def shared := createFactSpace("shared");
2 def private := createFactSpace("private");
3 def temp1 := shared.festival_visitor(Var.participantId, Var.festivalNbr);
4 def temp2 := shared.fan_info(Var.participantId, Var.festivalNbr, Var.band);
5 def temp3 := private.band_info(Var.id, Var.band, "headliner");
6 def description := makeIntensionalDescription(temp1, temp2, temp3);
```

The three prerequisites are defined on line 3, line 4, and line 5 respectively. Note that in NOW, these templates have an extra (first) argument. Before explaining the necessity of this extra argument, we explain how facts are shared.

On line 1 and line 2 in the code snippet above, two types of fact spaces are defined. Facts are published in these fact space, and depending on the kind of fact space, the facts are exchanged with fact spaces of nearby devices.

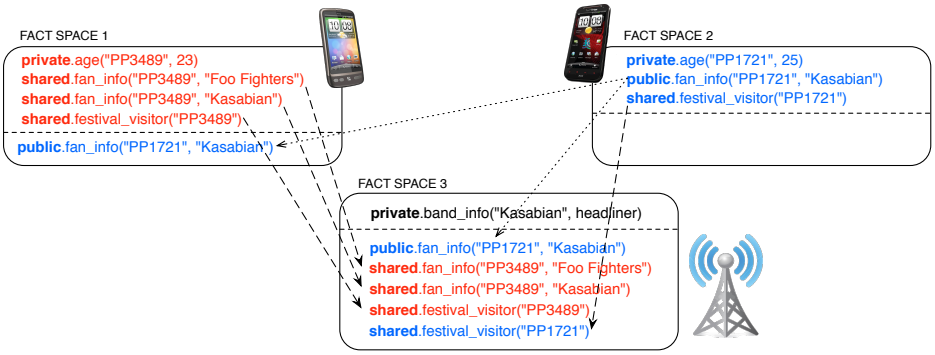


Figure 5.4: Federated fact spaces of colocated devices.

In Figure 5.4 we show the fact spaces of several devices: two fact spaces on the mobile device of a festival visitor, and one fact space on the fixed infrastructure of the festival. In `fact space 1` four facts are published: one private fact (the age of the visitor), and three facts that are shared with fact spaces that have subscribed their interest in these kinds of facts. In the second fact space, `fact space 2`, a private fact is published, a public fact that is exchanged with all connected fact

spaces, and a shared fact. The fact space residing on the fixed infrastructure, `fact space 3`, consists of a single private fact that is locally published. All other facts are originally published in a colocated fact space.

Remark that the prerequisites defined on line 3-5 in the NOW description have one extra argument than the facts that are originally published. For example, the facts of type `fan_info` are published with two arguments (`"PP3489"`, and `"Foo Fighters"`). However, the corresponding prerequisite is defined as a template with three arguments `Var.id`, `Var.festivalNbr`, and `Var.band`. When a fact is inserted in a fact space, NOW ensures that, before insertion, a reference to the service that published the fact is added as the fact's first argument.

This mechanism of adding a reference to the publisher is required so NOW can obtain references to the services that satisfy the intensional description. The intensional description we showed above has three prerequisites using four variables, namely `Var.participantId`, `Var.festivalNbr`, `Var.id`, and `Var.band`. By convention, the first argument of a fact matching the prerequisite is a reference to the service that published the fact. Hence, the variable `Var.participantId` is bound to a reference to the service running on the mobile device of a festival visitor. Similarly, the variable `Var.id` is bound to a reference to a service on the fixed infrastructure that published the fact of type `band_info`. The other variables, `Var.festivalNbr` and `Var.band` are bound to a unique identifier relating to a festival visitor or a band respectively.

Recall that an intensional description in NOW can be used to accumulate references to services that satisfy the given description. The rule is that those references are obtained by searching for all facts that satisfy the description, returning all the references to which the variable `Var.participantId` is bound. In Section 6.4.1 we explain how these references can be obtained.

In the previous example we used a single intensional definition. However, sometimes a combination of intensional definitions is preferred: "all children and parents who are present at the festival area", or "all adults who do not have children", etc. In order to implement these types of intensional definitions, NOW introduces the `union` and `diff` functions.

Consider the example scenario where the festival visitors who are a fan of "Kassabian" and not of the "Foo Fighters" need to be addressed. The intensional definition that describes these participants is:

```
1 def shared := createFactSpace("shared");
2 def temp1 := shared.festival_visitor(Var.participantId, Var.festivalNbr);
3 def temp2 := shared.fan_info(Var.participantId, Var.nbr, "Kassabian");
```

```
4  def temp3 := shared.fan_info(Var.participantId, Var.nbr, "Foo Fighters");
5  def description1 := makeIntensionalDescription(temp1, temp2);
6  def description2 := makeIntensionalDescription(temp1, temp3);
7  def description := diff(description1, description2);
```

In the code snippet above two intensional descriptions are defined. The first intensional description (defined on line 5) is used to capture all the participants who are at the festival and are a fan of Kassabian. Recall that references to these services are obtained using the `Var.participantId` variable that is bound to a reference to the service that published the facts satisfying the prerequisites. The second description, `description2` (defined on line 6), is used to obtain references to the festival visitors who are currently present at the festival and are a fan of the Foo Fighters. The final description is defined by taking the difference (`diff`) of these obtained references (see line 7).

5.5.2 Implementing the SURA Application in NOW

We present the implementation of the SURA application we introduced in Section 2.3.1. This scenario wants to address all fans of a festival's headliner and ask them to vote for the band's playlist. The scenario describing the SURA application exemplifies the need for addressing a group of services and synchronising the application for all these services.

Figure 5.5 shows the workflow implementing this scenario. This workflow consists of the following patterns (both control flow patterns and patterns for group orchestration), which are annotated in the figure using grey circles:

1. **Group**: Several tasks must be executed for multiple services (all fans). A Group pattern is used to capture these tasks and ensure that they are executed for all services satisfying the group's description.
2. **Filter**: This pattern is used to restrict the members of the group during the execution. In our example scenario, at the beginning all fans of the headliner are contacted. Afterwards, only those fans who are interested in participating in the voting process are contacted.
3. **Synchronised Task**: A Synchronised Task pattern is used to capture a part of the application that must be executed once for all group members. In our example, this pattern is used to wrap the task where all votes of the fans must be gathered and sent to the headliner. The condition upon which the execution of this task should be started is given, namely two hours before the start of the concert (i.e., at half past eight).

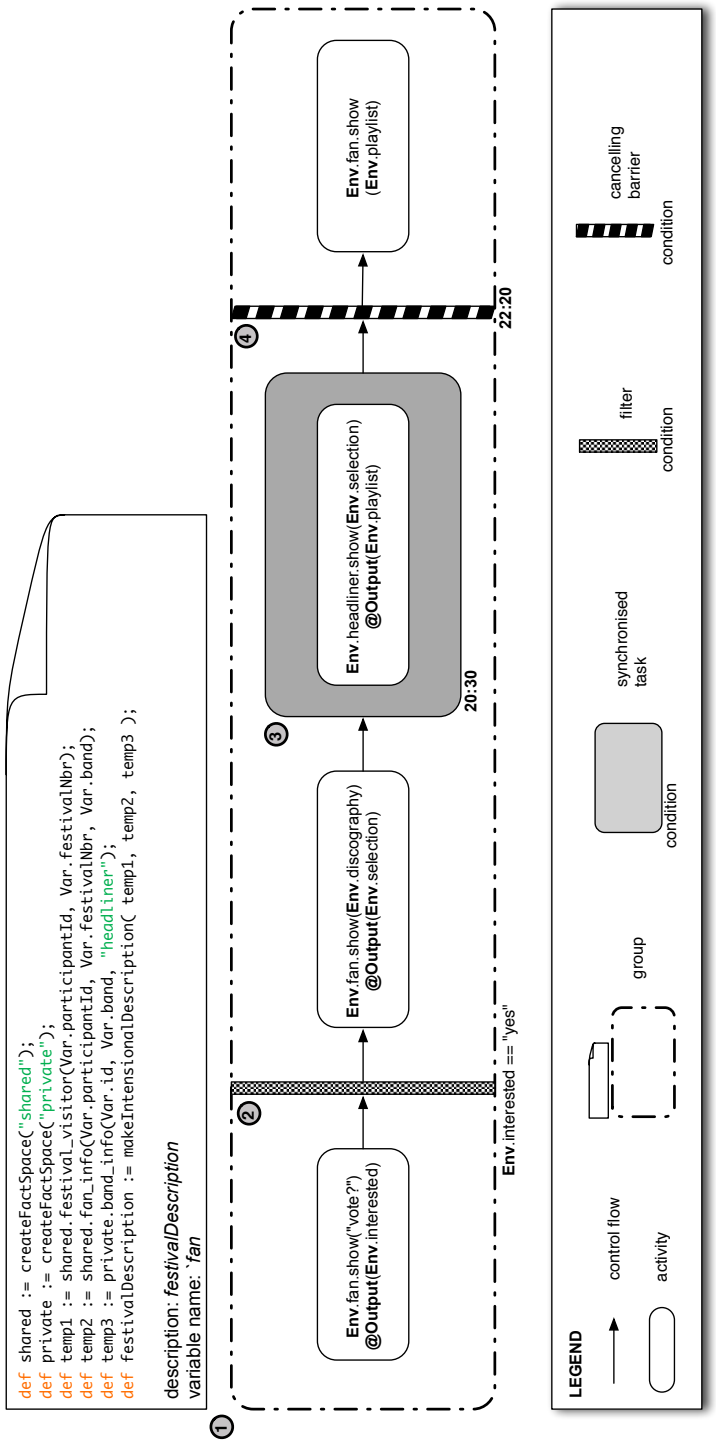


Figure 5.5: Workflow implementing the SURA application.

4. **Cancelling Barrier**: A cancelling barrier is used to block the execution of all group members until a certain condition is satisfied. In our scenario, this pattern is used to ensure that the final playlist is sent to the fans who voted, only ten minutes before the start of the show.
5. **Sequence**: A Sequence pattern is used to ensure that tasks and patterns are executed in the correct ordering. In the scenario, the Sequence pattern ensures, for example, that the Filter pattern, which is used to restrict the group members to be contacted, is executed after the fans are asked whether they want to vote for the playlist or not. Please remark that this control flow pattern is not explicitly annotated in Figure 5.5.

The description that is used to describe the members of the group is a logical CRIME expression which intensionally defines the group members. The expression has three prerequisites, namely be a festival visitor, be a fan of a band, and that band must be the headliner of the festival. This intensional description is the same as we showed earlier in Section 5.5.1.

```
1  def shared := createFactSpace("shared");
2  def private := createFactSpace("private");
3  def temp1 := shared.festival_visitor(Var.participantId, Var.festivalNbr);
4  def temp2 := shared.fan_info(Var.participantId, Var. festivalNbr, Var.band);
5  def temp3 := private.band_info(Var.id, Var.band, "headliner");
6  def description := makeIntensionalDescription( temp1, temp2, temp3 );
7
8  Group( description,
9         'fan,
10        Sequence( Env.fan.show("vote?")@Output(Env.interested),
11                  Filter( {|env| env.find('interested) == "yes"} ),
12                  Env.fan.show(Env.discography)@Output(Env.selection),
13                  SynchronisedTask(
14                     Env.headliner.show(Env.selection)@Output(Env.playlist),
15                     at(time(20,30,0)) ),
16                  CancellingBarrier( at(time(22,20,0)) ),
17                  Env.fan.show(Env.playlist) ) );
```

Listing 5.9: Implementation of the SURA application in NOW.

The code in Listing 5.9 is the implementation of the SURA application in NOW. This scenario is implemented as a Group pattern which wraps a sub workflow. The Group pattern is instantiated with the intensional description (defined on lines 1-6), and has the symbol `fan` as its second argument (line 9).

The intensional description is the same we used earlier: The description consists of three prerequisites using four variables, namely `Var.participantId`, `Var.festivalNbr`, `Var.id`, and `Var.band`. The first argument of a fact matching a prerequisite is a reference to the service that published the fact. Hence, the variable `Var.participantId` is bound to a reference to the service running

on the mobile device of a festival visitor. Similarly, the variable `Var.id` is bound to a reference to a service on the fixed infrastructure that published the fact of type `band_info`. The other variables, `Var.festivalNbr` and `Var.band` are bound to a unique identifier relating to a festival visitor or a band respectively.

The variable used to instantiate the group ensures that a reference to the service (i.e., group member) is available at runtime. The sub workflow that is wrapped by the Group pattern is a Sequence pattern (lines 10-17).

The first activity that must be executed in this pattern implements the sending of a message to each group member, asking whether they want to vote (on line 10). Afterwards, a Filter pattern is used to restrict the group members to only those who were interested in voting (line 11). The Filter pattern is instantiated with a closure, and returns a boolean denoting whether the value of the variable `interested` equals `"yes"`. Note that this variable is added to the data environment as a consequence of executing the first activity of the Sequence pattern (i.e., the variable is specified using `@Output`). Subsequently, the fans who are interested in voting are contacted to cast their votes, given the band's discography.

At half past eight, the results of the votes need to be sent to the headliner such that they can decide the order of the songs that will be played during their show. This is implemented using a Synchronised Task pattern, which is the fourth element of the Sequence pattern (lines 13-15). The Synchronised Task pattern is instantiated with a sub workflow (in our example, a single activity) and a condition. In this scenario, the condition is a time constraint, namely the deadline (`20:30`). The activity wrapped by the pattern implements the sending of the votes to the band, and receiving, as a result, the playlist of their show (line 14). Note that no data merging strategy is used to instantiate the pattern. The abstract grammar, shown in Listing 5.8, shows that the Synchronised Task pattern has an optional third argument, namely the merging strategy used to merge the environments of the instances of each group member. NOW's implementation of the Synchronised Task pattern has a default merging strategy provided, namely the "merge" strategy that accumulates all values of the variables.

The fifth element of the Sequence pattern is a Cancelling Barrier pattern (line 16). This pattern blocks the execution of the individual instances of all group members whose execution reached the pattern until a certain condition is satisfied. In this example, only ten minutes before the concert starts (i.e., at `22:20`) the fans receive the playlist. Therefore, this activity (the sixth element of the Sequence) must be placed after the Cancelling Barrier pattern (line 17).

5.6 Failure Handling

We already described the control flow patterns that are supported by NOW and which allow the orchestration of services in a nomadic network. We also introduced the patterns that allow the orchestration of a group of services in these types of networks. We now present the necessary patterns to allow failure handling by defining compensating actions for specific kinds of failures.

The grammar of the failure handling patterns in NOW is shown in Backus-Naur form in Listing 5.10.

```
<component>     := <activity> | <pattern>
<duration>      := "hours(" <integer> ")" | "minutes(" <integer> ")" |
                   "seconds(" <integer> ")"
<pattern>       := <std_pattern> | <sync_pattern> | <group_pattern> |
                   <trigger_pattern> | <group_sync_pattern> |
                   <failure_event> | <compensation>
<std_pattern>   := "Failure(" <component> ", [" <description>* "])"
<description>   := "Description(" <failure_event> "," <compensation>
                    [ "`group" ] ")"
<failure_event> := "Disconnection()" | "Timeout(" [ <duration> ] ")" |
                   "NotFound(" [ <duration> ] ")" | "Exception()" |
                   "PDisconnection()" | "PTimeout(" [ <duration> ] ")" |
                   "PNotFound(" [ <duration> ] ")" | "PException()"
<compensation>  := "Retry(" [ <integer> "," <compensation> ] ")" |
                   "Rediscover(" [ <integer> "," <compensation> ] ")" |
                   "Restart(" [ <integer> "," <compensation> ] ")" |
                   "RestartAll(" [ <integer> "," <compensation> ] ")" |
                   "Skip()" |
                   "SkipAll()" |
                   "Wait(" <duration> "," <compensation> ")" |
                   "Replace(" <component> ")" |
                   "Alternative(" <component> ")" |
                   "Drop()" |
                   "WaitAndResume()"
```

Listing 5.10: Abstract grammar of failure handling patterns in NOW.

This grammar extends the abstract grammars we presented in Listing 5.1, Listing 5.6 and Listing 5.8. Two new types of patterns are introduced, namely *failure events* and *compensations*. The first of these categories is used to capture specific types of failures, whereas the second category consists of patterns that are used to overcome these failures. As we can see, the category of standard patterns is extended with a Failure pattern. Such a Failure pattern wraps a sub workflow (called `component`) and specifies compensations for specific kinds of failures. This specification is given by a `Description` which maps a (chain of) compensation(s) to a failure event. The failure events that are supported by NOW are disconnections, timeouts, service unavailability and exceptions. Possible compensating actions are

- retrying to invoke the same service;
- rediscovering a (possibly different) service of the same service type;
- restarting the activity of which the execution failed;
- restarting the execution of the sub workflow wrapped by the Failure pattern;
- skipping the activity that failed;
- skipping the entire sub workflow that is wrapped by the Failure pattern;
- waiting a specific time before trying another compensating action;
- replacing the failed activity by executing a sub workflow; and
- replacing the wrapped sub workflow with the execution of another sub workflow;
- dropping a participant from the group members (recall that this a compensating action that can only be used for participant failure events); and
- waiting for a group member to reconnect, upon which the execution can be resumed (this compensation can only be used for participant failure events).

Failure descriptions are used to specify the compensating action for a specific type of failure. Recall that failure handling must also function for the group orchestration patterns provided by NOW. It must be possible to react upon a failure that occurs during the individual process execution of a single member of the group, and, moreover, mechanisms must be provided to detect and handle failures at the group level, and to propagate individual failures to the group level. Therefore, descriptions have an optional third argument, which can be used to specify whether the compensation should have an effect on the execution of the entire group.

5.6.1 Failure Handling for Service Orchestration

Consider the example of updating a screen at the airport with the necessary information (flight number, belt number, estimated duration before luggage is on the belt) so that passengers can retrieve their luggage. By using the failure pattern (drawn as a dashed rectangle wrapping part of a workflow), we can specify compensating actions, as is shown for the trailer service in Figure 5.6.

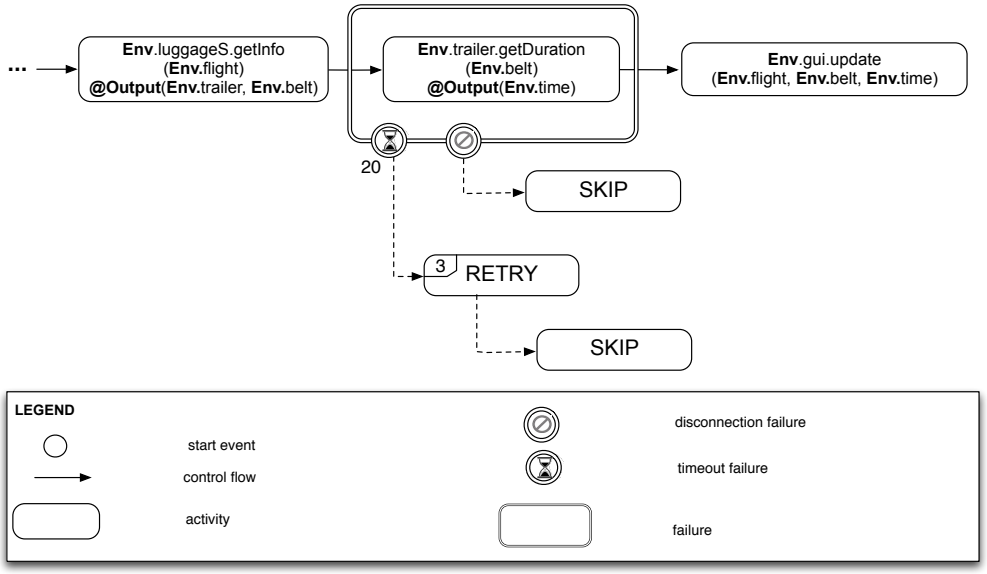


Figure 5.6: Compensating actions specified for different kind of failures.

When the second activity has a timeout (after 20 seconds no reply is returned), we try to resend the message three times. If this is still unsuccessful we move on to the next compensating action, which just skips this activity (so no time gets displayed on the screen). In case of a disconnection, however, the wrapped activity is skipped in which case no time is displayed on the screen. The implementation of this workflow in NOW can be seen in the following code snippet.

```
Sequence( Env.luggageS.getInfo(Env.flight)@Output(Env.trailer, Env.belt),
          Failure( Env.trailer.getDuration(Env.belt)@Output(Env.time),
                   [ Description(Timeout(seconds(20)), Retry(3, Skip())),
                     Description(Disconnection(), Skip()) ]),
          Env.gui.update(Env.flight, Env.belt, Env.time) );
```

Failure patterns can be nested, so different strategies can be formulated on different levels of granularity. A whole workflow can be surrounded by a failure pattern specifying “after a disconnection, wait 20 seconds and then try to rediscover" and smaller parts of this workflow can be wrapped with more specific failure patterns, which possibly override (shadow) the behaviour imposed by the outermost failure pattern. In NOW an inside-out policy is used to determine the compensations that must be executed for an occured failure event.

Recall that the language provides default compensations for failure events. Hence, failures that occur during the execution of a compensation itself are taken care

of by the language. However, application developers need to be careful when specifying compensations for failures. For example, compensating actions are tried indefinitely when no number indicates the number of attempts the compensation should be executed.

5.6.2 Failure Handling for Group Orchestration

We now discuss the different possible compositions of the patterns for group orchestration and failure handling. Recall that there are three possible compositions that allow failure handling for group orchestration. Table 5.2 summarises how these compositions can be implemented in NOW.

Composition	Implementation in NOW
Composition 1 Figure 4.24(a)	`Group( TypeTag, 'varName, Sequence( ..., Failure( ..., [ Description( Disconnection(), RestartAll() ) ] ) ));`
Composition 2 Figure 4.24(b)	`Failure( Group( TypeTag, 'varName, Sequence( ... ) ), [ Description( Disconnection(), RestartAll() ) ] );`
Composition 3 Figure 4.24(c)	`Group( TypeTag, 'varName, Sequence( ..., Failure( ..., [ Description( Disconnection(), RestartAll(), 'group ) ] ) ));`

Table 5.2: Failure handling for group orchestration in NOW.

Table 5.2 shows the implementation of the three possible compositions of failure handling for group orchestrations, where the compensation for a disconnection is

"restart all". As we can see, for the third composition, the Failure pattern needs to be created inside the group pattern. By instantiating the failure description with a third argument, the symbol `'group`, we ensure that the compensating action will have an effect on the execution of the entire group.

5.6.3 Implementing the SWOOP Application in NOW

We present how a scenario that uses these failure handling patterns can be implemented in NOW. The scenario we use is the one of the SWOOP application we presented in Section 2.3.1. This application is used to help organise workshops at the university. This scenario introduces several failure-related concepts, such as the need for automatic failure recovery in case of network failures. Moreover, the SWOOP application exemplifies the necessity for recovery during the orchestration of services in a nomadic network, by allowing the specification of application-specific compensations, both for service orchestration and group orchestration.

Figure 5.7 shows the workflow implementing the scenario described for the SWOOP application.

The workflow implementing the SWOOP application consists of the following patterns (annotated using grey circles):

1. **Failure**: In the scenario of the SWOOP application, a lot of failure handling is described. For instance, in case the mobile device of a workshop assistant disconnects, this is signalled on the computer of the administration desk, such that the assistant can be contacted via email. In order to implement compensations for specific kinds of failures, a Failure pattern needs to wrap the (part of the) workflow that is affected by the compensating action. For this scenario, four Failure patterns are used: one to wrap the entire workflow, one pattern such that individual failures with students can be overcome, and two patterns that wrap a single activity.
   Note that NOW has a default failure handling strategy, hence, it is not necessary that compensations are specified for each of the failure types we support (i.e., disconnection, timeout, exception, and service unavailability). Failure patterns only need to be used in case the default compensation needs to be overridden with an application-specific compensation.

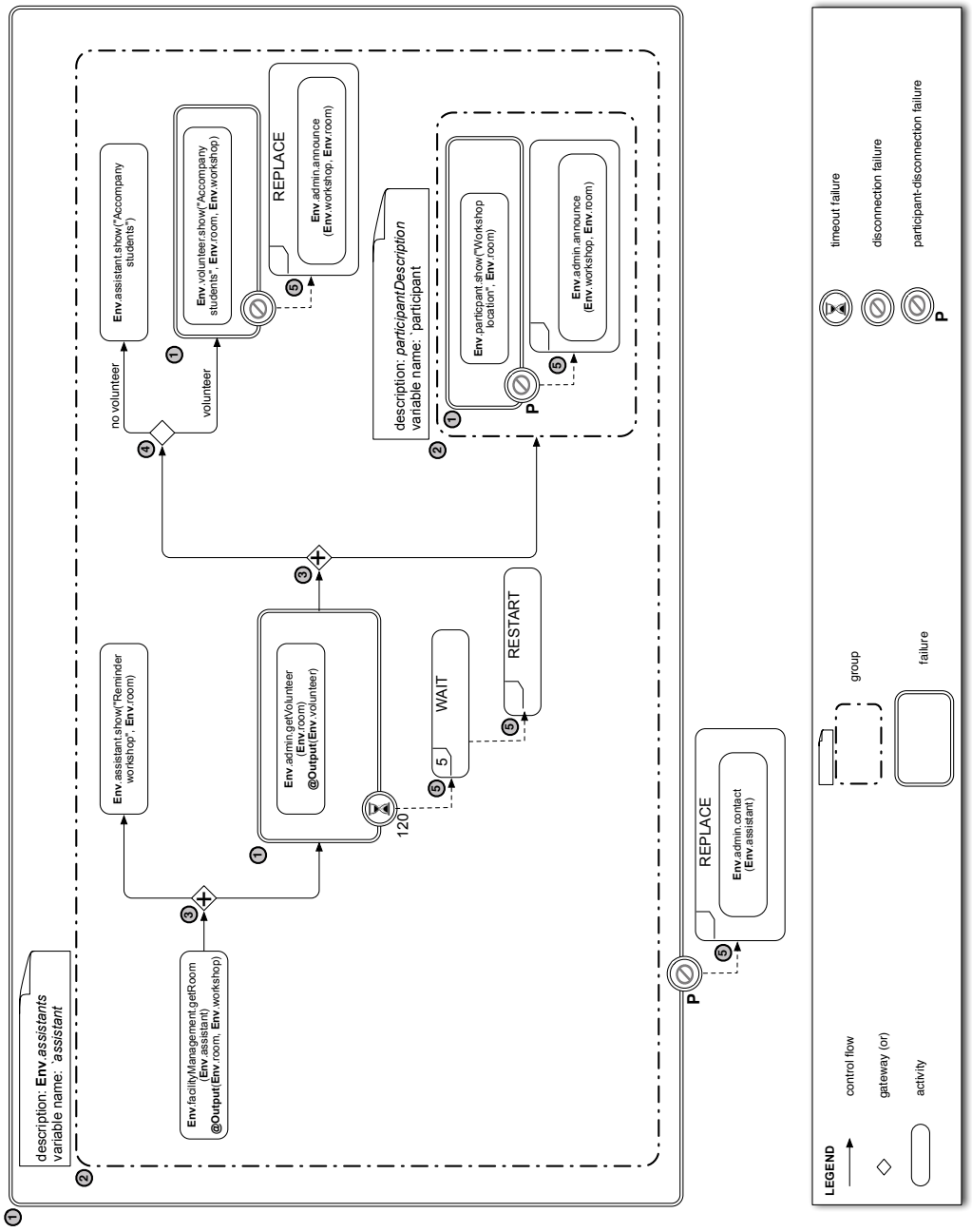


Figure 5.7: Workflow implementing the SWOOP application.

2. **Group**: In the example scenario, two groups of services are addressed. First of all, several tasks are performed for all workshops assistants. Secondly, all students of a particular workshop are contacted such that they are notified of the workshop's location.
   For the first group, an extensional description of the group members is given, i.e., the references to the workshops assistants are stored in the data environment, that is used to start the execution of the workflow.
   For the second group, an intensional description is used, since these group members are not known a priori (i.e., the group members are categorised as user services), and need to be discovered when required.
3. **Parallel Split**: Two Parallel Split patterns are used to implement this application in NOW. The first Parallel Split is used such that the task of reminding the workshop assistant happens in parallel with the rest of the application (searching for a student volunteer, etc.). The second pattern ensures that the task of either informing the workshop assistant or student volunteer happens parallel to informing all students that are registered for the workshop.
4. **Exclusive Choice**: This pattern is used to ensure that the right task(s) are executed depending on whether a student volunteer has been found or not. When no student volunteer has been found, the workshop assistant is informed to guide the students himself/herself. Otherwise, the student volunteer is informed about the room of the workshop.
5. **Compensation**: Several compensating actions are used to build the workflow of this example scenario. For instance, the Alternative compensation is used such that when a disconnection occurs during communication with a single student, an announcement is made. Another compensating action is the Replace compensation, which is used such that in case a disconnection happens during communication with a student volunteer, the persons at the administration desks are informed to make an announcement.

This workflow can be implemented in NOW as is shown in Listing 5.11.

```
1  Failure(
2    Group(
3      Env.assistants,
4      'assistant,
5      Sequence(
6        Env.fMgmt.getRoom(Env.assistant)@Output(Env.room, Env.workshop),
7        ParallelSplit(
8          Env.assistant.show("Reminder workshop", Env.room),
9          Sequence(
10           Failure(
11             Env.admin.getVolunteer(Env.workshop)@Output(Env.volunteer),
12             [ Description(Timeout(seconds(120)),
13                 Wait(seconds(30), Restart())) ]),
14            ParallelSplit(
15              ExclusiveChoice( { |volunteer| volunteer == false },
16               Env.assistant.show("Accompany students", Env.room),
17               Failure(
18                 Env.volunteer.show("Accompany", Env.room, Env.workshop),
19                 [ Description(Disconnection(),
20                     Replace(Env.admin.announce(Env.workshop, Env.room))) ])),
21             Group(
22               participantDescription,
23               'participant,
24               Failure(
25                 Env.participant.show("Workshop location" , Env.room),
26                 [ Description(PDisconnection(),
27                     Alternative(
28                       Env.admin.announce(Env.workshop, Env.room))) ])))))),
29   [ Description(PDisconnection(),
30       Replace(Env.admin.contact(Env.assistant))) ]);
```

Listing 5.11: Implementation of the SWOOP application in NOW.

As we can see in Figure 5.7, the application consists of a Failure pattern that wraps the entire workflow. This Failure pattern is created on lines 1-30 in Listing 5.11. The pattern is instantiated with a sub workflow (created on lines 2-28) and with one failure descriptions (lines 29-30). This failure description is used to specify the application-specific compensating action for a participant disconnection failure. In case a disconnection occurs, with a service that is a member of the group, the compensation defined on line 30 is executed, resulting in replacing the activity that failed with the task of informing the administration desk that the assistant must be contacted.

The sub workflow that is wrapped by the Failure pattern consists of a Group pattern, which is created on lines 2-28. This Group pattern is used to address all assistants that teach a workshop, and is described using an extensional description (defined on line 3). In the data environment that is used to start the workflow's

execution, references to these assistants are stored in a table (the value of the variable `assistants`). The variable that can be used to refer to these services inside the group is `assistant`, as we can see on line 4. The Group pattern is responsible for executing several tasks for all its services (i.e., for all assistants).

In our scenario, a sequence of tasks (contacting facility management, student volunteers, students, etc.) needs to be performed. Hence, a Sequence pattern (lines 5-28) is used to compose the activities. First, facility management is contacted and asked which room is reserved for the particular assistant (line 6). When the room and the workshop are retrieved, several tasks need to be executed in parallel. First of all, the assistant must receive a reminder that he/she must teach a workshop. Secondly, the administration desk needs to search for a student volunteer that can guide the students to the correct location, inform the students, etc. Therefore, a Parallel Split pattern is introduced on lines 7-28.

The first outgoing branch of that pattern constitutes of a single activity, namely sending a reminder to the assistant (see line 8). In the second outgoing branch of the Parallel Split pattern several activities need to be executed. First of all, the administration desk is asked to look for a student volunteer that can guide students to the location of the particular workshop. Afterwards, either students are informed about the workshop's location and either the assistant or a student volunteer need to be contacted. In order to implement this, a Sequence pattern is introduced on lines 9-28.

The Sequence pattern is instantiated with a single activity (line 11), which is wrapped by a Failure pattern, and is followed by a Parallel Split pattern (lines 14-28). The activity that implements the task of asking the administration desk to look for a student volunteer, is wrapped by a Failure pattern that implements application-specific compensations for a timeout failure (lines 10-13). In case a timeout occurs during communication with the administration desk, recovery consists of waiting 30 seconds, before restarting the execution of the failed activity.

The second component of the Sequence pattern is the Parallel Split pattern, which is defined on lines 14-28. That pattern has two outgoing branches, consisting of either an Exclusive Choice pattern or a Group pattern. Depending on whether a student volunteer has been found, either the assistant or that student volunteer needs to be contacted. This is implemented using an Exclusive Choice pattern (defined on lines 15-20). In case no student volunteer was found, the workshop assistant is informed that he/she needs to accompany the students (line 16). Otherwise, the student volunteer is informed of the location of the workshop he/she needs to accompany students to (line 18). Application-specific failure handling is specified

for this activity: when a disconnection occurs during communication with the volunteer, the administration desk is asked to announce the relevant information, such that the student volunteer is informed about his/her work duties (lines 19-20).

The second outgoing branch of the Parallel Split pattern consists of a Group pattern. This pattern is used to address all students that are registered for the assistant's workshop. The intensional definition that is used to instantiate this group, is defined as

```
def shared := createFactSpace("shared");
def temp1 := shared.workshop_participant(Var.participantId, Var.workshop);
def temp2 := shared.workshop_assistant(Env.assistant, Var.workshop);
def participantDescription := makeIntensionalDescription( temp1, temp2 );
```

This intensional description captures those references (`Var.participantId`) to students that are registered to the workshop (`Var.workshop`) of the particular assistant (`Env.assistant`). All these students receive a message, informing them of the workshop's location (line 25). In case a disconnection occurs during the communication with one student, an announcement should be made (once). This is achieved by wrapping the activity with a Failure pattern and specifying a compensating action for a participant disconnection. The compensating action is an Alternative compensation, because the compensation should only be executed once for the entire group. This is implemented on lines 26-28.

5.7 NOW Related to the State of the Art

In this section we describe related work and evaluate existing work using the criteria for orchestration in nomadic networks we presented in Section 2.3. First, we give a survey of workflow languages, followed by an evaluation of existing coordination languages.

5.7.1 Workflow Languages

Workflow languages were introduced to develop business processes. Throughout the years, those languages gained popularity, and right now a variety of workflow languages are available. These languages can be divided into several categories. First, we distinguish traditional workflow languages, such as YAWL [vdAtH05] and WS-BPEL [JE+07], which are concerned with the composition of (web)services. The second category differs from traditional workflow languages by allowing a distributed execution [MWW+98] because the workflow execution engine is distributed. The idea of a federated workflow is introduced by Micro-Workflow [Man00] where sub workflows can be divided in a distributed setting. Another

category is concerned with so-called scientific workflows which are used to describe the process in terms of computations that arise in scientific problem-solving. Scientific workflows systems, like Kepler [LAB+06], differ from traditional ones, as the computations are heavy, complex, use scientific data, and can take a long time to finish. The workflow engine is distributed as well; tasks can be divided among computation nodes in a computer cluster or computing grid. A last category of workflow languages are those workflow languages which are tailored for development in MANETs and nomadic networks.

In this section we give a survey of workflow languages that are relevant for our research. We limit ourselves to the both the traditional workflow languages, as these languages introduce the necessary abstractions for service orchestration, and the latter category that focusses on workflow languages for MANETs and nomadic networks.

Traditional Workflow Languages

Yet Another Workflow Language YAWL [vdAtH05] is developed as a result of the examination of workflow patterns conducted by the Workflow Pattern Initiative. This study showed that support for several workflow patterns were missing, as well as a formal semantics. This observation led to the development of this workflow language. YAWL is implemented on top of high-level Petri nets [vv02].

YAWL has support for basic control flow patterns, advanced branching and synchronization patterns, structural patterns, patterns involving multiple instances, state-based patterns, and cancellation patterns. Therefore, the usage of patterns to describe the control flow of a process ensures that the flow is made explicit.

Just like most workflow management systems, YAWL completely relies on XML-based standards like XPath and XQuery [vvH94]. The workflow language provides both net and task variables, which can be categorised further into input variables, output variables and local variables. *Net variables* are used to store data that needs to be accessed/updated by tasks in a net, whereas *tasks variables* are used to store data that needs to be accessed or updated in the context of an individual instance of a task. YAWL makes the distinction between "*input and output*" variables and "*input only or output only*" variables. In general, data is read from output variables and written to input variables and local data can only be used for net variables. The workflow language supports both internal data transfer by means of data passing between variables, and external data transfer which is accomplished through data interaction between the process and its operating environment (i.e. the engine and

the services). YAWL also supports contextual data, which can be modelled using *worklets*, a workflow specification that is dynamically assigned at runtime.

YAWL has no discovery mechanism and cannot react upon the availability of services in the neighbourhood, since all services are known a priori. Therefore, there is no decoupling in space.

Services are invoked in YAWL using Remote Procedure Calls (RPC), which block the execution of a single activity. However, the language spawns the necessary threads in case multiple activities need to be executed, for instance when a parallel split or multiple instances pattern is used. Each task is executed by a dedicated service. The association between a task and its service can either be done explicitly at design time, or it can be the default Resource Service of YAWL. The service invocation must be synchronous (i.e. request/response). In order to allow asynchronous invocations of services, custom YAWL services must be used. The custom services of YAWL communicate with the execution engine using XML/HTTP messages. When such a service is invoked, it marks its status as "executing" until the execution of the task is finished.

YAWL has built-in support for failure handling for atomic tasks (i.e., a single activity and not a sub workflow). Hence, defining recovery strategies that cover several tasks of the workflow, such as restarting the execution of the entire workflow, is not straightforward. YAWL provides an Exception Service which handles both expected and unexpected exceptions (e.g., events/occurrences not defined in the process model), such that the process can continue unhindered. When an unexpected exception is handled, it becomes an implicit part of the process specification, ensuring a continuous evolution of the process. YAWL has support for ten different kinds of exceptions and provides several actions to handle these failures, either at the level of an activity, a single instance, or for all instances. Amongst these actions are cancelling, suspending, and restarting the execution of the activity, instance, or all instances. Although the workflow language has support for exception handling, there are no ways available to deal differently with network failures compared with other exceptions that occur. The language is able to detect a deadline expiry, but has no provisions to catch disconnections of services.

To summarise, we evaluate YAWL using the criteria we postulated in Section 2.3.

- **time decoupling**: YAWL provides decoupling in time through the usage of a data structure. This data structure, called *worklist handler*, is the component used to assign work to users of the system. Users can accept *work items*[2] through this worklist handler and signal their completion once finished.

[2] A work item is a runtime instance of an atomic task definition.

- **space decoupling**: YAWL cannot react upon the availability of services in the neighbourhood, as all services communicated with must be known beforehand.
- **synchronisation decoupling**: YAWL uses synchronous communication for the invocation of services by default.
- **explicit control flow**: The usage of patterns to describe the control flow of a process ensures that the flow is made explicit.
- **intensional definition**: Intensional definitions in YAWL must be specified relying on the language's data flow mechanism that relies on XML-based standards like XPath and XQuery [vvH94].
- **arity decoupling**: Because the language provides a "multiple instances without a priori run-time knowledge" pattern, a process can be executed multiple times. The number of times the process is executed can however not vary over time.
- **dynamic modification**: YAWL is not equipped with patterns that allow the dynamic modification of the members of a group (i.e., the number of executing instances of a multiple instances pattern).
- **synchronisation mechanisms**: YAWL does not have support for the more advanced group synchronisation mechanisms, like the barrier patterns we propose.
- **automatic failure handling**: Because YAWL only allows failures to be handled on the level of an atomic task, the workflow language has only limited support to automatically handle failures.
- **explicit failure handling**: YAWL only partially enables explicit failure handling, since no compensations can be specified to overcome network failures.

Because YAWL does not have dedicated patterns for the orchestration of a group of services, the workflow language does not support the criteria concerned with failure handling for groups (i.e., individual failure handling, and failure handling for groups).

Business Process Execution Language Orchestrating web services is often achieved using the workflow language BPEL, the Business Process Execution Language [JE+07]. This workflow language provides a standard way to specify business processes using XML-based standards. BPEL makes the distinction between abstract processes and executable processes. An *abstract process* is the specification of a process in BPEL which is not intended to be executed by a workflow engine, whereas *executable processes* provide a full specification of the process and can hence be executed by an engine.

A BPEL process consists of partner links, variables and a main activity. The *partner links* describe the partners (i.e., web services) the process interacts with during its execution. The web service that will be invoked is selected using the service type that is specified by the partner link using a WSDL (Web Service Description Language) interface. Because services are described using WSDL, which specifies amongst other the location of the service, BPEL has no decoupling in space. Services can either be invoked using request-response (`<invoke>` activity) or one-way operations. Asynchronous communication can be initiated using BPEL's `<request>` activity.

Variables enable capturing the process' messages. Such messages are frequently received from partners or must be sent to partners. Furthermore, these variables can hold the state of the process (global data) which is not exchanged with partners.

The main activity that is specified by the process represents the workflow that must be executed. BPEL has support for 20 different types of activities, which are divided in basic activities and structured activities. *Basic activities* (like `<invoke>` and `<reply>`) describe steps of the process behaviour, whereas *structured activities* (for instance, `<sequence>`) are used to describe the control flow of the process. The process' main activity is most often one of the latter category, namely a structured activity composing several activities. So we conclude that BPEL has support for explicit control flow through its structured activities.

BPEL provides fault handlers to react upon "failures", i.e., when a web service returns data other than what was expected. The workflow language makes the distinction between two categories of failures, namely business faults and runtime faults. *Business faults* are generated in case there is a problem with the data that is processed, whereas *runtime faults* are caused when executing a process or web service. The last category of faults are not user-defined and are thrown by the workflow management system. In contradiction to YAWL, BPEL does have support to specify fault handlers for multiple activities. To this end, the language introduces a *scope activity* which is a simple container for activities for which fault handlers and compensations can be provided.

We evaluate BPEL with respect to the criteria we postulated in Section 2.3.

- **time decoupling**: BPEL messages are buffered in a queue, therefore, the workflow language enables decoupling in time.
- **space decoupling**: In BPEL services are described using WSDL, which specifies amongst other the location of the service. Therefore, BPEL has no decoupling in space.

- **synchronisation decoupling**: BPEL's `<request>` activity initiates asynchronous communication, enabling decoupling in synchronisation.
- **explicit control flow**: BPEL has support for explicit control flow through its structured activities.
- **automatic failure handling**: The workflow language has no built-in recovery mechanism to overcome detected failures.
- **explicit failure handling**: Although the language provides support for failure handling by allowing the specification of fault handlers, BPEL has no support for connection-independent failure handling.

To the best of our knowledge, BPEL does not provide abstractions that allow the orchestration of a group of services. Because BPEL only implements the "multiple instances without synchronisation" pattern, it is not possible to model NOW's Group pattern. Therefore, we state that none of the criteria with respect to group orchestration are enabled by BPEL. However, the language does have support for intensional definitions of services, because it relies on XML standards like XPath and XQuery [vvH94]. Because BPEL does not provide abstractions for group orchestration, the two criteria with respect to failure handling for group orchestration (namely individual failure handling, and failure handling for groups) are not supported by this workflow language.

Workflow Languages for Mobile Environments

Collaboration in Ad Hoc Networks CiAN (collaboration in ad hoc networks) [SRG08, Sen08] is a workflow management system for mobile ad hoc networks. CiAN makes a shift from centralised workflow management towards distributed management. CiAN does not only have support for a distributed management system and an appropriate communication protocol to function in MANETs, it also has support to adapt the workflow execution depending on the context of its surrounding environment.

Since the goal is to function in a dynamic network, CiAN employs appropriate communication and coordination protocols. The workflow engine works in a choreographed manner, meaning that no central entity is needed to coordinate its execution. Peltz [Pel03] states that for orchestration the interactions are described from the perspective of one controlling service. For choreography, on the other hand, the description is from a global point of view.

CiAN introduces the concept of a *host*, which constitutes of both the mobile device and its user. Initially all hosts are colocated, and after the execution is started, hosts can be separated. Each host has the following information: a name, a list of offered services, and a schedule containing entries of when the host is

unavailable. Moreover, each has as a local knowledge base where information about other hosts is being stored. As all hosts are connected upon instantiation, these knowledge bases are ensured to contain information for all hosts in the network. Once the execution is started and hosts are no longer guaranteed to be colocated, there can be asymmetric information in the network as updates to hosts are being propagated.

Recall that CiAN functions in a choreographed manner, and hence, no central coordinating entity is available. Therefore, an allocation algorithm is provided which divides the responsibility for each task a priori. CiAN operates in two different modes, namely planning and standard. During the *planning mode*, tasks are being allocated, whereas the *standard mode* is used when an allocated task is being executed by a host.

A workflow specification in CiAN is modelled using a directed acyclic graph. The language has support for 9 control flow patterns defined by Russell [RtHvdAM06], namely all basic control flow patterns and advanced synchronisation patterns. The data that flows throughout the workflow is modelled on the incoming and outgoing edges of each task.

CiAN uses request-response in the form of a SOAP message to invoke services. CiAN relies on Sliver [HGR07] for the invocation of services. Sliver is a a lightweight BPEL execution engine that is developed to function on mobile devices.

CiAN introduces a *host listener* to notify of hosts that come in or out of communication range. Hence, the workflow language can react upon the (un)availability of services in the neighbourhood. However, CiAN has no mechanism to detect (network) failures or exceptions that occur during communication with a service, and hence, does not have any recovery mechanism provided as well.

We summarise by deliberating over the proposed criteria for orchestration in nomadic networks.

- **time decoupling**: CiAN relies on Sliver [HGR07] for its invocation of services. Sliver is a lightweight BPEL execution engine that is developed to function on mobile devices. Just like BPEL, Sliver does enable decoupling in time.
- **space decoupling**: CiAN has no decoupling in space, because all hosts need to be colocated initially.
- **synchronisation decoupling**: CiAN uses request-response in the form of a SOAP message to invoke services. Therefore, the execution of a process is blocked until the service invocation is finished.

- **explicit control flow**: CiAN provides 9 control flow patterns.
- **automatic failure handling**: CiAN provides some (restricted) automatic failure handling through its host listener.
- **explicit failure handling**: CiAN has no mechanism to detect (network) failures or exceptions that occur during communication with a service, nor does it provide any recovery mechanism.

To the best of our knowledge, CiAN does not provide any support for group orchestration. Therefore, all criteria we postulated with respect to group orchestration are not enabled by CiAN. Because no abstractions for group orchestration are provided, CiAN does neither have individual failure handling, nor failure handling for groups.

The Open Workflow Initiative is a project which builds upon the research realised when developing CiAN. Open Workflow [TWRG09] allows the construction of custom, context-specific workflows in mobile ad hoc networks. The language allows workflows to be "plugged in" at runtime in order to acquire a more concrete workflow description. Although the concept of merging workflows dynamically in a mobile environment is very interesting, it is not the focus of our work. Moreover, the short evaluation we performed for CiAN still holds for this workflow language, as it builds upon the same principles. For instance, the allocation algorithm used by Open Workflow is the same that is used by CiAN, entailing that all services are known beforehand, and hence, violating the space decoupling criterion.

WORKPAD WORKPAD [CdRdL+06, CdM+08] is a workflow management system that is developed for mobile ad hoc networks. WORKPAD focusses in particular on the peer-to-peer collaboration between human users in MANETs. The architecture of WORKPAD consists of two layers, namely a front end and back end layer. The *front end layer* comprises several teams of which the team members are connected in a MANET. Each team consists of several devices (called *actors*) which provide several services, and a coordinator device which orchestrates the team members. This coordinator device acts as a gateway to the back end layer. The *back end layer* is a peer-to-peer network that lets teams collaborate. The back end layer is responsible for the coordination and the exchange of information.

To do this, WORKPAD uses a process management system [vv02] to coordinate the tasks of a single team. Therefore, WORKPAD extends the workflow specification with directives allowing tasks in the workflow to pass data to subsequent tasks, steering the execution of the workflow. Each device has a *worklist handler* which stores the assigned task for a particular actor (device of a team member). When

a task is picked by the team member, the worklist handler starts the application that can perform the task. Information is exchanged between the system and the handler by invoking remote methods of web service interfaces.

WORKPAD can propose configurations of services based upon the team members that are reachable (i.e., in communication range). The system is also able to react upon disconnections of team members: When an actor goes out of reach, the system assigns another device to act as a bridge. In order to support this, WORKPAD relies on a central entity to coordinate the mobile devices and to manage disconnections. However, the language has no support to handle reconnections of actors, something which happens frequently in a mobile network.

To summarise, we evaluate WORKPAD using the postulated criteria for orchestration in nomadic networks.

- **time decoupling**: WORKPAD's engine is based on Sliver [HGR07], the lightweight BPEL execution engine for mobile devices. Just like BPEL, Sliver buffers messages, enabling decoupling in time.
- **space decoupling**: WORKPAD relies on RESCUE [JD08], an open-source middleware for communication with services. RESCUE employs a protocol for service discovery using active advertisements sent out by service providers. Therefore, WORKPAD adheres to decoupling in space.
- **synchronisation decoupling**: RESCUE consists of a local database containing an updated view of the environment, i.e., the services that are currently connected. This middleware enables one way invocation of web services through asynchronous requests/responses, enabling synchronisation decoupling.
- **explicit control flow**: WORKPAD uses structured workflows to model processes. Hence, the control flow of a process is made explicit.
- **automatic failure handling**: WORKPAD distinguishes two types of failure events, namely communication failures and (network) node failures. WORKPAD tries to provide strategies to reschedule activities in order to overcome each of these events.
- **explicit failure handling**: The workflow language does not allow the specification of compensating actions for specific failure events.

To the best of our knowledge, WORKPAD does not provide any support for group orchestration. Therefore, all criteria we postulated with respect to group orchestration are not enabled by WORKPAD.

FlowMark The Exotica Research Project [MAGK95, AGK+95] of IBM is mainly focussed on workflows and advanced transaction management. The project wants to support disconnected computing by integrating disconnected clients into a workflow management system. For their research, Exotica/FDMC uses FlowMark [Lut94], a workflow management system of IBM.

FlowMark is based upon a client-server architecture and uses a central object-oriented database to store information about the schema and runtime information of the process. The system provides several clients, namely the *runtime client*, the *program execution client*, and the buildtime client. The *buildtime client* takes care of several tasks, amongst others the definition and creation of the workflow and the assignment of roles. To this end, the buildtime client needs to have knowledge about all clients in the system.

The workflow language provided by FlowMark consists of the following components: processes, activities, control connectors, data connectors, and worklists. *Control connectors* are used to describe the control flow of the processes, whereas *data connectors* are used to specify the data that is needed to start a particular activity.

Worklists are used to determine the work items that are distributed to a particular client. A single work item can be distributed to several clients at the same time, and the server determines the clients an activity (work item) is distributed to. However, the system ensures that a work item is not executed multiple times, by deleting a work item that is selected by one client in the list of all other clients in the system. The invocation of a service can be done either synchronous or asynchronous.

FlowMark supports *disconnected clients*, meaning that a mobile or disconnected user has the capability to process work while not being connected to the FlowMark server. When the client is connected to the server, all work items that require work are available. When the client selects a work item from its worklist, the FlowMark server marks the work item as "running" for that client and "disabled" for the other clients that were assigned to that work item. The client can disconnect, perform the work, and upon reconnection to the server, the completed work items are checked in.

FlowMark introduces the concept of *planned disconnection*: clients that can disconnect for a certain time period and commit to perform work during that disconnection phase. The disconnection is said to be "planned" as both the client needs to notify the server of its intention and both the server and the client need to agree upon the disconnection. In order to allow such disconnected clients, clients should have enough information as they can no longer query the central server. This can either be realised by providing clients with enough information before the discon-

nection, or by allowing clients to perform navigation themselves by transferring parts of the process to the clients. When a client does not perform the activity after a dedicated time period a timeout is signalled to the server, which reacts appropriately upon it.

We evaluate FlowMark running through the postulated criteria (Section 2.3).

- **time decoupling**: FlowMark enables decoupling in time through its support for disconnected clients.
- **space decoupling**: FlowMark's buildtime client assigns tasks to clients. To this end, the buildtime client needs to have knowledge about all clients in the system. Hence, the FlowMark workflow management system has no decoupling in space.
- **synchronisation decoupling**: The FlowMark Definition Language has support to set an activity in asynchronous mode, therefore enabling decoupling in synchronisation.
- **explicit control flow**: FlowMark enables the control flow of processes to be made explicit through its notion of control connectors.
- **automatic failure handling**: By introducing the notion of disconnected clients, the system is able to deal with planned disconnections. However, the system has no mechanisms to cope with spontaneous disconnections which happen frequently in a mobile environment dominated by volatile connections. Hence, we can state that FlowMark only has (restricted) support for automatic failure handling.
- **explicit failure handling**: FlowMark guarantees *forward recovery*, ensuring that a process makes progress when a failure is detected. In order to allow *backward recovery*, Exotica introduces *spheres of compensation* [Ley95] which specify the scope in case of failures. Within the Exotica project, advanced transaction models on top of FlowMark have been developed, such that the workflow can be extended with the definition of a compensating task for each ordinary task [AKA+94, AAA+96].

Concerning group orchestration, to the best of our knowledge, FlowMark does not provide abstractions that allow the orchestration of a group of services. Because FlowMark does not have dedicated patterns for the orchestration of a group of services, the workflow language does not support the criteria concerned with failure handling for groups (i.e., individual failure handling, and failure handling for groups).

5.7.2 Coordination Languages

Coordination languages are concerned with the communication and cooperation between the different entities in a system. These can be situated in a distributed environment or hosted on the same device. Coordination languages implement a coordination model, in contrast to computation languages which implement a computational model. According to [PA98], such a *coordination model* is the glue that binds separate activities into an ensemble. In most low-level mainstream languages like C [KR88] and C++ [Str00], communication is not embedded in the language itself, rather special purpose libraries are used to orchestrate communication which are forced to adopt the paradigm of the language. This is in contrast with coordination languages where the paradigm itself is dedicated to communication and cooperation between processes.

The survey paper on coordination models and languages [PA98] makes the distinction between data-driven and control-driven coordination. The *data-driven* categorisation captures these coordination models that are evolved around the notion of a *shared dataspace*. Communication between processes is achieved through this medium by broadcasting information to this medium, or by copying it. The second category of coordination models, called *control-driven* or *process-oriented*, achieves almost a complete separation between coordination and computation concerns. In control-driven coordination languages, processes communicate by means of interfaces, which are usually referred to as *input or output ports*. Relationships between the processes (producer and consumer) are created using *streams* or *channels* between the output ports of the producer and the input ports of the consumer. Next to these ports, processes use *events* or *control message* to inform other processes of state changes.

In this section we describe these two categories in more detail. We start by discussing the data-driven coordination languages, and present the first genuine coordination language that falls into this category. Thereafter, we describe two popular coordination languages that are sculpted for service orchestration in a mobile environment, and are categorised as control-driven coordination languages.

Data-Driven Coordination Languages

Data-driven coordination languages use a shared dataspace that allows indirect communication between processes. Communication is realised by processes that publish and/or retrieve information from this shared dataspace. This mechanism allows both decoupling in time and space, because processes do not need to be alive at the same time, nor do their identities need to be known a priori.

Historically, Linda [Gel85] is the first coordination language that implements such a shared dataspace. Linda uses the model of *generative communication* for coordination, and needs a host language for its computation. When data needs to be exchanged, a new data object (called a *tuple*) must be created and stored in the shared dataspace (called *tuple space*). A distinction is made between tuples which have solely data fields, and tuples which have at least one field containing executable code. The former kind of tuples are called *passive tuples* whereas the latter are called *active tuples*. The difference between both kinds is that a passive tuple can be stored directly into the tuple space. In contrast, an active tuple contains code which must be evaluated by a special operator to a *passive value*. When all active fields have been processed to passive values, a passive tuple is constructed and inserted into the tuple space. All tuples reside in a tuple space which is accessible by a number of concurrent processes, possibly distributed over different devices. Processes can publish and withdraw tuples from the tuple space, which acts as a shared distributed associative memory. Communication between concurrent processes is handled solely through the tuple space. In order to let two concurrent processes communicate, one process publishes tuples into the tuple space and the other process withdraws them.

Linda has been designed for a distributed environment where there is access to a globally shared tuple space. It provides primitives adopted from the tuple space model for the access to this memory. These primitives can be used to simulate the standard coordination patterns, like semaphores and the rendez-vous pattern. Moreover, these primitives provide additional properties, such as time and space decoupling.

Linda in a Mobile Environment LIME [PMR99] is the first coordination language based on Linda's tuple space model that adapted the model to operate in a dynamically changing network. Linda's tuple space model enables both time and space decoupling, properties which align well with a mobile environment where the network topology is constantly changing. However, the restriction that Linda's tuple space must be globally accessible for all entities in the network does not fit nicely for mobile environments.

In order to overcome this restriction, LIME breaks down the Linda global tuple space into a hierarchy of tuple spaces. Every process has its own tuple space which contains the tuples it wants to share with other processes. The union of all tuple spaces, which resides on the same host, are grouped together in a *host level tuple space*. These host level tuple spaces are merged together in a *federated tuple space*.

This federated tuple space can be compared with the Linda tuple space, though the contents of this tuple space is dynamically changed according to the connectivity of the hosts, as well as upon the arrival of new processes on those hosts. In Linda, the tuple space coordinates the data that is associated with the process communication, whereas LIME uses the tuple space to provide context information depending on connectivity [PMR99].

LIME's mechanism of breaking down the tuple space into a hierarchy of tuple spaces provides a powerful abstraction, because the process does not need to be concerned with the dynamically changing network topology, which is reflected through changes in the tuple space. A LIME *group* consists of a group of devices within one another's range. All tuples of a LIME group are transiently shared, and the tuple space perceived by a mobile device through its *interface tuple space* is the conjunction of all interface tuple spaces in that group.

LIME is evaluated using the criteria for orchestration in nomadic networks we proposed in Section 2.3.

- **time decoupling**: LIME is based upon Linda's tuple space model which enables decoupling in time.
- **space decoupling**: LIME is based upon Linda's tuple space model which enables decoupling in space.
- **synchronisation decoupling**: Synchronisation decoupling is not enabled in Linda's tuple space model, where blocking operations exist to extract tuples from the tuple space. LIME extends the basic tuple space model with *reactions*, which are callbacks that can trigger asynchronously when a matching tuple is asserted in the tuple space. These reactions enable decoupling in synchronisation.
- **explicit control flow**: The usage of callbacks to react upon the assertion of tuples in the tuple space, the control flow of the process is not explicitly modelled.
- **intensional definition**: The language does also not allow intensional definitions of the processes that are allowed to access tuples in the tuple space.
- **arity decoupling**: In tuple spaces, arity decoupling is enabled because a tuple can be read by multiple processes. This is only achieved using read operations (not retract operations), which leave the tuple in the tuple space such that it can be read by other processes [EFGK03].
- **dynamic modification**: There are no ways to restrict the number of processes that read tuples from the tuple space.

- **synchronisation mechanisms**: There are no advanced synchronisation mechanisms available to synchronise the processes that are reading certain tuples.
- **explicit failure handling**: With respect to failure handling, LIME introduces a read-only tuple space, of which the asserted tuples represent information, such as the connected hosts. Connectivity with other hosts can be observers by registering reactions, enabling explicit failure handling.

LIME does provide not provide automatic failure handling, nor failure handling with respect to group orchestration (individual failure handling, and failure handling for groups).

Tuples on the Air In contrast to LIME, coordination in TOTA [MZ04] is not achieved by a tuple space distributed among the mobile hosts in the network. TOTA provides the facilities for tuples themselves to hop from host to host using a migration policy. These tuples do not belong to a specific host but are injected by a host in the network and flood the network according to a pattern specified by the injected tuple.

The network topology of a group of TOTA nodes is organised as a peer-to-peer network where every node has access to a limited number of colocated hosts. Every host can store tuples and provides the necessary support to let the tuples propagate to other connected hosts. A tuple can no longer be viewed as merely data, it is in fact a mobile program that hops from host to host. TOTA tuples consist of both content (list of fields) and a *propagation rule*. The propagation rule describes whether a tuple can be propagated, and also has the opportunity to transform the tuple.

By letting the propagation rules change the tuple information, the passing of tuples is changed from merely making copies to a distributed computation that is achieved by passing the TOTA tuple around. Because of the propagation rules, the distribution of tuples is no longer restricted to the physical network topology but can be extended to any virtual network topology supported by the physical one. For example, limiting the propagation of tuples to a certain distance from the host which published the tuple, can easily be achieved by computing the distance to the original host every time the tuple is migrated. The propagation stops when the tuple's distance is too far from its originating host.

TOTA does not only support active distribution of the tuples, but also dynamic adaptation. As the underlying network topology is changed, these changes are propagated to tuples and tuples can be automatically re-propagated taking into account the changed network. Reconfigurations of the network, like the addition of a host to the network, are propagated by checking the propagation rules of the already stored tuples and the re-propagation of tuples to this newly added host in

the network may be triggered depending on the propagation rules. Using the same mechanism, the distribution of tuples is adapted according to movements of a host through the network.

Communication in TOTA is thus reduced to *injecting* tuples to the network and detecting tuples in the local tuple space. Applications written in TOTA are able to access the local tuple space, publish tuples with a content and a propagation rule, and can be notified about changes in the context.

TOTAM [SGD09] is an extension of TOTA, where the propagation of tuples can be restricted. TOTAM is a tuple space model sculpted towards mobile ad hoc networks and provides a dynamic scoping mechanism that limits the transportation of tuples. TOTAM tuples can be scoped themselves, meaning that these tuples can dynamically adjust their scope when hopping from one host to another. Programmers can scope the tuples by providing a *tuple space descriptor*, and hence, can prevent tuples to be propagated to unwanted hosts. Because this scope is determined before transmission of the tuple, the physical movement of the tuple can be prevented.

TOTAM uses the combination of leasing with a replication-based tuple space model in order deal with both intermittent and persistent failures. In order to deal with intermittent connectivity, TOTAM replicates tuples to other tuple spaces in the network. Applications are, by default, not aware of the intermittent disconnections of other TOTAM systems in the network since the model abstracts the configuration of the network.

However, tuples can contain rules that describe runtime conditions under which tuples should be visible in the receiving tuple space. Dealing with permanent disconnections can be achieved by injecting tuples with a lease, determining how long the tuple must remain in the tuple space.

We evaluate TOTAM using the criteria we proposed for orchestration in nomadic networks.

- **time decoupling**: TOTA's replication-based model provides decoupling in time.
- **space decoupling**: Decoupling in space is naturally supported in data-driven coordination models.
- **synchronisation decoupling**: TOTA(M) uses an intermediate coordinator to transmit messages, and therefore enables decoupling in synchronisation.

- **explicit control flow**: In TOTAM, *reactions* are registered on the tuple space for given templates. Therefore, the control flow of the application is not apparent.
- **intensional definition**: Group orchestration can be realised in TOTAM by means of its propagation protocol. Members of the group can be described by adapting the protocol's `inScope` method. The protocol provides a `doAction` method, which allows the execution of some action on the local tuple space the tuple is injected into. This way, a process can be executed for a group of members satisfying a description.
- **arity decoupling**: TOTAM tuples can adjust their scope when hopping from one host to another. Therefore, the number of hosts communicated with is not known a priori and can fluctuate over time.
- **dynamic modification**: TOTAM's propagation protocol can be adapted such that before transmission of the tuple, the hosts to which the tuple may be propagated can be adapted.
- **synchronisation mechanisms**: TOTAM's tuple space model does not provide the advanced synchronisation mechanisms we propose.
- **automatic failure handling**: TOTAM deals with intermittent connectivity, using replication. However, the language does not provide any default recovery mechanisms for reconnections of TOTAM systems.
- **explicit failure handling**: TOTAM enables explicit handling of failures, through the introduction of leases.

Concerning failure handling for groups, TOTAM does not provide any abstractions that allow recovery for a group of TOTAM systems.

Control-Driven Coordination Languages

Orc Orc [KQCM09] is a programming language with explicit support for service orchestration. The programming language uses a process calculus to express the coordination of different processes. The process definition in this language is simple as the invocations are abstracted and represented by *keywords*, and service composition is specified with *four concurrent combinators*.

The calculus introduces the notion of *sites* to refer to (external) services. Sites can be several types of software components, like web services, JAVA classes, or custom Orc sites. A program communicates with the environment by calling these sites which create channels of communication.

Calling a site constitutes of the following execution steps: invocation of the site, receiving a response from the site, and publication of that result. Those three steps can be interleaved at any moment with other site calls, or can even be delayed

indefinitely. A handle is used to connect a site call to a site return. This handle is blocked waiting for the response from the site call. When there is no response, the call blocks indefinitely. An asynchronous semantics of Orc in which all events (other than external response) are processed as soon as possible is presented by Misra [MC07].

Chromatic Orc [MK09] allows exception handling by introducing throw and try catch expressions that can both run in parallel and hence do not cause termination (criterion 10). The exception handling mechanism introduced in Chromatic Orc differs from how exceptions are handled in traditional sequential programming languages. First of all, multiple exceptions may be raised (possibly in parallel) in the throw clause. Moreover, the catch handler that catches the exceptions is executed in parallel with the expression that raised the exception. This mechanism ensures that throwing an exception does not cause the expression to terminate.

We evaluate this coordination language using the postulated criteria for orchestration in nomadic networks.

- **time decoupling**: Orc uses temporary proxy objects to manage external communication for site calls [AM10]. Such a proxy object serves as a buffer for the site's response. Therefore, Orc adheres to decoupling in time.
- **space decoupling**: The sites that are used by an Orc program are known a priori (for instance through a URL), disabling decoupling in space.
- **synchronisation decoupling**: An asynchronous semantics of Orc is presented, enabling decoupling in synchronisation.
- **explicit control flow**: The Orc programming language has been extended with definitions for several control flow patterns [CPM06]. The language extension has support for 16 control flow patterns that were categorised by Russell [RtHvdAM06]. These patterns are introduced in Orc as reusable definitions, ensuring that they can be used to create larger programs.
- **intensional definition**: X-Orc [MKC07] is an extension of Orc, enabling the coordination language with an XML data model and XML-specific data management capabilities from XQuery. Intensional definitions of services can be achieved using XQuery.
- **automatic failure handling**: Chromatic Orc has no built-in recovery strategies provided.
- **explicit failure handling**: Chromatic Orc introduces abstractions for exception handling, the language assumes a stable network interconnecting the services. Hence, there are no mechanisms provided to capture and recover from network failures.

However, to the best of our knowledge, Orc does not provide abstractions for orchestrating a group of services. Orc does not enable arity decoupling, dynamic modification and group synchronisation mechanisms, nor the criteria for failure handling for group orchestration (i.e., individual failure handling, and failure handling for groups).

Reo Reo [Arb04] is a glue language that allows the orchestration of different heterogeneous, distributed and concurrent software components. Reo has a coordination model wherein complex coordinators, called *connectors*, are composed of simpler ones (where the simplest ones are channels). These different types of coordinators dictate the coordination of the simpler connectors, which eventually coordinate the software components that they interconnect. Reo is based on the notion of *mobile channels*, which consist of two ends (*source* and *sink*) and a *constraint* limiting the data flow that is observed by those ends. The source ending of the channel is used to accept data into the channel, whereas the sink disposes data out of it. Reo provides several types of channels, namely synchronous, asynchronous, and lossy channels [Arb04]. A channel is called *synchronous* when it delays the success of the appropriate pairs of operations on its two ends such that they can succeed only simultaneously. This is in contrast to an *asynchronous channel*, which may have a buffer (to hold the data items it has already consumed through its source, but not yet dispensed through its sink) and can impose a certain order on the delivery of its contents. A *lossy channel* must not deliver all of its received data items, i.e., it may only deliver some and lose the remainder.

Reo's channel-based communication model can be used to model primitives of other communication models (such as message passing, shared spaces, or remote procedure calls). Channel-based communication models adhere to time decoupling.

Reo is already applied in areas like business process modelling [AKM08] and web service composition [MA07]. Reo supports patterns to compose sub-processes and exception handling is possible by using so called *routers* (which can interrupt a process) to propagate cancel messages. Each sub process can be interrupted by such a cancel message or an internal exception.

We run through the criteria for orchestration in nomadic networks we proposed in Section 2.3, and discuss whether or not Reo adheres to it.

- **time decoupling**: Reo's channel-based communication models enables decoupling in time.

- **space decoupling**: Reo uses a smart *registry service* which can be queried to acquire a particular service implementation when needed. Therefore, the coordination language had no decoupling in space.
- **synchronisation decoupling**: Reo's asynchronous channel is used to model asynchronous communication. Each data element enters and exits the asynchronous channels as long as both the input and output are available to do the transmission. Asynchronous channels ensure that the order of the data elements that enter the channel remains consistent. Reo's asynchronous channel enables synchronisation decoupling.
- **explicit control flow**: Tools exist that can translate Business Process Modelling Notation (BPMN) [Obj11], and BPEL models into Reo [AKM08]. In their paper [LN86], Ladani et al. argue that Reo is suited for the orchestration of web services. In this paper, BPEL's control flow patterns are modelled using Reo channels.
- **intensional definition**: Reo allows the orchestration of web services, which are described using its signature, states, and its non-functional properties. A web service's signature is represented by the ports of the service and data constraints. Using these constraints, an intensional description of services can be achieved.
- **automatic failure handling**: When software components in Reo are disconnected, one has to manually invoke the migration of a component to a different node; however the channels connecting the component are automatically rebound.
- **explicit failure handling**: In Reo, failures can also be handled with timed connectors, enabling the explicit handling of failures.

To the best of our knowledge, Reo does not have dedicated abstractions to allow the orchestration of a group of services, as we defined in Section 2.3.2. Therefore, this coordination language does not adhere to the criteria dedicated to group orchestration.

5.7.3 Summary

Table 5.3 summarises our survey of related work. It indicates, for each programming language we discussed previously, whether it adheres to the criteria for orchestration in nomadic networks. In this table we use `yes` to indicate that the languages adheres to the specific criterion, and `no` when it does not. A `+-` sign is used to designate that the language only partially complies with the criterion.

	Service Orchestration				Group Orchestration				Failure Handling			
	1	2	3	4	5	6	7	8	9	10	11	12
Workflow Languages												
YAWL	*Yes*	No	+-	*Yes*	*Yes*	No	No	No	+-	+-	No	No
BPEL	*Yes*	No	*Yes*	*Yes*	*Yes*	No	No	No	No	+-	No	No
CiAN	*Yes*	No	No	*Yes*	No	No	No	No	*Yes*	No	No	No
WORKPAD	*Yes*	*Yes*	*Yes*	*Yes*	No	No	No	No	*Yes*	No	No	No
FlowMark	*Yes*	No	*Yes*	*Yes*	No	No	No	No	+-	*Yes*	No	No
Coordination Languages												
LIME	*Yes*	*Yes*	*Yes*	No	No	*Yes*	No	No	No	*Yes*	No	No
TOTAM	*Yes*	*Yes*	*Yes*	No	*Yes*	*Yes*	*Yes*	No	+-	*Yes*	No	No
Orc	*Yes*	No	*Yes*	*Yes*	*Yes*	No	No	No	No	+-	No	No
Reo	*Yes*	No	*Yes*	*Yes*	*Yes*	No	No	No	*Yes*	*Yes*	No	No
Ambient-Oriented Programming Languages												
AMBIENTTALK	*Yes*	*Yes*	*Yes*	No	*Yes*	*Yes*	No	No	No	*Yes*	No	No

Table 5.3: Survey of related work.

Service Orchestration	**Group Orchestration**	**Failure Handling**
1. time decoupling	5. intensional definition	9. automatic failure handling
2. space decoupling	6. arity decoupling	10. explicit failure handling
3. synchronisation decoupling	7. dynamic modification	11. individual failure handling
4. explicit control flow	8. synchronisation mechanisms	12. failure handling for groups

The following observations can be derived from the survey presented in Table 5.3:

- Workflow languages employ patterns to describe the control flow of the application, and therefore, those languages adhere to criterion 4 (explicit control flow). The category of control-driven coordination languages, for which we surveyed the languages Orc and Reo, also complies with this criterion.
- In general, we can make the observation that the support for group orchestrations is very limited. As is indicated in Table 5.3, only three (out of ten) programming languages enable arity decoupling. This is a necessary requirement to allow the orchestration of a group of services, where the number of participants is not known a priori and can vary over time.
- The languages we scrutinised do not adhere to the two criteria we postulated with respect to failure handling for group orchestration. This lack of support can be explained, because these programming languages only provide partial (or none at all) abstractions that allow group orchestration in a nomadic network.

5.8 Conclusion

In this chapter we introduced the workflow language NOW which is sculpted towards orchestration in nomadic networks. First, we described how the notion of an activity as a placeholder for a service invocation, is supported by NOW. Afterwards, we introduced the data flow mechanism adopted by NOW, which allows data passing between activities in a workflow. Thereafter we presented the patterns that are supported by NOW to allow service orchestration, group orchestration, and failure handling. We also presented NOW from the point of view of the application developer, and showed how these patterns can be composed to implement the scenarios of the applications we presented in Section 2.3.1. We also presented a survey of related work, where we scrutinise existing workflow languages and coordination languages using the criteria for orchestration in nomadic networks, we postulated in Section 2.3. In the next chapter we present the actual implementation of the workflow language NOW and describe how the language can be extended with novel patterns.

6

IMPLEMENTING NOW

Our nomadic workflow language NOW is built as an abstraction layer on top of AMBIENTTALK. Throughout this chapter we explain the implementation of this workflow language. First we show the implementation of an activity (Section 6.1). Secondly, we explain how the data flow mechanism employed by NOW is implemented (Section 6.2). Thirdly, we describe the implementation of the patterns that allow orchestration in a nomadic network. We start by presenting the implementation details of several control flow patterns, and discuss how new patterns can be added to the language (Section 6.3). Subsequently, we describe the implementation of a set of group orchestration patterns (Section 6.4). Thereafter, we discuss the implementation of the failure handling patterns that are provided by NOW (Section 6.5).

6.1 Activities

Before presenting the implementation of an activity, we recapitulate the different execution steps that must be performed when starting an activity (see Section 4.1):

- Service discovery;
- Service invocation;
- Response management.

Recall that service discovery is not necessary when a reference to the service is given. The second execution step invokes the (possibly discovered) service. Service invocation is achieved by sending an asynchronous message to that service such that the processes are not blocked when other services with which they are interacting do not respond. Response management takes care of processing the result of the service invocation (as we explained in more detail in Section 5.3).

The definition of an activity, as we showed in Listing 5.2, returns an AMBIENTTALK object. This object has a `start` method, which implements the different execution steps an activity performs. In Listing 6.1 we show an outline of the implementation of this method.

```
1  /* When the start method is called, it returns a future and stores
2     the future's corresponding resolver, so it can be explicitly
3     resolved at another moment in time */
4  def start(env) {
5     // Creation of an explicit future
6     def [result, resolver] := makeFuture();
7     when: typeTag discovered: { |service|
8        /* Make a first class asynchronous message object, given an
9           operation and arguments */
10       def msg := reflectOnActor().createMessage(operation, arguments,
11                                                 [FutureMessage]);
12       // Send the message object using the <+ operator
13       when: service <+ msg becomes: { |reply|
14          env.insertOutputValues(reply, output);
15          // To explicitly resolve the future
16          resolver.resolve(env);
17       };
18    };
19    result;
20 };
```

Listing 6.1: Start method of the Activity object.

Starting an activity results in obtaining an AMBIENTTALK future, which is resolved once the last execution step of the activity (response management) is finished. This future is defined on line 6 and returned as the result of the method invocation (on line 19).

Depending on the kind of activity, i.e., whether a reference to a service or a service type is given, the service discovery execution step can be omitted. Therefore the implementation we show in Listing 6.1 is only an outline of the implementation of this method, which only applies for activities that are instantiated with a service type, instead of a reference to a particular service. When the activity is instantiated with a service type (i.e., an AMBIENTTALK type tag), a `when: discovered:` event handler is installed to await the discovery of a service of the correct type (as can be seen on line 7).

After such a service has been discovered, the discovered service can be invoked. To this end, an asynchronous message needs to be sent to the service (which is implemented as a remote AMBIENTTALK object, as we showed in Section 5.2). The asynchronous message send is created using the activity's operation and input arguments (lines 10-11). When these arguments are not variables (nor AMBIENTTALK expressions), these arguments are the result of looking up the activity's input parameters in the data environment. The last argument, `FutureMessage`, is used to annotate the message, such that a future is returned when the asynchronous message is sent. The service invocation is implemented on line 13 where the message is sent to the discovered service. In order to await the result of the service invocation, a `when: becomes:` event handler is installed.

Once the result of the service invocation is retrieved, i.e., when the future returned by `service<+msg` is resolved, the last execution step (response management) can be executed. This is implemented on line 14 where the retrieved values are bound to the corresponding output variables and inserted in the data environment.

When this last execution step is performed, the future (created upon starting the method invocation) can be resolved. This is implemented on line 16.

Note that, just like a workflow, an activity can be started multiple times. Every time an activity is started, a new copy (clone) of the activity is created. The fact that every running instance has its own copy of the activity (and patterns, as we explain in Section 6.3) is important because these copies contain runtime information which is specific for the execution of a single instance.

Starting an activity returns an AMBIENTTALK future, which makes it possible to register a `when: becomes:` event handler to await the result of executing an activity.

```
def type := createServiceType(`LuggageHandler);
def act := type.getInfo(Env.flightNr)@Output(Env.trailer, Env.belt);
when: act.start(env) becomes: { |nEnv| ... };
```

In the above code snippet we register such an event handler to await the completion of the execution of the activity (`act`). The result of this execution is a (possibly) updated data environment (`nEnv`).

Before exploiting this registering of event handlers to chain together activities to implement control flow patterns, we give some implementation details on how the data flow mechanism we presented in Section 4.2 is supported.

6.2 Data Flow

The data flow mechanism that is employed by NOW is straightforward: a data environment is passed through the workflow linked to its control flow. The data environment is implemented as an object that is passed when an activity or pattern is started. The futures that link together the different components of a workflow are always resolved with an updated environment, ensuring that the data environment is passed from one activity to another.

This object is composed with a dictionary that contains variable bindings and is used upon service invocation, or updated with new variable bindings as the result of a service invocation. Besides this dictionary, the object implementing the data environment also contains references to several objects, such as the failure descriptions that are at hand. Each workflow instance has its own data environment, and, therefore, it contains all the instance-dependent information that is necessary for the execution of a workflow. The data environment has:

- an id: the unique identifier for this environment. Each time a new instance is started, a new data environment is instantiated with a unique id.
- a dictionary: variable bindings that are used for service invocations.
- failure descriptions: describing the compensations for the failure events NOW supports.
- data merging strategy information: recall that synchronisation patterns are instantiated with a data merging strategy that dictates how the data environments of the pattern's incoming branches need to be merged. This data merging strategy can be implemented as a user-provided function that resolves possible conflicts. NOW provides built-in merging strategies, such as "prioritise", and "random". When such a merging strategy needs to be employed upon synchronisation, the strategy must have access to instance-dependent information.

  For instance, the "prioritise" merging strategy choses the data environment of the preferred incoming branch of the synchronisation pattern. Therefore, each

outgoing branch of a split pattern must be labelled such that upon synchronisation the labels of these branches can be compared with the preferred label that is specified by the "prioritise" merging strategy. This strategy uses the data environment of a specific incoming branch, of which the label is specified, as the resulting data environment after synchronisation. Hence, when a split pattern is started, the data environment used to start each outgoing branch contains the label of that branch. For the same reason, the label of the preferred incoming branch needs to be stored, such that the "prioritise" merging strategy knows which incoming data environment must be chosen by the synchronisation pattern.
NOW also provides a "restore" data merging strategy that, upon synchronisation, restores the data environment that was used to start the split pattern with. This merging strategy is not always desired, because all variable bindings that were added as the result of the execution of the different branches of the split pattern are lost. In order to allow the usage of this strategy, the data environment must also remember the data environment that was passed to the split pattern. Therefore, a reference to this data environment is also stored in the data environment.

In this section, we also describe how data environments are merged by synchronisation patterns. As we already explained in Section 5.3, a data merging strategy can be implemented as a user-provided function that resolves possible conflicts. However, NOW provides four built-in merging strategies, of which we now explain the implementation.

1. **Prioritise one of the incoming branches when resolving conflicts.**
   This merging strategy is implemented as a function which takes a list of data environments of all incoming branches as its single argument and searches the data environment of the preferred incoming branch.

```
def prioritise(envs) {
    def nEnvs := envs.filter: {|e| e.branchNr == envs[1].prefNr};
    nEnvs[1];
};
```

   In order to use this merging strategy, the data environment must keep track of the label of the branch it flows through and know the label of the preferred incoming branch. Therefore, when a split pattern (such as the Parallel Split pattern) is started, each of its outgoing branches receives a copy of the data environment which stores the label of the outgoing branch of the pattern. Note that the label of the preferred branch is stored in the data environment

upon instantiation of the Synchronization pattern, as we explain in Section 6.3.2.

The `prioritise` function iterates over the data environments and filters out the data environments for which the branch label (`branchNr`) does not equal the label of the preferred branch (`prefNr`). This returns a new list `nEnvs` (only containing a single data environment), of which its first element is returned as a result.

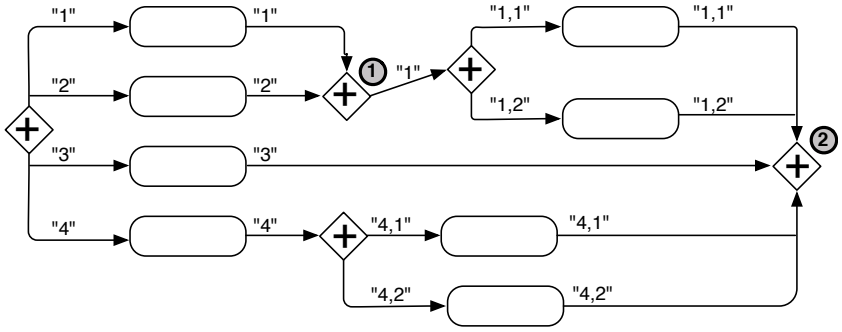


Figure 6.1: Prioritise merging strategy for synchronisation patterns.

Consider the example workflow depicted in Figure 6.1. We use this workflow to explain how the outgoing branches of a split pattern are labelled. As we can see in the figure, this label is a string that is created by concatenating the number of the outgoing branch of a split pattern to the string that represents the label of the split pattern's incoming branch. When the prioritise merging strategy is used for a synchronisation pattern, the programmer can either

- specify the entire string of the label;
- specify a regular expression;

The workflow in Figure 6.1 uses two Synchronization patterns. The merging strategy that is used by the first Synchronization pattern is the prioritise strategy, where the preferred branch must have the label `"1"`. Therefore, the data environment that is tide to the outgoing branch of that Synchronization pattern is labeled with the string `"1"`. When the label of the second Synchronization pattern is

- the string `"3"`, the data environment that is chosen is the one where the outgoing branch label equals this string.
- the regular expression `".*,1"`, there are two data environments that have a matching label. The resulting data environment, after merging, is the first data environment that matches the regular expression.

2. **Pick the data environment of one incoming branch and ignore the others.**
   This merging strategy picks a random data environment of one of its incoming branches. This is implemented by the following function:

```
def random(envs) {
       // Calculate r with: 0.0 <= r < 1.0
       def r := jlobby.java.lang.Math.random();
       def index := (r * envs.length).ceiling();
       envs[index];
};
```

   The `random` function also has a list of data environments as its argument. A random index is computed, and the data environment at that position in the list is returned as a result.

   Please note that through the symbiosis between AMBIENTTALK and JAVA, the `random` function provided by JAVA's Math library can be used.

3. **Merge conflicts into a collection containing the different values.**
   This merging strategy is implemented by the `merge` function:

```
def merge(envs) {
    def nEnv := DataEnvironment.new();
    nEnv.id := envs[1].id;
    nEnv.merge(envs);
    nEnv;
};
```

   This function defines a new data environment that has the same identifier as the identifier of all the data environments of the synchronisation pattern's incoming branches. All the variables and values of the data environments of these incoming branches are then inserted into this newly defined data environment, which is returned as the result of this function.

4. **Remember the "scope" from before splitting and restore it.**
   The `restore` function implements this merging strategy. This function also has a single argument, namely a list of data environments.

```
def restore(envs) {
    envs[1].splitEnv;
};
```

   In order to support this merging strategy, the data environment that is used to start the split pattern needs to be stored. The reference to this data environment is stored (`splitEnv`) in the data environment of each of the split pattern's outgoing branches. The `restore` function just needs to return

this reference by retrieving it from one of the data environments in the list (for example, the first data environment `envs[1]`).

6.3 Service Orchestration

In the abstract grammar (Listing 5.6), we used the proposed categorisation of control flow patterns, namely standard, synchronisation, and trigger patterns (see Section 4.3). Before discussing the differences in the implementation for these types of control flow patterns, we explain how, in general, a control flow pattern is implemented.

Implementation of a Pattern

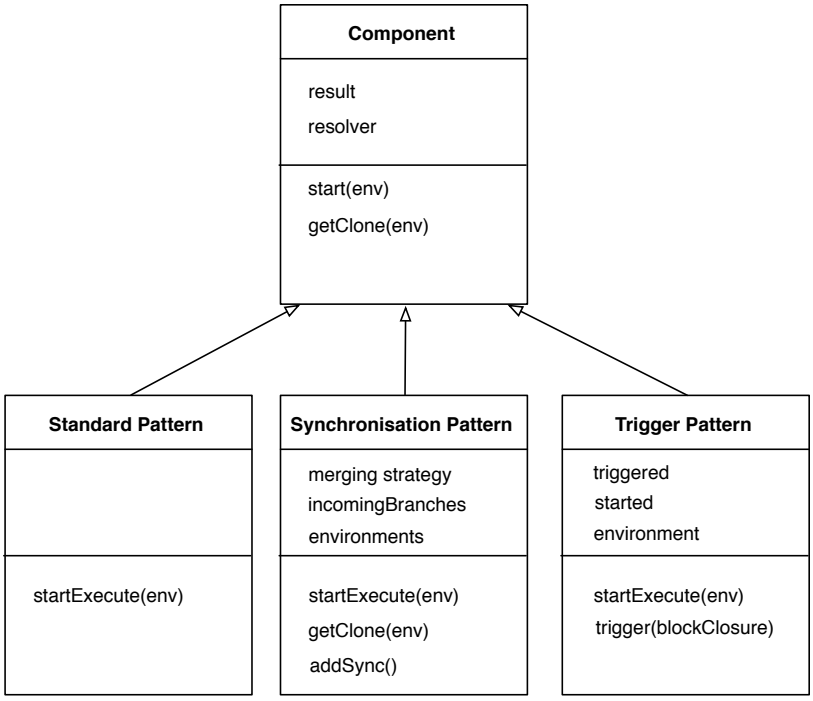


Figure 6.2: Object diagram for service orchestration patterns.

Figure 6.2 shows an object diagram for the control flow patterns. As we can see, each type of pattern (standard, synchronisation, trigger) is implemented as an object which inherits from the `Component` object. This `Component` object has two fields, namely `result` and `resolver`, which store the future and its resolver

that must be returned when a pattern is started. Besides the `start` method, this object also implements a `getClone` method. Because each pattern stores runtime information (such as the number of incoming branches that have been enabled), every workflow instance must have their own copy of the patterns involved. A workflow can be started multiple times, and hence a pattern can be started multiple times as well, for each instance a new copy of the patterns/activities must be made. This way, every instance uses their own objects which store the runtime information of that instance. Such a copy (clone) is retrieved by the `getClone` method. Note that, although we did not mention this explicitly in Section 6.1, this method is also implemented by the `Activity` object.

The difference between the three categories of patterns is discussed in more detail in Section 6.3.1, Section 6.3.2, and Section 6.3.3.

Because the execution of each control flow pattern is different, the actual execution of each pattern is implemented by the pattern itself (`startExecute` method). Therefore, the implementation of the `Component`'s `start` method is as follows:

```
1  def Component := object: {
2      def result;
3      def resolver;
4
5      def start(env) {
6          [result, resolver] := makeFuture();
7          self.startExecute(env);
8          result;
9      };
10
11     def getClone(env) {
12         ...
13     };
14 } taggedAs: [Pattern];
```

Listing 6.2: Implementation of NOW's Component object.

Since we want to interact with activities in the same way as patterns in our implementation, the execution of a pattern must also return a future. When a control flow pattern is started, a future is created (line 6) and, before returning that future (line 8), the actual execution of the pattern is started (`startExecute` on line 7). Recall that the `self`-send is used to indicate the receiver object, i.e., the standard, synchronisation, or trigger pattern itself (see Section 3.3.2).

Now that we have shown the object diagram, and presented an outline of the `Component` object's implementation, we demonstrate how a programmer can extend NOW with the implementation of a new pattern.

```
1  def PatternName(@components) {
2      def obj := extend: Component with: {
3
4          def startExecute(env) {
5                  ...
6          };
7      } taggedAs: [Pattern];
8      obj;
9  };
```

Listing 6.3: Implementing a new pattern.

The code in Listing 6.3 shows the implementation of a new pattern. The constructor function (`PatternName`) has several components as its argument and returns an object, which is tagged with the `Pattern` tag (line 7). The object *must* implement the `startExecute` method which takes a data environment as its argument (lines 4-6). The body of this method implements the execution of all components used to instantiate the pattern. In general, when the execution of all components is finished, the pattern's execution is terminated as well. This rule does not always apply, for instance, a "Multiple Instances without Synchronization" pattern does not require its component to be finished in order to terminate the execution of the pattern itself.

6.3.1 Standard Patterns

In this section, we describe the implementation of the first category of control flow patterns, namely the standard patterns. NOW uses a different categorisation for its control flow patterns than the one proposed by van der Aalst et al. [RtHvdAM06]. The classification employed by NOW is based upon the way these patterns are composed and implemented, and will become clear when we present the implementation.

1) Implementation of the Sequence Pattern

The Sequence pattern is categorised as a "basic control flow pattern" by van der Aalst et al. [RtHvdAM06]. In order to implement this pattern in AMBIENTTALK, we follow the outline of the implementation that is given in Listing 6.3. We define a function, called `Sequence`, that has a collection of components as its argument. This function returns an object that is tagged as a `StdPattern` and implements the `startExecute` method. Listing 6.4 shows the implementation of the Sequence pattern.

```
1  def Sequence(@args) {
2      def obj := extend: StandardPattern with: {
3          def components := args;
4          def instanceComponents := [];
5
6          def startExecute(env) {
7              execute(1, env);
8          };
9
10         def execute(idx, env) {
11             if: (idx <= components.length) then: {
12                 def cmp := components[idx].getClone(env);
13                 instanceComponents := instanceComponents + [cmp];
14                 when: cmp.start(env) becomes: { |nEnv|
15                     execute(idx + 1, nEnv);
16                 };
17             } else: {
18                 super.resolver.resolve(env);
19             };
20         };
21     } taggedAs: [StdPattern];
22     obj;
23 };
```

Listing 6.4: Implementation of the Sequence pattern.

The `startExecute` method of a pattern is used to implement how the pattern's components must be executed (i.e., sequential, in parallel, etc.). For the Sequence pattern, the components must be executed one after the other, and when the last component finishes its execution, the execution of the pattern itself is finished.

As we can see on line 7 of Listing 6.4, the body of the `startExecute` method is a method call. The `execute` method iterates over all the pattern's components and executes one after the other.

Recall that patterns store runtime information (such as the number of times it is started), and therefore, every workflow instance must have its own copy of patterns and activities. So, instead of starting the execution of a component, a clone of that component is taken (line 12) and stored (line 13). Storing the clones of the pattern's component is necessary to allow the cancellation of a component, as we explain later in Section 5.6.

After a clone is taken and stored, that cloned object is started with the data environment (on line 14). In order to await the result of that component's execution, a `when: becomes:` event handler is installed. The result of executing a component is always the reception of an updated data environment. Once such a data environment is received, the next component can be started.

When the last component of the Sequence pattern has finished its execution, the execution of the pattern itself is finished, and the pattern's future can be resolved (see line 18).

2) Implementation of the Multi-Choice Pattern

The second pattern of which we present the implementation is the Multi-Choice pattern. This pattern is defined as "*the divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches*" [RtHvdAM06]. The Multi-Choice pattern's implementation follows the outline that is given in Listing 6.3. Listing 6.5 shows the implementation of the pattern.

```
1  def MultiChoice(@args) {
2      def obj := extend: StandardPattern with: {
3          def components := args;
4          def instanceComponents := [];
5          def indices := [];
6
7          def startExecute(env) {
8              1.to: components.length+1 do: { |idx|
9                  def elem := components[idx];
10                 def block := elem[1];
11                 def method := block.method();
12                 def bindings := env.bind(method.parameters());
13                 if: block.apply(bindings) then: {
14                     indices := indices + [idx];
15                 }; };
16             if: (indices.isEmpty()) then: {
17                 raise: XNoCondition.new("No matching condition found");
18             } else: {
19                 execute(env);
20             };
21         };
22
23
24         def execute(env) {
25             def finished := 0;
26             def clonedEnv := clone: env;
27             indices.each: { |idx|
28                 def cmp := components[idx][2];
29                 cmp := cmp.getClone(env);
30                 instanceComponents := instanceComponents + [cmp];
31                 /* Store the env that is used to start the split pattern.
32                    Needed when a synchronisation pattern uses
33                    the Restore data merging strategy */
34                 clonedEnv.splitEnv := env;
35                 /* Store the number of the outgoing branch.
36                    Needed for the Prioritise data merging strategy */
37                 def label := clonedEnv.branchNr + "," + idx;
38                 clonedEnv.branchNr := label;
```

```
39                  when: cmp.start(clonedEnv) becomes: { |nEnv|
40                      finished := finished + 1;
41                      if: (finished == indices.length) then: {
42                          super.resolver.resolve(nEnv);
43                      };
44                  };
45              };
46          };
47      } taggedAs: [StdPattern];
48      obj;
49  };
```

Listing 6.5: Implementation of the Multi-Choice pattern.

This pattern is instantiated with multiple arguments (corresponding to outgoing branches), where each argument is a list consisting of a block closure and a component. For example,

```
MultiChoice( [ {|total| total == 50}, Sequence(...) ],
             [ {|total| total < 10}, Sequence(...) ],
             [ {|total| total > 10}, Sequence(...) ] );
```

In this example, depending on the value of the variable `total`, which is stored in the data environment, one or more Sequence patterns must be executed. The `startExecute` method implements how these components need to be executed.

First, the indices of those outgoing branches that need to be executed need to be retrieved (i.e., the branches for which the condition evaluates to true). This is implemented on lines 8-15 of Listing 6.5. For each of the arguments that are used to instantiate the pattern, the block closure is applied (with values that are the result of looking up the parameters in the data environment). Only the indices of the arguments for which the block closure evaluate to true are stored in the list `indices`.

After having retrieved the indices of those outgoing branches that need to be executed, the components of those branches can be executed. Executing these components happens in parallel, because service invocations are implemented as asynchronous message sends (see line 13 of Listing 6.1). For each of the retrieved indices, the corresponding component is executed by performing the following steps:

- a clone of the data environment is taken [line 26];
- a clone of the component is taken [line 29];
- the cloned object is stored (added to the list `instanceComponents`) [line 30];

- the environment that is used to start the Multi-Choice pattern is stored in the cloned data environment (this is possibly needed when a synchronisation pattern, following this pattern, uses the "restore" data merging strategy) [line 34];
- the label of the outgoing branch is stored in the cloned data environment (this is possibly needed when a synchronisation pattern, following this pattern, uses the "prioritise" data merging strategy) [line 38];
- the cloned object is started [line 39]. In order to await the component's result, a `when: becomes:` event handler is installed (line 39). When all components of the outgoing branches that needed to be executed have finished their execution, the execution of the Multi-Choice pattern is finished. At that moment, the pattern's future can be resolved (line 42).

Recall that each outgoing branch of a split pattern (a pattern with multiple outgoing branches) has its own local copy of the data environment that is used to start the split pattern. Therefore, a clone of the data environment is taken before starting the execution of the component.

Because a split pattern is possibly followed by a synchronisation pattern, the necessary precautions need to be taken to assure the correct functioning of the data merging strategies used by the synchronisation patterns. One of these merging strategies is restoring the data environment before splitting. Therefore, each copied data environment stores this environment, such that, when needed, the merging strategy can restore the data environment to the stored scope. Another data merging strategy (called "prioritise") prefers the data environment of a specific branch, depending on the label of the outgoing branch of the preceding split pattern. In order to allow the correct working of this strategy, the label of each outgoing branch of a split pattern needs to be stored. This is achieved by storing the label of the branch in its own local data environment.

3) Implementation of the Interleaved Routing Pattern

The third standard pattern of which we show the implementation is the Interleaved Routing pattern. This pattern is categorised as a "state-based pattern" and has the following definition: "*Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e., no two tasks can be active for the same process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated.*" [RtHvdAM06].

In order to implement this pattern in AMBIENTTALK, we follow the outline of the implementation that is given in Listing 6.3. We define a constructor function, called `InterleavedRouting`, that takes a collection of components as its

argument. This function returns an object that is tagged as a `StdPattern` and implements the `startExecute` method. Listing 6.6 shows the implementation of the Interleaved Routing pattern.

```
1  def InterleavedRouting(@args) {
2      def obj := extend: StandardPattern with: {
3          def components := args;
4          def instanceComponents := [];
5          def indices := [];
6
7          def startExecute(env) {
8              def index := searchIndex();
9              execute(index, env);
10         };
11
12         def searchIndex() {
13             /* Compute random index: 0 < idx < components.length
14                 Ensure that the component (of that index) has not been
15                 executed.
16                 List indices stores index of all executed components */
17             ...
18         };
19
20         def execute(idx, env) {
21             def component := components[idx].getClone(env);
22             instanceComponents := instanceComponents + [component];
23             when: component.start(env) becomes: { |nEnv|
24                 def index := searchIndex();
25                 if: ! (indices.length > components.length) then: {
26                     execute(index, nEnv);
27                 } else: {
28                     super.resolver.resolve(env);
29                 };
30             };
31         };
32     } taggedAs: [StdPattern];
33     obj;
34 };
```

Listing 6.6: Implementation of the Interleaved Routing pattern.

The implementation of the `startExecute` method resembles the one of the Sequence pattern we showed in Listing 6.4. The difference is the order in which the components are executed. For the Sequence pattern, the components are executed one after the other, following the ordering used to instantiate the pattern. The components of the Interleaved Routing can be executed in a random order.

As we can see in Listing 6.6, the function `searchIndex` (line 12) is used to compute a random index. The component at that place is executed, and when its execution is finished, the next index is computed. When all components have finished their execution, the pattern's execution is finished and its future can be resolved (line 28).

Note that the execution of a single component results in cloning that component, storing the cloned object and starting the execution of the cloned object (line 21-23). Again, the result of the execution of a component is awaited by installing a `when: becomes:` event handler (line 23).

6.3.2 Synchronisation Patterns

The second category of control flow patterns we define is the category of synchronisation patterns. Synchronisation patterns are control flow patterns that have multiple incoming branches. As we already mentioned, synchronisation patterns differ from standard patterns in the way data flow is handled. Recall that synchronisation patterns need to specify a data merging strategy which specifies how the data environments of the pattern's incoming branches must be merged. Moreover, the composition of synchronisation patterns requires special care, i.e., synchronisation patterns must be composed using so-called connections (see Section 5.4.1). In this section we discuss a third difference of these patterns to standard patterns, concerning cloning the patterns.

We now present how a synchronisation pattern can be implemented in AMBIENTTALK. As we could already deduct from the object diagram we showed in Figure 6.2, the prototype object of a synchronisation pattern consists of additional fields and methods, compared to the prototype object of a standard pattern.

- First of all, a synchronisation pattern needs to know its number of incoming branches. This is required in order to know when synchronisation succeeds (when all incoming branches have been enabled, when `n` out of `m` incoming branches have been enabled, etc.).
- Secondly, the data environments of incoming branches that are enabled are stored, such that they can be merged when necessary. In order to merge these data environments, a data merging strategy needs to be provided.
- Thirdly, special care must to be taken when cloning a synchronisation pattern. Recall that patterns and activities store runtime information, and every instance needs to have its own copies of those objects. For activities and standard patterns it suffices to take a clone *each time* such a component needs to be executed. However, this cloning strategy does not apply for synchronisation pattern because multiple incoming branches can refer to the *same* synchronisation pattern (i.e., the same synchronisation pattern's execution can be started multiple times for one instance).

Synchronisation patterns store runtime information, such as the number of times the pattern is started. This information is required because this pattern may, by definition, only proceed once all its incoming branches have been enabled (i.e., at the moment the pattern is started as much times as it has incoming branches). Because several instances may be executing the same workflow, it is important to distinguish the enablements of the branches per instance. Therefore, a clone of the Synchronization pattern can *only* be taken when the pattern is started *the first time per instance*. It is possible to retrieve the instance for which the pattern is started by looking at the data environment that is used to start the pattern. Recall that the data environment has a unique identifier which is different for every instance. So, to recapitulate, for synchronisation patterns a clone of the pattern is only taken once for each instance, namely for the first incoming branch that is enabled.

In Listing 6.7 we show the implementation of the Synchronization pattern in AMBIENTTALK.

```
1  def Synchronization(cmp, strategy := {|envs| merge(envs)}, prefNr := "1") {
2      def obj := extend: SynchronisationPattern with: {
3          def incomingBranches := 0;
4          def nextComponent := cmp;
5          def instanceComponent;
6          def environments := [];
7          /* Dictionary: key = env id, value = clone of this syncPattern
8            When multiple instances wraps a sync pattern, for each instance
9            a new clone of a syncPattern needs to be made:
10           MIWithPrioriDTKnow( Sequence(..., Synchronization, ...) ).
11           The instance of the synchronization can be retrieved by
12           checking the id of the data environment. */
13         def instances := Dictionary.new();
14
15
16         def getClone(env) {
17            def res :=instances.find(env.id);
18            if: (res == nil) then: {
19                res := clone: component;
20                instances.insert(env.id, res);
21            };
22            res;
23         };
24
25
26         def addSync() {
27             incomingBranches := incomingBranches + 1;
28         };
29
30
31
```

```
32          def startExecute(env) {
33              environments := environments + [env];
34              /* Store the preferred nbr of the incoming branch,
35                  needed for preference data strategy */
36              env.prefNr := prefNr;
37              execute(env);
38          };
39
40          def execute(env) {
41              if: (environments.length == incomingBranches) then: {
42                  def nEnv := strategy(environments);
43                  def cmp := nextComponent.getClone(env);
44                  instanceComponent := cmp;
45                  when: cmp.start(nEnv) becomes: { |newEnv|
46                      super.resolver.resolve(newEnv);
47                  };
48              };
49          };
50      } taggedAs: [SyncPattern];
51      obj;
52  };
```

Listing 6.7: Implementation of the Synchronization pattern.

The Synchronization pattern is implemented as a function that returns an object that is tagged as a `SyncPattern`. This function has three arguments, namely the component that must be executed once synchronisation succeeds, an optional data merging strategy, and an optional argument representing the label of the preferred outgoing branch of the last split pattern proceeding this pattern. Note that a default data merging strategy (`merge`) and a default prioritisation label (`"1"`) are given.

We first explain the execution of this pattern. When an incoming branch is enabled, i.e., when the pattern's execution is started with a data environment, that data environment is stored such that it can be merged with all incoming data environments when needed (line 33 in Listing 6.7). In order to allow the correct working of the "prioritise" data merging strategy, the preferred label must be stored in the data environments. Afterwards it is verified whether synchronisation succeeded, which is the case when all incoming branches are enabled (i.e., the number of stored data environments equals the number of incoming branches). When this is the case, all incoming data environments are merged using the provided merging strategy (line 42). Thereafter, the execution of the component (that must be executed after synchronisation) is started with that merged data environment (line 45). Note that first a clone of that component needs to be taken and stored (lines 43-44). The Synchronization pattern's execution is finished at the moment this component's pattern's execution is finished. So, when an updated data environment is returned, the Synchronization's future can be resolved (line 46).

Besides the `startExecute` method, the implementation of this object also consists of an `addSync` method (lines 26-28). This method is used to let the connection pattern inform the synchronisation pattern of its number of incoming branches. Whenever the execution of a Connection pattern (that wraps a synchronisation pattern) is started, the `addSync` method is executed.

Other than the two methods we just mentioned, a synchronisation pattern also implements the method `getClone` (lines 16-23). A synchronisation pattern stores its own clones in a dictionary (`instances`). When the execution of a synchronisation pattern is started, it is first verified whether a new clone must be taken or not. A clone must only be taken when the first time the pattern is started for a new instance. Because each instance has a unique identifier, it suffices to verify whether a clone was already taken previously for that identifier. When no such clone was taken before (i.e., it is the first time an incoming branch of the pattern is enabled for that specific instance), a new clone is taken and stored in the dictionary.

6.3.3 Trigger Patterns

The third category of control flow patterns we define is the category of trigger patterns. Trigger patterns are control flow patterns that react upon external events. van der Aalst et al. [RtHvdAM06] define two patterns, namely the Transient Trigger pattern and Persistent Trigger pattern, which are both provided by NOW.

Trigger patterns also differ from standard patterns and synchronisation patterns concerning the runtime copies (i.e., cloned objects). Recall that when a pattern is executed, each of its components is cloned. A special policy is required for synchronisation patterns, as these patterns have multiple incoming branches that refer to the *same* pattern. Concerning trigger patterns, yet another approach must be taken. Because trigger patterns must be accessible at runtime, these objects cannot be cloned. Therefore, the `getClone` method of the `Component` object (given in Listing 6.2) has a dedicated implementation for trigger patterns.

Note that when a trigger pattern is used inside a multiple instance pattern, only one trigger pattern is used for *all* running instances. As trigger patterns are not cloned, every instance refers to the *same* trigger pattern. So, once a single trigger pattern receives an external event, it influences the execution of all instances.

Listing 6.8 shows how the Persistent Trigger pattern can be implemented in AMBIENTTALK. This pattern is defined as "*The ability for a task to be triggered by a signal from another part of the process or from the external environment. These*

triggers are persistent in form and are retained by the process until they can be acted on by the receiving task." [RtHvdAM06].

```
1  def PersistentTrigger(cmp) {
2      def obj := extend: TriggerPattern with: {
3          def component := cmp;
4          def instanceComponent;
5          def triggered := false;
6          def started := false;
7          def startedEnv;
8
9          def startExecute(env) {
10             execute(env);
11         };
12
13         def trigger(closure) {
14             triggered := true;
15             closure();
16             if: started then: {
17                 executeCmp(startedEnv, instanceComponent);
18             };
19         };
20
21         def execute(env) {
22             def cmp := component.getClone(env);
23             instanceComponent := cmp;
24             executeCmp(env, cmp);
25         };
26
27         def executeCmp(env, cmp) {
28             if: triggered then: {
29                 when: cmp.start(env) becomes: { |nEnv|
30                     started := false;
31                     super.resolver.resolve(env);
32                 };
33             } else: {
34                 started := true;
35                 startedEnv := env;
36             };
37         };
38     } taggedAs: [TriggerPattern];
39     obj;
40 };
```

Listing 6.8: Implementation of the Persistent Trigger pattern.

The execution of the component that is used to instantiate the Persistent Trigger can only take place once an external event is (or has been) received. This is shown on line 28 in Listing 6.8 where an `if` test is performed to verify whether the `trigger` method has already been called. When the external event was already received, the component can be started (line 29). Otherwise, the necessary information to start the execution of the component is stored, such that, when an

event is received, the execution can start immediately. Therefore, it is indicated that the pattern's execution is started, and the data environment used to start the pattern is stored (line 34-35).

As we already mentioned, a trigger pattern has an extra `trigger` method (lines 13-19). This method has a single parameter `closure`, which has as its default value an empty block closure. This argument can be used to perform a task when the external event triggers the pattern. When the `trigger` method is called, it is indicated that an external event has been received (line 14). The AMBIENTTALK block closure is also executed, and when the pattern was already started before, the component can start its execution. In the other case, the pattern must just wait until it is started.

6.4 Group Orchestration

In this section we give the implementation of two patterns for group orchestration. We start with the implementation of the Filter pattern, which is categorised as a standard pattern, and subsequently present the implementation of a group synchronisation pattern, namely the Synchronised Task pattern. Before presenting the implementation of these patterns, we explain how group members can be defined.

6.4.1 Definition of Group Membership

Group membership can be either described extensional or intensional. In order to allow an intensional definition of group members, NOW uses the logic coordination language CRIME. In this section we first describe CRIME, before explaining how the logic coordination language is integrated in NOW.

CRIME

CRIME's implementation of the federated fact spaces is achieved by a distribution architecture similar to the one used by LIME [PMR99]. The Fact Space Model of CRIME provides a logic coordination language for reasoning about context information that is represented as facts in a federated fact space. This logic language uses the forward chaining strategy for deriving new conclusions as this data-driven technique is very suitable for the event-driven nature of CRIME. The use of a *forward chainer* on the other hand is useful when applications need to reason over a fluctuating distributed knowledge base. Forward chaining is a data-driven reasoning strategy, this implies that, in contrast with the backward chaining strategy, the inference engine is triggered automatically when changes to the knowledge base are

made. The main difference with the backward chaining strategy is the behaviour when new data becomes available. Instead of starting from scratch, the inference engine builds its proofs bottom-up, resulting in a rederivation of only those parts affected by the appearance of the new data. The benefit of using this strategy in a distributed context is that irrelevant changes to the knowledge base are filtered out in the first step of reasoning: Filtering is done by checking the prerequisites of the rules, if none of these prerequisites match with the new information, this can be dismissed.

The Fact Space Model provides a logic coordination language for reasoning about a perceived environment. As this environment is a mobile ad hoc network, devices can go out of range due to the transient connectivity of the network. Such disconnections result in the retraction of facts, namely the quantified facts of the other device.

Integration of CRIME in NOW

In order to use CRIME for the intensional definitions of group members, both the fixed infrastructure and the services residing in the nomadic network must be equipped with a CRIME engine.

Fixed infrastructure In order to allow group orchestration, the backbone of the nomadic network, which is responsible for the execution of the workflow, needs to be adapted. First of all, the backbone needs to have its own CRIME engine that can be used to capture facts that are published by other CRIME engines in the neighbourhood. Secondly, all Group patterns need to be notified when a new service is discovered, or when a service reconnects or disconnects, as can be seen in the following code snippet.

```
crimeEngine.goOnline();
whenever: Service discovered: { |service|
   ...
   // Inform all executing groups of the discovery of a new service
   whenever: service disconnected: {
      ...
      // Inform executing groups of disconnection of a service
   };
   whenever: service reconnected: {
    // Inform executing groups of reconnection of a service
      ...
   };
};
```

As we can see in the code snippet, the backbone's CRIME engine is instructed to go online, meaning that it can be discovered by nearby engines and can itself

discover other engines. In order to inform groups (of which the execution is ongoing) of changes in the network the backbone needs to register event handlers that are triggered upon discovery, disconnection and reconnection of a service in the network. Therefore, each group must register itself to the backbone upon starting its execution. That way, when an event handler is triggered, the fixed infrastructure can iterate over these group observers and inform them of the event that happened.

For example, when a new service is discovered, all group patterns (either normal Group patterns or Snapshot Group patterns) are informed of the newly discovered service. These group patterns can react upon this discovery in an appropriate way: if the service satisfies the group's description, the service can join the group members (on the condition that the group pattern is not a snapshot group of which the execution was already started).

Services In order to allow the exchange of facts, all services must also be equipped with a CRIME engine. In Section 5.2 we explained how a service in NOW can be implemented. The programmer just needs to define an object (recall that it is actually an isolate) with its appropriate fields and methods. NOW provides a function, called `applicationService`, that takes, amongst others, care of publishing the service on the network. This function is also responsible for maintaining a CRIME engine and publishing and retracting the service's facts.

In Listing 6.9 we present the implementation of a service that asserts a fact in its CRIME engine. Note that this implementation is an extension of the one given in Listing 5.3 where no facts were published. Here, the service publishes one fact, which is defined on line 9.

```
1  deftype LuggageHandler <: Service;
2
3  def service := isolate: {
4      def companyName := "Aviapartner";
5      /* ... other fields */
6
7      def init(cn) {
8          self.companyName := cn;
9          fact("luggage handler", "Aviapartner");
10     };
11
12     // Methods
13 };
14
15 applicationService(LuggageHandler, service);
```

Listing 6.9: Implementation of a service that asserts a fact in its CRIME engine.

6.4.2 Patterns for Group Orchestration

Before presenting the implementation of a subset of NOW's group orchestration patterns, we show the object diagram for these group-related patterns. Figure 6.3 depicts this object diagram. As we can see, two new types of patterns are added, namely group patterns and group synchronisation patterns. Both types of objects are implemented as objects that inherit from the `Component` object.

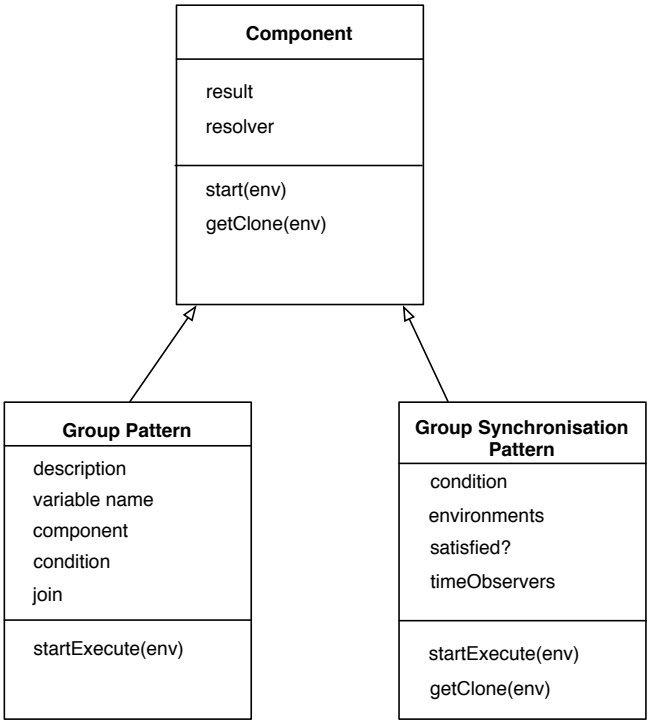


Figure 6.3: Object diagram for group orchestration patterns.

Group synchronisation patterns differ from (regular) synchronisation patterns, because group synchronisation patterns need to be shared for all members of a group. Regular synchronisation patterns are patterns that require special attention in the way that they are composed (because they have multiple incoming branches), and in the way these patterns need to be cloned. Recall that for each workflow instance, only the first enablement of an incoming branch may result in taking a clone of the pattern. Cloning group synchronisation patterns requires a different strategy, because these patterns are shared for members of the same group. Hence, when a group synchronisation pattern is started, it must be verified whether a clone was

already taken for (another) individual instance of that group. Only when this has not happened, the object needs to be cloned.

In the remainder of this section we present the implementation of two patterns for group orchestration. First, we show the implementation of the Filter pattern, which is categorised as a standard pattern, and afterwards describe the implementation of a group synchronisation pattern.

1) Implementation of the Filter Pattern

Restricting the group members during the execution of a group pattern can be realised using a Filter pattern. This pattern is not categorised as a group synchronisation pattern, because a filter pattern cannot block the execution of instances, and the pattern is not joint for all instances of a group. A Filter pattern can only cancel the execution of an instance, for which the pattern's condition is not satisfied. Therefore, the filter pattern does have an influence on the group: The number of group members can be altered by the Filter pattern. The implementation of the Filter pattern is given in Listing 6.10.

```
1  def Filter(condition) {
2      def obj := extend: StandardPattern with: {
3
4          def startExecute(env) {
5              if: condition(env) then: {
6                  resolver.resolve(env);
7              } else: {
8                  env.group.decrease();
9              };
10         };
11     } taggedAs: [StdPattern];
12 };
```

Listing 6.10: Implementation of the Filter pattern.

The function that is used to define a Filter pattern, has one argument, namely the pattern's condition, and returns an object that is tagged as a `StdPattern` (line 11). This object contains one methods, namely the `startExecute` method that must be implemented for each pattern in NOW.

A Filter pattern is started with an incoming data environment. The execution of that pattern verifies whether the pattern's condition is satisfied, and when this is the case, the pattern's execution is finished. When the condition is not fulfilled for that environment (i.e., for that specific group member), the execution of that instance is terminated (the pattern's future is not resolved), and the number of group members is decreased. This is implemented on lines 4-10 in Listing 6.10.

2) Implementation of the Synchronised Task Pattern

We now present the implementation of a group synchronisation pattern. The Synchronised Task pattern is a pattern for group orchestration that performs synchronisation of the execution of all instances of a group.

Group synchronisation patterns differ from traditional synchronisation patterns, because group synchronisation patterns break the boundaries of a single instance. As we already discussed, outgoing branches of a split pattern can point to the same synchronisation pattern. In this case, each outgoing branch has a reference to the same synchronisation pattern. Therefore, for a single instance multiple references to a synchronisation pattern can exist. However, each workflow instance has its own synchronisation patterns.

A group synchronisation pattern is a bit more complex, because these patterns are employed inside a Group pattern. When the execution of such a Group pattern is started, multiple instances are executed (namely one for each group member). Group synchronisation patterns are used to synchronise these different instances, and, therefore, break the boundaries of a single instance.

Moreover, group synchronisation patterns must maintain a collection of observers, i.e., objects that are interested in the time when the pattern is started for the first time. Recall that group synchronisation patterns are instantiated with a condition. An example of such a condition is `After( minutes(10), barrier )`, denoting that the execution of the instances can continue 10 minutes after the first instance has reached the group synchronisation pattern.

The implementation of the Synchronised Task pattern is given in Listing 6.11.

```
1  def SynchronisedTask(component, condition, strategy := merge) {
2      def obj := extend: GroupSynchronisationPattern with: {
3          def joinAllowed? := { |_| false };
4          def resultEnvs := [];
5          def resolvers := [];
6          def satisfied? := false;
7          /* Dictionary: key = group, value = clone of this syncPattern
8           Needed in order to ensure that all participants of a group refer
9           to the same SynchronisedTask pattern.
10          When a multiple instances pattern wraps a group pattern, for each
11          instance, a new clone of the SynchronisedTask pattern needs
12          to be made. For example,
13          MIWithPrioriDTKnow(Group(tag, ...,
14                                   Sequence(..., SynchronisedTask, ...)),
15                             3)
16          with 2 services of type 'Tag' that are discovered at runtime.
17          The Synchronised Task should be the same for each group, hence,
18          there should only be 3 different SynchronisedTask clones.
19          The object-instance of the SynchronisedTask can be retrieved
20          by checking the group, which is stored in the data environment. */
```

```
21          def instances := Dictionary.new();
22          def instanceCmp;
23
24          def getClone(env) {
25              def res := instances.find(env.group);
26              if: (res == nil) then: {
27                  res := clone: self;
28                  instances.insert(env.group, res);
29              };
30              res;
31          };
32
33          def startExecute(env, resolver) {
34              if: ! satisfied? then: {
35                  resolvers := resolvers + [resolver];
36                  when: super^execute(env) becomes: { |reply|
37                      if: reply then: {
38                          env.getGroup().setParticipants(futures.length);
39                          // Merge all incoming data environments
40                          def mergedEnv := DataEnvironment.new();
41                          mergedEnv.merge(resultEnvs, true);
42                          // Execute component once with the merged
43                          // data environment
44                          def cmp := component.getClone(env);
45                          def instanceCmp := cmp;
46                          when: cmp.start(mergedEnv) becomes:  { |newEnv|
47                              /* New variable bindings need to be added to
48                               the incoming data environments, before
49                               resolving the futures */
50                              def diff := newEnv.difference(resultEnvs[1]);
51                              def idx := 0;
52                              resolvers.each: { |resolver|
53                                                  idx := idx+1;
54                                                  def nEnv := resultEnvs[idx];
55                                                  nEnv.merge([diff]);
56                                                  resolver.resolve(nEnv); };
57                          };
58                      };
59                  };
60              };
61          };
62      } taggedAs: [GroupSyncPattern];
63      obj;
64  };
```

Listing 6.11: Implementation of the Synchronised Task pattern.

A Synchronised Task pattern is defined as a function that has three arguments (the component that must be executed, a condition, and a data merging strategy) and returns an object that is tagged as `GroupSyncPattern` (line 62). The object has the following fields:

- `joinAllowed?` (line 3): closure that evaluates to a boolean. When a new service is discovered, the group pattern is notified of its discovery. The group pattern subsequently verifies whether letting the new service join the execution is still useful by checking for all group synchronisation patterns, of which the execution is started, whether joining is still allowed.
- `resultEnvs` (line 4): list containing the data environments of instances that reached the pattern.
- `resolvers` (line 5): list containing the resolvers of all instances of the group that reached the pattern.
- `satisfied?` (line 6): boolean: false when the pattern's condition is not yet fulfilled, true otherwise.
- `instances` (line 21): dictionary containing the instances of this object for a specific group pattern.
- `instanceCmp` (line 22): used to store a reference to the cloned component (i.e., sub workflow that is wrapped by the Synchronised Task pattern).

The object has two methods, namely `getClone` and `startExecute`. Starting the execution of the Synchronised Task pattern results in verifying whether the pattern's condition is fulfilled. In case the pattern's condition is already fulfilled (the value of `satisfied?` is true), the instance for which the pattern's execution is started cannot continue executing. Recall that the Synchronised Task pattern is by definition cancelling, meaning that once the pattern's condition is fulfilled, instances that arrive later at the pattern cannot continue their execution.

Otherwise, i.e., the value of `satisfied?` is false, the resolver of the future is stored, such that it can be resolved once the condition is satisfied. Checking whether the condition is already satisfied, is implemented by the `execute` method, which is part of the `SynchronisationPattern` object. When the condition is fulfilled, i.e., when the value of the method call to `execute` is received (the variable `reply` on line 36), the execution of the component that is wrapped by the Synchronised Task pattern can be started.

First, the number of group members is restricted to the number of instances that reached the pattern before the pattern's condition is satisfied (line 38). Moreover, the data environments of all these instances are merged (lines 39-40). The resulting data environment is used to start the execution of the pattern's component. Recall that before starting the execution of a workflow component, a clone of that component is retrieved and stored (lines 44-45).

A `when: becomes:` event handler is installed to await the result of starting the component that is wrapped by the Synchronised Task pattern (line 46). The variable bindings that are added to the data environment during the execution of this component, need to be retrieved so that they can be added to the data environments

of the instances of all group members. The execution of *all* instances continue with these (possibly) extended data environments (lines 52-57). So, after the execution of the pattern's component is finished, there are again several instances executing.

6.5 Failure Handling

NOW employs a failure handling mechanism that allows to wrap a sub workflow and specify compensating actions for specific failure events. The implementation of this mechanism exploits NOW's data flow mechanism, where a data environment is linked to the control flow of the workflow. As we already mentioned in Section 6.2, this data environment contains the failure descriptions, which specify the compensations for failure events.

Whenever a Failure pattern is encountered, these stored failure descriptions should be updated with the ones defined by the pattern, and when the execution of the Failure pattern is finished they need to be restored. Consider the example we show in Figure 6.4 where two Failure patterns are used to wrap part of a workflow.

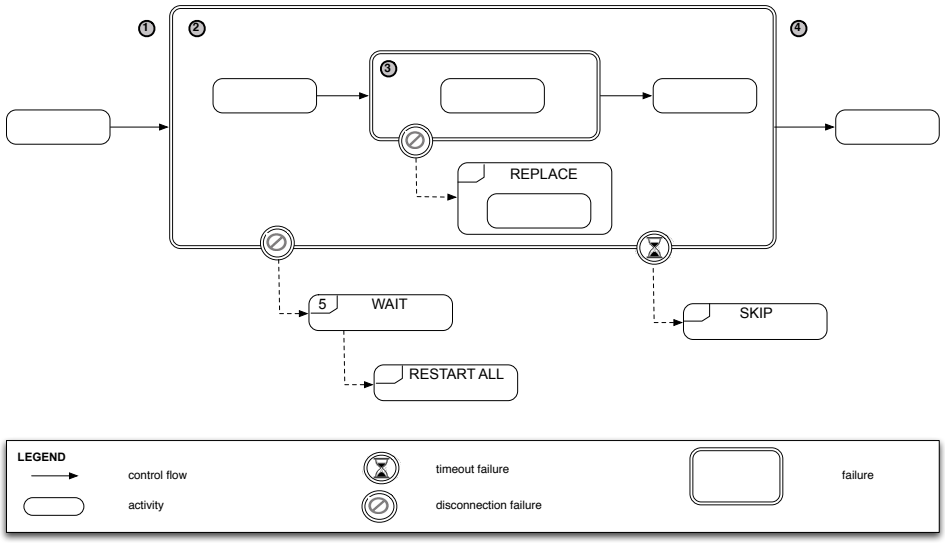


Figure 6.4: Application-specific failure handling by nesting Failure patterns to override (default) compensations.

The failure descriptions that are stored in the data environment are summarised in the following table:

	position 1	position 2	position 3	position 4
notFound	Rediscover	Rediscover	Rediscover	Rediscover
disconnection	Restart	Wait(5) - RestartAll	Replace	Restart
timeout	Retry	Skip	Skip	Retry
exception	Retry	Retry	Retry	Retry

Recall that timeout failure types are used to capture message timeouts, i.e., messages are annotated with a duration and when no result is received after this dedicated time period, a timeout failure is signalled.

For every position in the workflow (marked by the grey circles depicted in Figure 6.4), the compensating action for a failure event is given. For position 1 and position 4, which represent activities in the workflow that are not wrapped by a Failure pattern, the default compensating actions apply (see Section 4.5.1).

Activities that are wrapped by the outermost Failure pattern (position 2) execute application-specific compensating actions for the disconnection and timeout failure events. In case of a disconnection, the execution waits for 5 seconds, before restarting the sub workflow that is wrapped by the Failure pattern. When a timeout occurs during the execution of an activity, the activity that failed is skipped.

A second Failure pattern is used to override the compensation for a disconnection failure that is defined by the outermost Failure pattern. This Failure pattern specifies that when a disconnection failure event occurs during the execution of the activity, placed inside the innermost Failure pattern (position 3), the compensation that must be executed replaces the failed activity with a sub workflow. Note that the compensating action that is executed in case of a timeout failure, is the action that is specified by the outermost Failure pattern.

Please remark that, for conciseness, we omit the compensations for the participant-failures and only present the compensating actions for the (normal) failure events.

6.5.1 Automatic Failure Handling

In Section 6.1 we presented the implementation of an activity's `start` method, which corresponded to the lifecycle diagram of an activity, as shown in Figure 4.1. We presented an updated lifecycle diagram for an activity in Figure 4.20, where failure handling is added to ensure that a failed activity can automatically recover from failures that occur during its execution. In this section we show the implementation of an activity's `start` method, which corresponds to this extended lifecycle diagram.

In order to implement this extended lifecycle diagram, several extra event handlers need to be installed. First of all, it must be possible to catch timeout exceptions and exceptions that are raised by a service during invocation. Secondly, an event handler must be registered in order to observe a possible disconnection of a service. Lastly, in order to detect a service unavailability failure, an event handler that is triggered after a specific time must be installed. The code of the `start` method, implementing these extra event handlers, is given in Listing 6.12.

```
1  def start(env) {
2      def [result, resolver] := makeFuture();
3      def msg := createAsyncMsg(operation, arguments);
4      def duration := getTimeoutDuration(env);
5      // When a timeout is specified, create async msg with @Due annotation
6      if: ! (duration == nil) then: {
7          msg := createTimeoutMsg(operation, arguments, duration);
8      };
9
10     def discoverSub := when: typeTag discovered: { |service|
11         def invocationSub := when: service <+ msg becomes: { |reply|
12             disconnectionSub.cancel();
13             unavailableSub.cancel();
14             env.insertOutputValues(reply, output);
15             resolver.resolve(env);
16         } catch: Exception using: { |exception|
17             disconnectionSub.cancel();
18             unavailableSub.cancel();
19             if: (is: exception taggedAs: TimeoutException) then: {
20                 timeoutOccurred(service, env);
21             } else: {
22                 if: (is: exception taggedAs: ServiceException) then: {
23                     exceptionOccurred(service, env);
24                 };
25             };
26         };
27
28         def disconnectionSub := when: service disconnected: {
29             discoverSub.cancel();
30             unavailableSub.cancel();
31             // In order to be able to resume when wanted
32             env.getGroup().savePoint(env, self);
33             disconnectionOccurred(service, env);
34         };
35     };
36
37     duration := getNotFoundDuration(env);
38     def unavailableSub := when: duration elapsed: {
39         discoverSub.cancel();
40         disconnectionSub.cancel();
41         notFoundOccurred(env);
42     };
43
44     result;
45 };
```

Listing 6.12: Start method of the Activity object with extra event handlers installed for failure handling.

As we can see in the code in Listing 6.12, the following event handlers are installed:

- `when: discovered:` (line 10): This event handler observers the discovery of a service with the correct service type (i.e., the correct `typeTag`). When such a service is found, the second execution step of the activity can be performed, namely the invocation of the service that has been found. Recall that the discovery execution step must only be executed for activities that are instantiated with a service type, and not for activities that have a reference to the service that must be invoked.

- `when: becomes: catch:` (line 11, 16): This event handler observes two events that are necessary for the correct functioning of an activity's execution. First of all, the `when: becomes:` event handler is installed to await the result of the service invocation. Secondly, possible exceptions that can be raised must be caught by this event handler. The exceptions that are caught are either classified as `TimeoutExceptions` or `ServiceExceptions`. A timeout exception is raised when a service does not respond within a given time period (specified by annotating the asynchronous message send with `@Due`) after the service has been invoked. A service exception can be raised by the service to signal some kind of exception.

  When this event handler is triggered, the other event handlers that are installed need to be cancelled (lines 17-18). This is necessary in order to ensure that only one compensating action is executed. When we would not cancel the other event handlers, it is possible that after having triggered this event handler with for example a `TimeoutException`, the service disconnects. This would trigger the `when: disconnected:` event handlers, resulting in executing the compensating action for a disconnection, while the compensation for a timeout failure is also being executed.

- `when: disconnected` (line 28): This event handler is installed in order to react upon the disconnection of the service. Just like exceptions, when a disconnection event is signalled, the other event handlers (`invocationSub` and `unavailableSub`) must be cancelled such that only one compensating action is executed, namely the compensation that is specified for a disconnection failure event.

- `when: elapsed:` (line 38): The `when: elapsed:` event handler is used to signal that a service cannot be found within a given time period. The time period is specified by the failure description for the `NotFound` failure

event. When no failure description is given for this type of failure event, the default failure description is used, which specifies a time period of 60 seconds.

As we already mentioned for the previous two event handlers, when an event handler is triggered, the other two event handlers must be cancelled such that they cannot be triggered. This is needed in order to guarantee that only one compensating action is executed, namely the compensation specified for the failure event that is detected first.

6.5.2 Patterns for Failure Handling

NOW introduces several new patterns in order to support failure handling. On the one hand, there is the Failure pattern that is used to define the "scope" of the failure handling mechanism, i.e., the activities upon which the compensations (can) have influence. On the other hand, there are patterns that implement the compensating actions. In this section we show the implementation of one of these compensations, namely the Alternative compensation. The implementation of the other compensations is similar to the one we present here.

Cancellation

Executing some compensating actions, such as Skip and Alternative, requires the execution of activities and/or patterns to be cancelled. To this end, both the `Activity` and `Component` object implement a `cancel` method. Cancelling an activity results in ruining the activity's future, such that it can no longer be resolved. Cancelling a pattern is implemented by cancelling all the pattern's components.

Implementation of the Alternative Compensation

Alternative is used to execute a sub workflow as a compensation for a failure of an activity. The difference between the Alternative and Replace compensation is the fact that the Alternative compensating action does not only replace the activity of which the execution is failed, but replaces all activities that are wrapped by the Failure pattern.

The compensation is implemented as a function that has a single argument, namely the sub workflow that must be executed, and returns an object. The object has one method, namely `start`, similar to the implementation of activities and other patterns. This method has four arguments, namely the data environment, the activity of which the execution failed, the failure description (containing the failure event of which the compensation is Alternative), and the service that was invoked.

Executing the Alternative compensation results in executing the sub workflow specified by the compensation. Note that the execution of the remainder of the sub workflow that is wrapped by the Failure pattern is cancelled.

The implementation of this compensation is given in Listing 6.13.

```
1  def Alternative(component) {
2      object: {
3          def start(env, activity, failureDescription, service) {
4              def failureEvent := failureDescription.failureEvent;
5              // Drop the participant of the group
6              env.getGroup().dropMember(env, service);
7              // Cancel the remainder of the sub workflow.
8              failureDescription.getFailurePattern().cancel(env);
9              component.start(env);
10         };
11     } taggedAs: [AlternativeType];
12 };
```

Listing 6.13: The Alternative compensating action.

As we can see in Listing 6.13, starting the execution of the compensating action results in cancelling the activities wrapped by the Failure pattern that are succeeding the failed activity. Afterwards, the sub workflow that replaces the execution of these cancelled activities is started (line 9). When the execution of this sub workflow is finished, i.e., when a future is resolved, the execution of the remainder of the workflow can proceed.

Cancelling the activities succeeding the failed activity is implemented on line 8. The failure description, of which the compensating action is being executed, stores a reference to the Failure pattern that wraps the failed activity. This enables the remainder of the sub workflow inside that Failure pattern to be cancelled.

Besides cancelling the activities of the sub workflow that is wrapped by the Failure pattern, and executing the sub workflow that needs to replace these cancelled activities, the Alternative compensation action also influences the execution of Group patterns. As we can see in the implementation of the Alternative compensation, we see that the compensating action drops the participant from the group. We explain why this action implements the correct enactment, depending on how the Failure pattern and Group pattern are composed. The following compositions are possible:

- **There is no Group pattern involved.** In this case, there are no effects that need to be enacted upon.

  The data environment that is passed throughout the remainder of the workflow stores a reference to the Group pattern that is being executed. When

no such pattern is executed, a reference to a dummy Group object is stored. Dropping a participant from that group does not have any side effects.

- **The activity that failed is wrapped by a Failure pattern, which is defined inside a Group pattern.** In this case, the compensating action only affects the execution of an individual group instance. So, when the Alternative compensation is executed, the group member for whom the execution failed, should be removed from the group's members.

  Dropping the participant from the group, as is done on line 6, implements the correct behaviour.

- **The activity that fails is wrapped by a Group pattern, which is defined inside a Failure pattern.** Alternative is categorised as a compensating action that has an effect on the entire wrapped sub workflow. Therefore, when an activity fails, the Alternative compensation cancels the sub workflow that is wrapped by the Failure pattern. In this case, the Group pattern is part of that sub workflow, and hence, the execution of the Group pattern is cancelled.

  So, dropping the group member from the group is not really necessary, but does not cause any problems.

6.6 Conclusion

In this chapter we have shown how patterns for orchestration in nomadic networks can be implemented as an extra layer of abstraction on top of the ambient-oriented programming language AMBIENTTALK. We have shown how the execution of an activity can be implemented by installing the necessary event handlers for the discovery of a services, and to await the result of the service invocation. We also presented the implementation of the data environment and explained the information it must store, and discussed the implementation of NOW's built-in data merging strategies. This chapter also described how the programmer can add extra patterns to the language and showed the implementation of a subset of NOW's patterns. We showed implementations of patterns for service orchestration, group orchestration and failure handling.

7

NOW UP AND RUNNING

The nomadic workflow language we presented in this dissertation is related to research domains such as ubiquitous computing and ambient intelligence. These research domains are very active and serve a wide range of applications. In order to implement these kinds of applications for environments with volatile connections, application developers can depend on languages like AMBIENTTALK, where network failures are tackled at the heart of their programming model. As a validation of our work we show that implementing applications for a nomadic network is still not straightforward in a language like AMBIENTTALK. We highlight some problematic properties (such as maintainability and readability) of programs written in an event-driven style as in AMBIENTTALK and show how workflow abstractions can be used to overcome these.

In this chapter we first discuss the implementation of the three example applications we presented in Section 2.3.1. In Section 7.1 we discuss the implementation of the iMPASSE application, where we focus on the composition of control flow patterns. Thereafter, we analyse the implementation of a second scenario, namely the SURA application, which is revolved around orchestrating a group of services (Section 7.2). The third scenario we review is centred around the detection and handling of failures, for service orchestration as well as group orchestration (Section 7.3). In Section 7.4 we present the results of experiments used to measure the overhead of introducing workflow abstractions on top of AMBIENTTALK and also measure the overhead introduced by failure detection. Afterwards, we show the results of these experiments conducted for the three example applications we used in this dissertation.

7.1 Application stressing Service Orchestration

Our first example application implements the scenario of the airport's iMPASSE application, which is described in Section 2.3.1. This application focuses on the composition of control flow patterns. For conciseness we omit the detection and handling of failures, which is explained in detail by a subsequent example application (in Section 7.3).

In this section we discuss the code complexity of the implementations of this scenario in AMBIENTTALK and in NOW. Recall that we already showed parts of the direct-style implementation in AMBIENTTALK in Section 3.8, whereas the implementation in NOW is presented in Section 5.4.2.

The implementation of this example application is more compact in our nomadic language compared to a direct style implementation in AMBIENTTALK (27 LoC in NOW compared to 93 LoC in AMBIENTTALK). Moreover, there is less nesting of code in the implementation in NOW.

An unwanted property of AMBIENTTALK is the fact that the application logic is divided amongst several event handlers that can be triggered independently of one another [CM06], since the language has an event-driven architecture. The control flow of an application is thus no longer determined by the programmer but by external events. This phenomenon is known as *inversion of control*. This makes the event-driven programming style not always straightforward or suitable for large-scale applications [HO06]. As a library of this ambient-oriented programming language, NOW hides these event handlers for the application developers.

Moreover, as we can see in the code excerpt in Listing 3.5, the implementation of the three branches of the parallel split are very similar. The only difference between them is the name of the service that must be discovered and the asynchronous message send (lines 29-37, 38-56, and **??-??** in Listing 3.5). There is no way this can be abstracted easily in AMBIENTTALK, since this is conceptually the asynchronous version of initialising a reference and sending it a message.

From this we can conclude that the implementation in NOW is shorter and contains less nested code. This is a result of employing control flow patterns that ensure that the control flow and fine-grained application logic are not interwoven.

7.2 Application stressing Group Orchestration

The second application, called SURA, presented in Section 2.3.1 focusses on group orchestration. In this scenario, the fans of a festival's headliner need to be contacted, such that they can vote and influence the songs that will be played during

the show. In Section 5.5.2 we already presented the implementation of this application in NOW. Implementing this application in plain AMBIENTTALK may or may not be achieved by using AMBIENTTALK's language construct "ambient references" (see Section 3.7). In this section we first present an implementation that does not use this language construct. Afterwards, we discuss an implementation of the SURA application using ambient references.

In order to address a group of services in AMBIENTTALK, references to these services need to be stored. When a service of the right type is discovered, its far reference needs to be manually added to this set, and upon disconnection needs to be removed. The implementation of the SURA application is shown in Listing 7.1.

```
1  def fans := [];
2  def selections := [];
3
4  def observer := whenever: FestivalVisitor discovered: { |visitor|
5      when: visitor<-isFan(band) becomes: { |isFan|
6          if: isFan then: {
7
8              when: visitor<-vote() becomes: { |interested|
9                  if: interested == "yes" then: {
10                     fans := fans + [visitor];
11                     when: visitor<-select(discography) becomes: { |sel|
12                         selections := selections + [sel];
13                         if: (selections.length == fans.length) then: {
14                             decidePlaylist();
15                         };
16                     };
17                 };
18             };
19         };
20     };
21 };
22
23 def decidePlaylist() {
24     def curTime := jDate.new();
25     def duration := calculateTimeDiff(curTime, contactBandTime);
26     when: duration elapsed: {
27         observer.cancel();
28         when: headliner<-show(selections) becomes: { |playlist|
29           curTime := jDate.new();
30           duration := calculateTimeDiff(curTime, contactFansTime);
31           when: duration elapsed: {
32              fans.each: { |fan| fan<-show(playlist); };
33           };
34         };
35     };
36 };
```

Listing 7.1: Implementation of the SURA application in AMBIENTTALK.

As we can see, on line 4 of Listing 7.1 a `whenever: discovered:` event handler is installed to capture all services with type tag `FestivalVisitor`. When such a service is discovered, the asynchronous message `isFan` is sent (line 5) to the service. The reply of this message is a boolean indicating whether the festival visitor is a fan of the band, or not. When the festival visitor is a fan of the headliner, the far reference to that visitor (`service`) is added to the table `fans`, which is defined on line 1.

Afterwards, the fan is contacted and asked to select songs from the band's discography (line 11), and when a selection is returned as a result of the asynchronous message send, it is added to the table `selections`, which stores the selected songs of all fans (line 2).

At a specified moment in time (i.e., at 20:30) those votes need to be sent to the headliner. A `when: elapsed:` event handler is installed on line 26, to await the time (`contactBandTime`) upon which the fans' votes can be sent. At that moment, no more fans are allowed to vote, and hence, the `whenever: discovered:` event handler is cancelled (line 27).

The final playlist is sent to all fans who voted at a specific moment in time. Therefore, a second `when: elapsed:` event handler is installed on line 31. The application terminates after the playlist is sent to all fans who voted (line 32).

The implementation we just presented, does not use ambient references. An ambient reference can be used to designate a group of services that are within communication range. The implementation of the SURA application in AMBIENTTALK that uses ambient references is given in Listing 7.2.

```
1  def fans := ambient: FestivalVisitor where: {|v| v.fanInfo == "Kassabian"};
2  def handle := fans<-vote()@[All, Sustain, Reply];
3
4  def observer := whenEach: handle.future becomes: { |fan|
5      if: ! (fan == nil) then: {
6          interestedFans := interestedFans + [fan];
7          when: fan<-select(discography) becomes: { |selection|
8              ...
9          };
10     };
11 };
```

Listing 7.2: Implementation of the SURA application in AMBIENTTALK using ambient references.

On line 1 of Listing 7.2 an ambient reference is defined with a filter. The filter specifies that the service's field `fanInfo` must equal `Kassabian`. On line 2 an asynchronous message is sent to this ambient reference. The message is one-to-many (`@All`), may be delivered indefinitely until explicitly retracted (`@Sustain`), and is two-way, meaning that a result must be returned (`@Reply`).

Using an ambient reference to address the set of services that satisfy a description (in this example, all festival visitors that are a fan of Kassabian) eases the development, because the management of the set of services does not need to be programmed explicitly. For instance, no table needs to be defined to store the far references, and discovered services do not need to be inserted into that table. On the other hand, in our example scenario we want to have a reference to those services, such that we can send other messages later on. Therefore, the return value of the asynchronous message `vote` is either a reference to the service or nil. When a fan is interested in participating in the voting process, a reference to the service is returned. This far reference is stored in a table (`interestedFans`), such that all interested fans can be addressed when necessary. For example, at the end of the application, the final playlist must be sent to all fans who voted.

When we compare the code complexity of the implementation of the SURA applications in NOW (see Section 5.5.2) and in AMBIENTTALK, we can conclude that the implementation in NOW is more compact (16 compared to 52 lines of code). The same conclusions can be made as we did for the comparison of the implementations for the iMPASSE application in Section 7.1. With respect to group orchestration, we claim that the NOW implementation is shorter. Moreover, the programmer does not explicitly need to take care of storing references to the services the group consists of.

7.3 Application Stressing Failure Handling

The third example application (called SWOOP) we show, focuses on the detection and handling of failures. We show that, although AMBIENTTALK has built-in support for both permanent and transient failures, the implementation of such an application using failures is not straightforward. The implementation we present here introduces an abstraction that implements compensations for failures that can occur during a service invocation. The function `invokeService` is shown in Listing 7.3.

Listing 7.3: Invoking a service in AMBIENTTALK.

```
1  def invokeService(service, msg, compensation := `default) {
2      when: service <+ msg becomes: { |res|
3          res;
4      } catch: Exception using: { |exception|
5          if: ((is: exception taggedAs: TimeoutException) ||
6               (is: exception taggedAs: ServiceException)) then: {
7              if: (compensation == `default) then: {
8                  invokeService(service, msg, compensation);
9              } else: {
10                 compensation();
11             };
12         };
13     };
14 };
```

The `invokeService` function has three formal parameters, namely the service that must be invoked (`service`), the asynchronous message that must be sent (`msg`), and an optional compensating action that must be executed (`compensation`). The third argument is by default the symbol `` `default ``, ensuring that the default compensation is executed in case of a failure.

A `when: becomes:` event handler is registered to await the result of the asynchronous message send (line 2). When the result is obtained, the result is returned (line 3).

The code on lines 4-12 is used to ensure that compensating actions are executed in case of a service exception or timeout failure. When no specific compensation is required (i.e., the function is called with only two arguments), the default compensation "retry invoking the same service" is executed, as we can see on line 8. Otherwise, the specified compensating action is executed, as can be seen on line 10.

As we can see parts of the implementation shown in Listing 7.3 some lines of code are highlighted, either in grey or in orange. We use the following colouring scheme: grey is used to highlight the lines of code that are concerned with service orchestration, orange is used to highlight the lines of code that are concerned with failure handling. In the following code snippet we also introduce cyan to colour the lines of code concerned with group orchestration. After we have presented the entire implementation of the application in AMBIENTTALK, we show a stacked graph comparing the NOW and AMBIENTTALK implementation, using this colouring scheme.

The implementation of the SWOOP application is given in Listing 7.4. As we can see on line 1, we register a `whenever: discovered:` event handler to

discover all services that are tagged with the type tag `Assistant`. For each assistant, the location of his/her workshop is retrieved by facility management (line 2). Afterwards the assistant is reminded about the workshop (line 4), and the administration desk looks for a volunteer that can guide the students to that location (line 7).

When no student volunteer can be found, the workshop's assistant is notified that he/she needs to guide the students to the correct location (line 9). Otherwise, the student volunteer is informed of the workshop location that he/she needs to accompany the students to (line 11).

In parallel, the students of the workshop are notified of the workshop's location. By registering a `whenever: discovered:` event handler (on line 16) all services that are exported with the type tag `Student` are captured. However, we only want to address those student that are registered for the particular assistant's workshop. Therefore, the result of the asynchronous message send `getWorkshop` needs to equal the given workshop (line 18). The students of that workshop are collected in a table `students` (on line 14). This information is required, because only when all students have received the message, the group's execution is finished. Therefore, the number of results (i.e., acknowledgements) that are received are also stored (line 15), and an `if` test verifies whether all acknowledgements have been received (line 23).

Listing 7.4: Implementation of the SWOOP application in AMBIENTTALK.

```
1  def observer := whenever: Assistant discovered: { |assistant|
2    when: invokeService(fMgmt, <-getRoom(assistant)) becomes: { |resT|
3      def [room, workshop] := resT;
4      invokeService(assistant, <-show("Workshop", room)@Due(seconds(120)));
5      def cmp := { when: seconds(5) elapsed: { /* restart */ } };
6      def msg := <-getVolunteer(workshop)@Due(seconds(120));
7      when: invokeService(admin, msg, cmp) becomes: { |volunteer|
8        if: volunteer == false then: {
9          invokeService(assistant, <-show("Guidance", room));
10       } else: {
11         invokeService(volunteer, <-show("Guidance", room, workshop));
12       };
13       def announced := false;
14       def students := [];
15       def results := 0;
16       def observer := whenever: Student discovered: { |student|
17         when: student<-getWorkshop() becomes: { |w|
18           if: w == workshop then: {
19             students := students + [student];
20             msg := <-show("WS room", room);
21             when: invokeService(student, msg) becomes: { |res|
22               results := results + 1;
23               if: results == students.length() then: {
24                 `done;
25               };
26             };
27           };
28         };
29         when: student disconnected: {
30           if: ! announced then: {
31             announced := true;
32             invokeService(admin, <-announce(workshop, room));
33           };
34         };
35       };
36     };
37   };
38   when: assistant disconnected: {
39     observer.cancel();
40     invokeService(admin, <-contact(assistant));
41   };
42 };
```

Besides the invocation of services, which we just described, extra lines of code are required to handle failures. Lines 29-34 implement the compensating action that must be executed in case a disconnection occurs during communication with a student. The first time such a disconnection occurs, the administration desk makes an announcement in order to inform all students of the workshop's location. This compensation is only made once for each workshop, namely the first time a disconnection occurs.

Moreover, on lines 38-41 the disconnection of an assistant is taken care of. When an assistant disconnects, the administration desk is asked to inform that assistant through other communication channels, such as for instance email.

Comparing the code complexity of the implementations of the SWOOP application in NOW and AMBIENTTALK follows the observations we made for the previous applications. First, the size of the implementation is smaller in NOW than in AMBIENTTALK (21, compared to 56 lines of code). Secondly, we observe that event handlers are scattered throughout the implementation. Therefore, the same conclusions can be drawn as the ones we described in Section 7.1. Since AMBIENTTALK has no support for default compensating actions in the communication with mobile services, more code (event handlers) is needed for the detection and handling of failures. Although we only show a small part of this example nomadic application, the nesting of the event handlers is 4 levels deep (without the event handlers of the service invocation) which makes it more difficult to follow the control flow of the application.

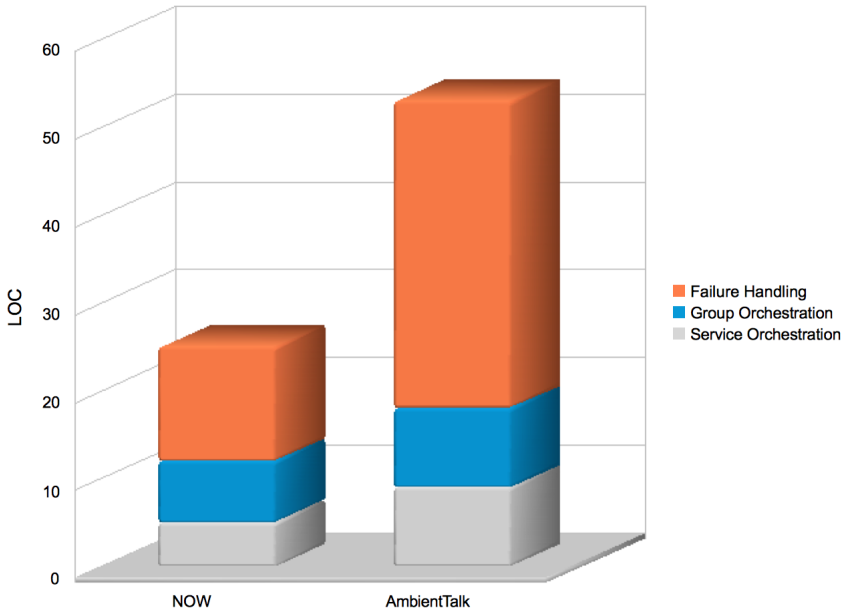


Figure 7.1: Lines of code for the code complexity of the SWOOP application in NOW and AMBIENTTALK.

In Figure 7.1 we compare the implementation of the SWOOP application in NOW and AMBIENTTALK and show the lines of code for different concerns. We use the same colouring scheme as we used to highlight the lines of code in the implementation. Note that we do not show the entire lines of code, since we only focus on the lines that are concerned with either service orchestration, group orchestration, or failure handling. For instance, we omitted the lines where the service is actually invoked.

As we can see, NOW uses less lines of code to describe the orchestration of services. By providing patterns, the control flow of an application can be easily grasped. In AMBIENTTALK, this must be achieved by nesting event handlers in the appropriate order. For example, for a simple example where three services must be invoked sequentially, three `when: becomes:` event handlers must be nested. And, in case the services must be discovered, three additional `when: discovered:` event handlers are required. In NOW, on the other hand, one single line of code is required, namely `Sequence(...)`.

Comparing the lines of code that are concerned with the orchestration of a group of services, both implementations behave similarly. This can be explained, because AMBIENTTALK already provides language constructs for group orchestrations. For example, the `whenever: discovered:` event handler enables the discovery of several services of the same type. The language also provides ambient references to designate a group of nearby services.

Because AMBIENTTALK has no built-in support to automatically recover from failures, a lot more failure handling code must be explicitly written by the programmer. Therefore, the number of lines of code that are concerned with failure handling is higher than the one for NOW.

From the above experiments, we conclude that the implementation of this example application is shorter in our nomadic workflow language compared to a direct style implementation in AMBIENTTALK. Not only is the implementation in NOW more compact, it is also less nested. In order to confirm/validate this claim, however, further user tests with a representative group of people need to be conducted.

7.4 Scalability Results

In this section we present the performance measurements of NOW compared to AMBIENTTALK. Part of the results of the experiments we present here are published in [PVJ13].

We implemented NOW as a library for AMBIENTTALK, and illustrated in the previous section that the introduction of workflow abstractions makes the code more compact and less nested. When absolute performance is a necessity, NOW could be implemented in the AMBIENTTALK interpreter itself. At this moment, the scalability of our language is our main concern. In this section we show that the NOW library is scalable and also measure the overhead introduced by the workflow patterns.

We show the results of three different experiments. In the first experiment, presented in Section 7.4.1, we measure the overhead of the control flow patterns introduced by NOW. For this experiment we implement two basic control flow patterns frequently used in workflows, namely a sequence and parallel split, and compare the execution time of the implementation in NOW and AMBIENTTALK. In Section 7.4.2 we show the overhead introduced by failure detection in both NOW and AMBIENTTALK and compare the execution times of both implementations using the same two control flow patterns. As a third experiment, we compare the performance of implementations of the three applications we presented in Section 2.3.1. Those applications use more control flow patterns, and in order to make the example more complex we wrap the application with a multiple instances pattern.

The experiments are executed on a MacBook with a 2.4 GHZ Intel Core 2 Duo processor and 4GB of 1066MHz DDR3 RAM. The used software includes OS X 10.6.6, JVM 1.6.0_22 and AMBIENTTALK 2.19.1. For each experiment two virtual machines are started (with a maximum heap size of 2GB). A first virtual machine is used for the execution of the example application (in our experiments the sequence or parallel split pattern), whereas the services are executed by a second VM. Each experiment was executed 10 times and the average execution time was computed. Note that all activities used in the first experiments presented in Section 7.4.1 and Section 7.4.2 have the same execution time.

7.4.1 Language Scalability

We conducted two experiments: one where we increased the number of activities of a sequence and one where we augmented the number of branches of a parallel split pattern. We compared the execution time of these example applications in plain AMBIENTTALK and in NOW. The implementation in AMBIENTTALK behaves in the same way as the way the control flow patterns are defined. However, the implementation of those patterns in AMBIENTTALK uses a more direct style of programming using event handlers and hence does not have as many layers of

abstraction as the implementation in NOW.

The first experiment we performed consists of increasing the number of activities in a sequence pattern and comparing the execution time of its implementation using plain AMBIENTTALK and using NOW. We measured the execution time of a sequence with 1, 10, 20, 30, 40 and 80 activities. The results of this experiment are shown in Figure 7.2(a).

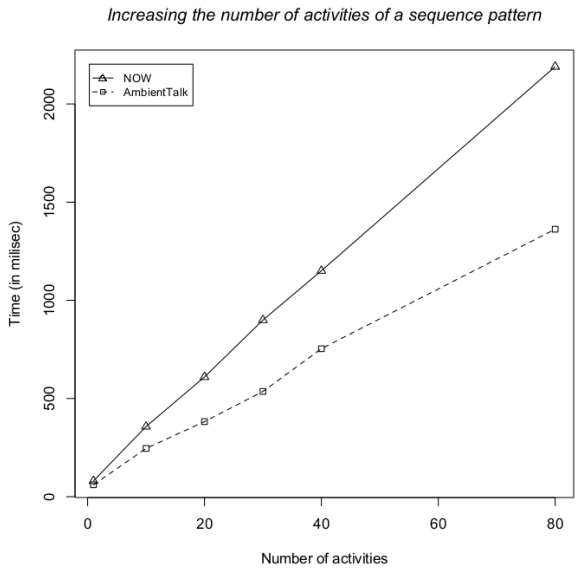
We can conclude that for each of the implementations there exists a linear correlation with the following coefficients:

	NOW	AMBIENTTALK
R^2	0.9993	0.9971
a	26.534	16.3902
b	80.700	62.2802

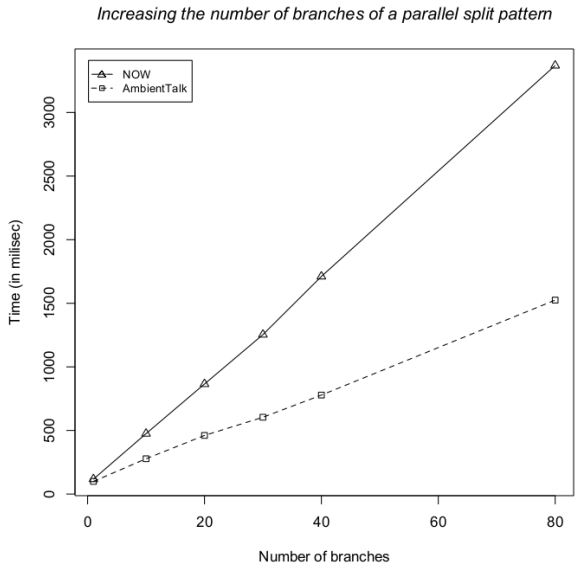
where R^2 is the squared correlation coefficient, and the linear model is defined as ax + b.

The linear correlation between the number of activities and the implementation in our workflow language shows that the implementation of NOW is scalable, as is the case for AMBIENTTALK as well. From the results we can also conclude that there is a small overhead when the NOW application is executed compared to the core AMBIENTTALK application. For instance, the execution time of a sequence of 80 activities is 2.2 seconds in our workflow language, whereas the execution takes 1.4 seconds when the implementation written in (plain) AMBIENTTALK is used. This overhead (an average factor of 1.62) is a result of, amongst other things, the management of the environment implementing the data flow through the workflow. The corresponding implementation in AMBIENTTALK uses local variables in event handlers which are assigned when a return value of a service is retrieved. Moreover, NOW introduces patterns and activities which are all implemented as AMBIENTTALK objects. So, the execution in our workflow languages uses more objects (for each activity, for the pattern itself, environment, ...), and there is a higher number of futures, used to chain patterns and activities together, that must be managed. We refer to Section 7.4.4 for a more elaborate discussion on the overhead introduced by NOW.

As a second experiment, we increase the number of branches of a parallel split pattern (where each branch consists of one activity which has identical execution time in all the branches). The measured application consists of a parallel split of 1, 10, 20, 30, 40, and 80 branches, followed by a synchronisation pattern which merges all branches of the parallel split pattern. The different execution times of its implementation in NOW and AMBIENTTALK are shown in Figure 7.2(b).



(a) Execution time of a sequence pattern.



(b) Execution time of a parallel split pattern.

Figure 7.2: Measurement of overhead introduced by patterns. Results for sequence pattern (a) and parallel split pattern (b).

Again, there is a linear correlation between the execution time and the number of branches. The table below shows the coefficients for each implementation.

	NOW	AMBIENTTALK
R^2	0.9995	0.9986
a	41.2915	17.854
b	52.9908	85.649

Executing a parallel split with 40 branches takes around 1.7 seconds using NOW's implementation, compared to 0.7 seconds when using AMBIENTTALK. The implementation in NOW is on average 2.32 times slower than the one in AMBIENTTALK. The overhead introduced by this workflow pattern is a result from the usage of more futures used to chain patterns and activities together. Moreover, no optimisations for NOW have been implemented yet. We discuss this difference in execution time between NOW and AMBIENTTALK more in Section 7.4.4.

When we compare the coefficients of the regression test $(ax + b)$ of the parallel split, with those of the sequence pattern obtained from the previous experiment, we see that the execution time of the sequence pattern is faster for the same number of activities. As we already mentioned, each experiment uses a separate virtual machine for the execution of the services of the application. For the sequence experiment, only one service is provided and this service is invoked sequentially as many times as the number of the pattern's activities. In order to perform the second experiment, we need several services (one for each branch of the parallel split pattern) in order to prevent a bottleneck in one single service. When a parallel split is being executed, all those services are invoked and can execute in parallel. However, the workflow, running in a separate virtual machine, cannot benefit from any parallelism because it is implemented as a single actor and the actor model specifically forbids inter-actor concurrency. Hence, in this experiment (where the execution time of a single activity is small) there is no real benefit of parallelism. When the execution time of the activities would be significantly larger, using a parallel split would be beneficial and the difference between the execution time of the two experiments would be the other way around.

The benchmarks of these two basic control flow patterns hint that our language implementation is scalable, as there is a linear correlation between the number of activities (branches) and the execution time of the application. In Section 7.4.3 we show the benchmarks of two more elaborate experiments that contain more patterns and use more advanced ones, such as multiple instances. The experiments we conducted for these two basic control flow patterns, sequence and parallel

split, are executed using two VMs on the same machine, which gives us very small latencies. However, in real-life there will be much more latency involved as services are running on different machines connected by a network. The overhead that is introduced by NOW will be less significant in these real-life applications since the network delays are larger.

7.4.2 Scalability of Language with Failure Detection

The experiments we conducted to test the scalability of the language implementation are repeated in order to test the scalability of languages with support for failure detection. In order to conduct these experiments we used the implementation of NOW with failure detection. This simulates the behaviour where each activity is wrapped by a failure pattern that can detect all 4 failures: timeouts, disconnections, exceptions and when a service is not found. In order to achieve the same behaviour when testing the implementation in AMBIENTTALK, we also need to detect those four failures. Instead of only using the `when: discovered:` and `when: becomes:` event handlers in AMBIENTTALK, we also install the event handlers that can catch exceptions (`catch:`), timeouts (by annotating the asynchronous message send with `@Due` and catching a `Timeout` exception), disconnections (`when: disconnected:`), and detect when a service is not found (`when: elapsed:`).

We repeat the experiment where the number of activities, each wrapped by a failure pattern, in a sequence pattern is increased. The results of this experiment are shown in Figure 7.3(a).

We can derive that for each of the performance graphs of implementations with failure detection code there exists a linear correlation between the number of activities and the total runtime with the following coefficients:

	NOW	AMBIENTTALK
R^2	0.9999	0.9973
a	52.268	26.7466
b	90.639	41.8289

where R^2 is the squared correlation coefficient, and the linear model is defined as ax + b.

From the results obtained by this experiment, we can deduce that the implementation of a sequence pattern in NOW is slower with a factor of 1.95 than the corresponding code in AMBIENTTALK. For a discussion on the difference between the execution times of these implementations we refer to Section 7.4.4. Moreover, we can conclude that the execution time of an implementation is increased when

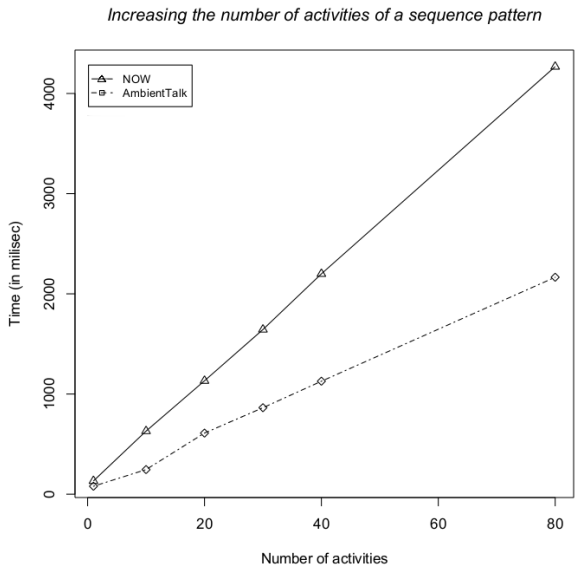
introducing failure detection. This phenomenon occurs for applications written in AMBIENTTALK as well as in NOW. For instance, where previously a sequence of 80 activities had an execution time of 1.4 seconds in AMBIENTTALK, it now has a duration of 2.2 seconds. This factor is 18% higher than the difference in performance measured for the same experiment without failure detection (1.92 compared to 1.62 previously). Recall that for this experiment, each activity is wrapped by a failure pattern that can detect the four types of failures we support. For the corresponding implementation in AMBIENTTALK, this behaviour is achieved by adding four event handlers, for each service that must be discovered and invoked. These extra event handlers cause the difference in execution time with the previous experiment presented in Section 7.4.1.

We also measured the execution times of the second experiment where the number of branches of a parallel split is increased. The different execution times of those implementations are shown in Figure 7.3(b). The table below shows the coefficients of the linear correlation for the implementations both in NOW and AMBIENTTALK.

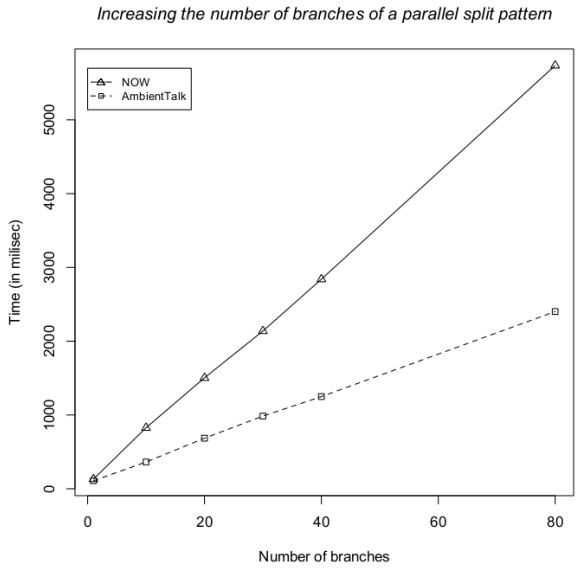
	NOW	AMBIENTTALK
R^2	0.9994	0.9995
a	70.4633	29.0120
b	70.0916	90.5204

Here again we notice that the implementations with failure detection support are slower than the ones without. We can also derive that the execution time of a parallel split in NOW is on average 2.43 times larger than the execution time of the corresponding implementation in AMBIENTTALK. When comparing this factor with the one obtained from the parallel split-experiment without failure detection (presented in Section 7.4.1), we see that this factor is 4.7% higher (the factor of the previous experiment is 2.32).

We showed that the introduction of failure detection increases the execution time (with a factor of 20% for a parallel split). However, we did not measure the overhead that is introduced by executing the actual compensating actions themselves for a certain failure. Executing performance tests for this experiment is meaningless, as the nature of a fault (timeout, disconnection, ...) leads to delays orders of magnitudes larger than its compensation. For instance, the compensation of a timeout of 20 seconds can take a mere 20 ms. We can conclude from these benchmarks that our language implementation with failure detection is scalable for basic control flow patterns (sequence, parallel split), as there is a linear correlation between the number of activities (wrapped by a failure pattern) and the execution



(a) Execution time of a sequence pattern.



(b) Execution time of a parallel split pattern.

Figure 7.3: Measurement of overhead introduced by failure detection. Results for sequence pattern (a) and parallel split pattern (b).

time of the application. In Section 7.4.3 we present benchmarks for applications using more, and more complex, workflow patterns.

7.4.3 Scalability of Example Scenarios

In this section we present benchmarks testing the overhead introduced by NOW measured using the three example applications, of which we presented the implementation in Section 2.3.1.

The iMPASSE Application

The airport example consists of 12 control flow patterns (sequences, parallel splits, synchronizations, connections, exclusive choice patterns, and a structured discriminator). We also want to measure the overhead of more complex control flow patterns, hence, we wrap this workflow with a multiple instances pattern (called *multiple instances with a priori design-time knowledge*). We repeat the experiment by increasing the number of multiple instances. The results are shown in Figure 7.4.

For this experiment, we can conclude that for the implementations in NOW and AMBIENTTALK there exists a linear correlation with the following coefficients:

	NOW	AMBIENTTALK
R^2	0.9999	0.9973
a	52.268	26.7466
b	90.639	41.8289

We can derive that for the airport application the implementation in NOW is 2.12 times slower than the one in AMBIENTTALK.

In this experiment we show that for more complex applications, using several control flow patterns (even a more complex multiple instances pattern), the implementation of NOW is scalable. We can also conclude that our workflow patterns introduce overhead, but still lead to scalable applications.

The SURA Application

The results of the aforementioned experiments have been published in [PVJ13]. The experiments we present now are executed on a MacBook Pro with a 2.7 GHZ Intel Core 2 Duo processor and 16GB of 1600MHz DDR3 RAM. The used software includes OS X 10.8.1, JVM 1.6.0_22 and AMBIENTTALK 2.21. Just like before, for each experiment two virtual machines are started (with a maximum

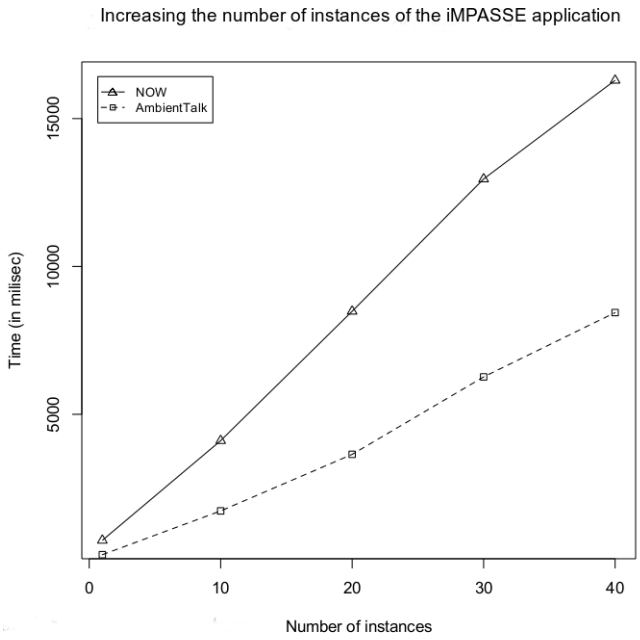


Figure 7.4: Measuring the addition of workflow patterns compared to plain AMBIENTTALK. Results are shown for a multiple instances pattern wrapping the workflow implementing the iMPASSE application.

heap size of 2GB), and each experiment was executed 10 times and the average execution time was computed.

The SURA application only has 6 patterns (group, sequence, filter, synchronised task, cancelling barrier, and group join). We also wrap this workflow with a multiple instances pattern and repeat the experiment with increasing the number of instances. The results of this experiment are shown in Figure 7.5.

For this experiment, we can conclude that for the implementations in NOW and AMBIENTTALK there exists a linear correlation with the following coefficients:

	NOW	AMBIENTTALK
R^2	0.9917	0.9995
a	48.43	42.0588
b	358.94	51.2897

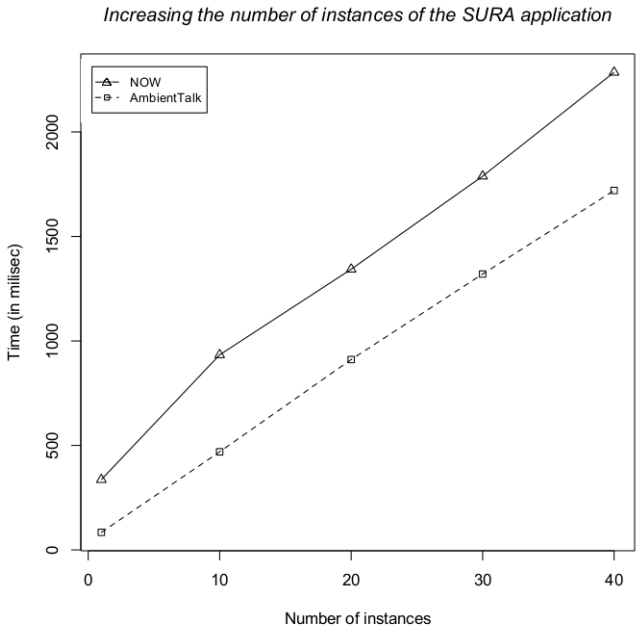


Figure 7.5: Measuring the addition of workflow patterns compared to plain AMBIENTTALK. Results are shown for a multiple instances pattern wrapping the workflow implementing the SURA application.

The SWOOP Application

The third application uses 10 patterns (sequences, parallel split, exclusive choice, groups, failures), and extra patterns for the handling of failures (for instance, alternative). We also wrap this workflow with a multiple instances pattern and repeat the experiment with increasing the number of instances. The results of this experiment are shown in Figure 7.6.

For this experiment, we can conclude that for the implementations in NOW and AMBIENTTALK there exists a linear correlation with the following coefficients:

	NOW	AMBIENTTALK
R^2	0.9984	0.9999
a	412.799	176.439
b	-119.275	68.014

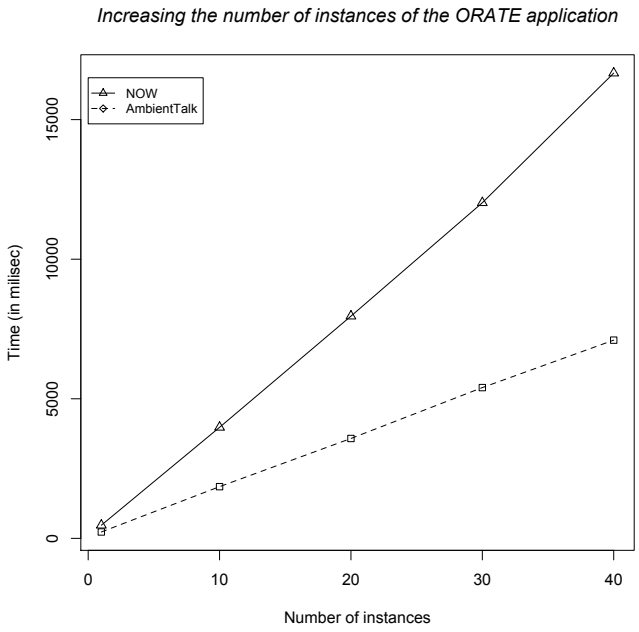


Figure 7.6: Measuring the addition of workflow patterns compared to plain AMBIENTTALK. Results are shown for a multiple instances pattern wrapping the workflow implementing the SWOOP application.

7.4.4 Discussion

In this section, we first discuss the difference in execution time between an application in our nomadic workflow language and its corresponding implementation in AMBIENTTALK. Thereafter, we describe the threats to validity of our scalability results.

The three experiments we presented each show that NOW's implementation is slower than the one in plain AMBIENTTALK. First of all, we want to stress that NOW is a proof-of-concept implementation that is implemented as a library for AMBIENTTALK. The goal of this proof-of-concept implementation is to show that languages can benefit from the extra abstraction layer of workflow patterns, such that the code becomes shorter and less nested. When the language has reached full maturity, it could be integrated in the AMBIENTTALK interpreter itself where more optimisations can be made. The goal of the conducted experiments is to show that the library is scalable.

The difference in execution time between the implementations in NOW and AMBIENTTALK can be explained, as our workflow language introduces more

futures and objects. For instance, instead of just assigning local variables (done in the AMBIENTTALK implementation of the experiments), the implementation in NOW uses an environment object which requires extra provisions in order to allow the data to flow through the entire workflow. Moreover, the workflow language introduces more objects (for each activity, pattern, etc.), which each create a new future when started. Hence, more futures must be managed by NOW. This extra complexity causes the extra overhead we showed in our experiments.

The benchmarks measuring the execution time of two basic control flow patterns without failure detection (in Section 7.4.1) show that the implementation in NOW is slower than their corresponding implementation in AMBIENTTALK. To determine whether our results are generalisable to other workflow patterns, more experiments are needed. We need to extend our experiments to other kinds of applications as well.

As a third experiment we presented the results of three applications that consist of several workflow patterns. From these experiments we could deduce that the implementation of our workflow language is scalable. However, more stress tests on complex applications are needed to further support this conclusion.

7.5 Conclusion

We have shown how NOW can be used by implementing three small, but representative applications for nomadic networks in Chapter 5. In this chapter, we have compared the code complexity of the implementations of these applications in NOW and in AMBIENTTALK. We can conclude that the implementation in our nomadic workflow language is, not only, shorter, its code is less nested. Afterwards, we presented the results of the benchmarks that measure the overhead introduced by the patterns on top of AMBIENTTALK. Although the introduction of workflow patterns introduces an overhead, the implementation of our nomadic workflow language is scalable for applications that use several control flow patterns, of which some of them are rather complex.

8

CONCLUSION

In this chapter, the conclusions of this dissertation are presented. First, the ideas and work presented in this dissertation are summarised stressing our contributions. Subsequently, we present directions for future research.

8.1 Summary and Contributions

The goal of this dissertation was to investigate how the development of nomadic applications can be facilitated by proposing a programming model that attends to the characteristics of these applications.

Nomadic applications are a type of distributed application that run in nomadic networks. Developing nomadic applications is achieved by orchestrating the plethora of services in the network. In stable networks, the orchestration of services is often achieved using technologies such as service-oriented computing [PG03]. The composition and interaction with services is typically specified using so-called workflow languages. Workflow languages do not only allow the composition of services, these languages are also suited to describe the control flow of an application. However, existing workflow languages are not fully equipped for the development of nomadic applications.

As a result, the work presented in this dissertation combines two existing paradigms, from different niches of computer science, namely the workflow paradigm and the ambient-oriented programming paradigm. The first pillar our research is built upon, namely the workflow paradigm, focusses on the orchestration of services. The other pillar supporting our research is the ambient-oriented programming paradigm that takes the hardware characteristics of mobile networks into account, making it suitable to program peer-to-peer mobile applications. In this summary the same sequence of steps is followed as in this dissertation: We first summarise the different criteria necessary for the orchestration of services in a nomadic network, and then design a programming model that adheres to these criteria. Finally, we describe a proof-of-concept implementation of this proposed programming model. In each of these steps, we indicate the contributions of this dissertation.

Nomadic applications are typically deployed and executed on the fixed infrastructure and orchestrate the abundance of services that are available in the network. Because nomadic networks are liable to volatile connections, special characteristics have to be considered. Not all services are necessarily known a priori, services can disconnect at any moment in time, and dynamic groups of services can fluctuate over time. The characteristics of nomadic networks impose several criteria the programming model must adhere to. These criteria can be classified into three categories, namely service orchestration, group orchestration, and failure handling.

The category of service orchestration consists of criteria that allow the invocation of services that are not necessarily known a priori, and can become (temporarily) disconnected. The orchestration of these services must be described in such a way

that the control flow of a nomadic application is separated from the fine-grained application logic, which is implemented by the services. The second category of criteria is concerned with the orchestration of a dynamically changing group of services. We postulate criteria that allow an intensional definition of services a group constitutes of, and allow the number of group members to fluctuate during execution. Moreover, the execution during the orchestration of such a group of services must be controllable, in order to allow processes to be synchronised, redirected, or aborted. The last category consists of criteria that are focussed on the detection and handling of (network) failures. The criteria we put forward have an effect on both service orchestration, and group orchestration. Nomadic networks are subject to intermittent connections, causing network failures to be considered the rule, rather than the exception. Besides the support for automatic recovery of failures, this behaviour can be overridden with application-specific compensating actions by application developers. This results in our first contribution:

Contribution 1: Based upon the characteristics of programming languages implementing either the workflow paradigm or the ambient-oriented programming paradigm, we postulate criteria for orchestration in nomadic networks [PVJ10, PVVJ12].

In order to support service orchestration in a nomadic network, we define high-level abstractions as workflow patterns. Therefore, existing control flow patterns [RtHvdAM06] need to be revised in the context of nomadic networks. We introduce a different interpretation of an activity, which is a placeholder for a service invocation, incorporating the fact that such a service is not necessarily known beforehand, and can become (temporarily) unavailable. We also present a data flow mechanism that can be used in these networks. We set forth a categorisation for these patterns based on the way the control flow patterns can be composed and how they deal with data. This leads to our second contribution:

Contribution 2: We revise traditional workflow languages, reintroduce the concept of an activity, and introduce a data flow mechanism to arrive at a programming model to support service orchestration in a nomadic network [PCJ+10, PVJ10].

In order to allow group orchestration, the programming model must provide novel patterns. First, we show the usage of intensional definitions to describe the members of the group. Such intensional descriptions can be achieved by either providing a service type, or a logical expression the service(s) must fulfil. We

propose a pattern that allows a process to be executed for these group members and present a pattern to restrict the members during the execution of that process. We also introduce novel patterns that allow the synchronisation of the execution of the process, in a way that exceeds the execution of an individual group member. This results in our third contribution:

Contribution 3: We define novel workflow patterns that allow the orchestration of a dynamically changing group of services in a nomadic network [PVVJ12].

Nomadic networks are dominated by volatile connections which force the application developer to take into account failures that can occur during communication with services. Because these failures, such as the disconnection of a service, must be considered the rule rather than the exception, we incorporate a default failure handling mechanism in our proposed programming model. This way, our model can recover from four types of failures, namely the disconnection and unavailability of a service, a timeout caused during communication with a service, and an error caused by the execution of the service. This default recovery mechanism can however be overridden with application-specific compensating actions. Therefore, we put forward a failure pattern that wraps a sub workflow and imposes a cascade of compensating actions for specific failure events. Possible compensating actions include restarting the execution of that sub workflow, retrying the service invocation that failed, and skipping the activity that caused the failure. The proposed failure handling mechanism can be utilised both in the context of service orchestration and group orchestration. This results in the fourth contribution of this dissertation:

Contribution 4: We define novel workflow patterns that allow the specification of compensating actions for failures that occur during service orchestration and group orchestration [PVJ10, PVVJ12].

We designed and implemented these novel patterns in the context of NOW, a novel workflow language for nomadic networks. We first present this novel workflow language from the perspective of the application programmer: We introduce the language's syntax and show how nomadic applications can be developed using NOW. Thereafter, we describe the implementation of the nomadic workflow language. NOW is implemented as an extra layer of abstraction on top of the ambient-oriented programming language AMBIENTTALK/2. We explain how the workflow patterns are implemented, and describe how the language can be extended with novel patterns. This results in the fifth and sixth contributions:

Contribution 5: We introduce a novel workflow language, called NOW, specifically sculpted for orchestration in nomadic networks [PVJ10].

We compared the code complexity of the implementations of three nomadic applications in the developed workflow language NOW and AMBIENTTALK/2, a state-of-the-art programming language for developing mobile applications. We also present an evaluation of the nomadic workflow language's performance and scalability. This results in the final seventh contribution:

Contribution 6: We present a validation of our approach by comparing the code complexity of NOW to a state-of-the-art programming language for mobile networks, namely AMBIENTTALK/2. Finally, we perform an initial quantitative evaluation of NOW's performance and scalability [PVJ13].

8.2 Discussion and Future Work

The assessment of NOW we presented in Chapter 7 leads us to think that the programming model we proposed is well suited for the development of nomadic applications. However, a number of issues need to be further explored. In this section we reflect upon the concepts this dissertation introduces, and present new research areas that are related to our work, but are outside the scope of this dissertation.

Security and Privacy of Nomadic Applications The execution of nomadic applications relies heavily on the information that is available about connected services. In order to orchestrate (dynamic groups of) services, these services must advertise themselves on the network. Moreover, each service publishes facts on the network that are exchanged with nearby servers, and possibly even other services. The resulting uncontrolled dissemination of information could pose a potential privacy risk. In this dissertation we did not consider both security and privacy. When security and privacy are concerns, our proposed programming model needs to incorporate additional requirements such as confidentiality, integrity, anonymity, etc. In order to add these requirements we can look at multi-level secure workflows [AHB00] where data and tasks are associated with a security level. We can also find inspiration in the world of web services where a policy language is used to represent the capabilities and requirements of web services as policies [DLGHB+05].

Formalisation In this dissertation we postulated criteria a programming model must adhere to in order to facilitate the development of nomadic applications. We presented the list of workflow patterns this programming model consists of, and we introduced a proof-of-concept workflow language implementing this model. These research artifacts and the nomadic applications we presented give us an idea of how our ideas and abstractions behave, and to which applications they can be applied. However, we have not formalised the semantics of the novel workflow patterns we introduced. In the workflow community, there is a tradition of formalising novel workflow concepts. Existing workflow languages are often built upon calculi, process algebras, petri nets, event-driven process chains or finite state automata. Introducing a formalisation could serve as a formal backing for the work introduced in this dissertation. Therefore, we could use the ambient calculus [CG98], a process calculus that can describe the movement of processes and devices and that can express the communication between (mobile) processes. Another possibility could be to use the operational semantics of the AmbientTalk programming language [VSHD12].

Executing More Complex Behaviour on Mobile Devices In NOW, the control flow of a nomadic application is specified and controlled by the fixed infrastructure, whereas the fine-grained application logic is provided by services in the network. These services are applications that are running on connected servers, or on colocated mobile devices. In this dissertation we rely on the nomadic network's fixed infrastructure that cannot become unavailable during the execution of an application. A NOW program consists of two main parts: the workflow description, and the implementation of the connected services. These services reside on devices that are either connected via a reliable or an unreliable communication link. In this dissertation, services implement the fine-grained application logic, whereas the control flow of the application is specified and controlled by the process running on the backbone. In the future we would like to explore the possibility of letting the servers and mobile devices run more complex applications moving towards a decentralised workflow. We envision mobile devices that can deploy and execute more complex behaviour, ranging from a sub workflow to a workflow describing an entire nomadic application. In the latter case, the mobile devices that are running the nomadic application could be considered the backbone of the network, and must orchestrate services residing on other devices in the network.

Collaboration of Nomadic Applications In this dissertation we made the assumption that the deployment and execution of a nomadic application takes place at

a server, i.e., the fixed infrastructure of the nomadic network. As we already discussed in the previous paragraph, we would like to investigate the possibility to have mobile devices operate as the backbone that is responsible for the execution of a nomadic application. When this is realised, we plan to investigate how such individually defined processes can collaborate. The cooperation of nomadic applications is an interesting research idea, which is crucial in situations such as disaster area where multiple parties are involved. In such a situation, every head of a team (police, fire brigade, etc.) can serve as the "backbone" and execute the nomadic application while interacting with the services. These services are the applications provided by the mobile devices of the team members. In order to support collaboration between the nomadic applications that describe the emergency procedure for each team, dedicated language abstractions are required. First of all, it must be possible to specify the transition or task in the process that is connected to the processes running on another device in the network. Moreover, since nomadic applications are possibly being executed on a mobile device, the description of the process can become unavailable. In order to allow the collaboration of nomadic applications, that are possibly running on mobile devices, our workflow language must be extended with novel language abstractions. Because we can no longer make the assumption that the nomadic application is executed on a fixed infrastructure, NOW's failure handling patterns must be extended. Not only must we investigate novel compensating actions, we must also extend the language such that connections to other processes can be specified. Inspiration for these language abstractions can be found in aspect-oriented programming [Kic96].

Context-Oriented Nomadic Applications In this dissertation, we presented NOW, a novel workflow language implementing the proposed programming model that facilitates the development of nomadic applications. One of the characteristics of the nomadic applications we envision, is the orchestration of a dynamically changing group of services, such as the services running on the devices of all passengers of a specific flight. Because a nomadic network is highly dynamic, an extensional description of the involved services cannot always be realised. Therefore, services a group constitutes of can be defined intensionally. In this dissertation we developed a proof-of-concept implementation of a workflow language that incorporates such intensional definitions of services. Therefore, an integration of the logic coordination language CRIME was required. This coordination language allows applications to specify logic rules which specify how the applications should respond to changes in its immediate environment. Such changes are modelled by the addition or removal of facts which contain context information. Because we are already integrated CRIME in NOW, we can extend our vision of nomadic

applications to nomadic applications that react upon context changes. For instance, as a result of a service invocation, novel facts can be asserted in the fact space. The addition of a new fact can be regarded as an external event which can trigger the execution of a workflow.

Reasoning about the Past and Present The workflow language we developed is integrated with the logic coordination language CRIME. This integration is required to allow an intensional definition of services that must be orchestrated. As we already mentioned in the above paragraph, we would like to extend NOW such that context-dependent nomadic applications can be developed. Additionally, we would like to reason about past context information, because past events may contribute useful information to make decisions about the present situation. This would allow the specification of more sophisticated intensional definitions, such as "the last service I used", "the services that have been used since the last time the tourist guide was away", etc. Because CRIME only allows reasoning about the current context, the language was extended with temporal operators in CRIMETIME [PSHM07].

Nomadic networks are subject to changes since the topology of the network is constantly changing. Therefore, the behaviour of services can evolve over time, although the composition of the services is not altered (i.e., the same workflow description is being used). In order to ensure that this evolution of services does not lead to incorrect behaviour, the functionality and quality of services must be probed during the workflow's execution. The idea of self-adaptive workflows that preserves their dependability and robustness has already been researched. In BPEL, so-called self-supervising processes were introduced such that a BPEL process can assess its behaviour by specifying rules for monitoring and recovery [BG11].

RFID-Enabled Nomadic Applications Nomadic networks do not only consist of a plethora of services, a lot of sensors are also available. The ambient-oriented programming paradigm has already been extended in order to allow the development of RFID-enabled applications [Lom11]. These applications arise when RFID technology is used in a mobile ad hoc networks. A possible direction for future work could be to extend our proposed programming model such that nomadic applications are reactive to the events generated by sensors (such as RFID technology).

BIBLIOGRAPHY

[AAA+96] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, Mohan Kamath, Roger Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 574–581, Washington, DC, USA, 1996. IEEE Computer Society. 106, 154

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004. 14

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 38

[AGK+95] Gustavo Alonso, Roger Günthör, Mohan Kamath, Divyakant Agrawal, Amr El Abbadi, and C. Mohan. Exotica/FMDC: Handling disconnected clients in a workflow management system. In *CoopIS*, pages 99–110, 1995. 153

[AHB00] Vijayalakshmi Atluri, Wei-Kuang Huang, and Elisa Bertino. A semantic-based execution model for multilevel secure workflows. *J. Comput. Secur.*, 8(1):3–41, January 2000. 229

[AKA+94] Gustavo Alonso, Mohan U. Kamath, Divyakant Agrawal, Amr El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Technical report, IBM, November 1994. 154

[AKM08] Farhad Arbab, Natallia Kokash, and Sun Meng. Towards using Reo for compliance-aware business process modeling. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of*

Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, volume 17 of *Communications in Computer and Information Science*, pages 108–123. Springer, 2008. 162, 163

[AM10] Musab AlTurki and José Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In *RTRTS*, pages 26–45, 2010. 161

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14:329–366, June 2004. 162

[BG11] Luciano Baresi and Sam Guinea. Self-supervising bpel processes. *IEEE Trans. Software Eng.*, 37(2):247–263, 2011. 232

[BI93] Andrew P. Black and Mark P. Immel. Encapsulating plurality. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 57–79, London, UK, UK, 1993. Springer-Verlag. 28

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. *SIGPLAN Not.*, 39(10):331–344, October 2004. 46

[BvHH+04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference, february 2004. http://www.w3.org/TR/owl-ref/. 78

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, March 2001. 14

[CCPP99] Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst.*, 24:405–451, September 1999. 107

[CdM+08] Tiziana Catarci, Massimiliano de Leoni, Andrea Marrella, Massimo Mecella, Berardino Salvatore, Guido Vetere, Schahram Dustdar, Lukasz Juszczyk, Atif Manzoor, and Hong-Linh Truong. Pervasive software environments for supporting disaster responses. *IEEE Internet Computing*, 12(1):26–37, jan 2008. 151

[CdRdL+06] Tiziana Catarci, Fabio de Rosa, Massimiliano de Leoni, Massimo Mecella, Michele Angelaccio, Schahram Dustdar, Begofia Gonzalvez, Giuseppe Iiritano, Alenka Krek, Guido Vetere, and Zdenek M. Zalis. WORKPAD: 2-layered peer-to-peer for emergency management through adaptive processes. *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 0:43, 2006. 151

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, FoSSaCS '98, pages 140–155, London, UK, UK, 1998. Springer-Verlag. 230

[CLK01] Dickson K. W. Chiu, Qing Li, and Kamalakar Karlapalem. *ADOME-WFMS: towards cooperative handling of workflow exceptions*, pages 271–288. Springer-Verlag New York, Inc., New York, NY, USA, 2001. 107

[CM06] Brian Chin and Todd Millstein. Responders: language support for interactive applications. In *Proceedings of the 20th European conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 255–278, Berlin, Heidelberg, 2006. Springer-Verlag. 55, 112, 204

[CPM06] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In Paolo Ciancarini and Herbert Wiklicky, editors, *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006. 161

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (JSON). RFC 4627, IETF, 7 2006. 78

[Ded06] Jessie Dedecker. *Ambient-Oriented Programming*. PhD thesis, Vrije Universiteit Brussel, Department of Informatics, 2006. 31, 32

[DLGHB+05] Giovanni Della-Libera, Martin Gudgin, Phillip Hallam-Baker, Maryann Hondo, Hans Granqvist, Chris Kaler, Hiroshi Maruyama, Michael McIntosh, Anthony Nadalin, Nataraj Nagaratnam, Rob Philpott, Hemma Prafullchandra, John Shewchuck, Doug Walter,

and Riaz Zolnofoon. Web services security policy language, july 2005. http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html. 229

[DVM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming. In Ralph E. Johnson and Richard P. Gabriel, editors, *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. 33

[DVM+06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag. 2, 7, 20, 31, 32

[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, June 2003. 157

[EL96] Johann Eder and Walter Liebhart. Workflow recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems*, COOPIS '96, pages 124–134, Washington, DC, USA, 1996. IEEE Computer Society. 108

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982. 80

[Gel85] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985. 33, 156

[GHS95] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distributed Parallel Databases*, 3(2):119–153, apr 1995. 14, 15

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 25

[GVV+09] Elisa Gonzalez Boix, Tom Van Cutsem, Jorge Vallejos, Wolfgang De Meuter, and Theo D'Hondt. A leasing model to deal with

partial failures in mobile ad hoc networks. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 231–251. Springer, 2009. 45

[HA98] Claus Hagen and Gustavo Alonso. Flexible exception handling in the OPERA process support system. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*, ICDCS '98, pages 526–533, Washington, DC, USA, 1998. IEEE Computer Society. 107

[HB04] Hugo Haas and Allen Brown. Web services glossary. w3c working group note, april 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/. 14

[HGR07] Gregory Hackmann, Christopher Gill, and Gruia-Catalin Roman. Extending BPEL for interoperable pervasive computing. In *Proceedings of the 2007 IEEE International Conference on Pervasive Services*, pages 204–213, 2007. 150, 152

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proceedings of the 7th joint conference on Modular Programming Languages*, JMLC'06, pages 4–22, Berlin, Heidelberg, 2006. Springer-Verlag. 55, 204

[Hol95] David Hollingsworth. Workflow management coalition - the workflow reference model. Technical report, Workflow Management Coalition, Jan 1995. 15

[JD08] Lukasz Juszczyk and Schahram Dustdar. A middleware for service-oriented communication in mobile disaster response environments. In *Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '08, pages 37–42, New York, NY, USA, 2008. ACM. 152

[JE+07] Diane Jordan, John Evdemon, et al. Web Services Business Process Execution Language, version 2.0, April 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. 7, 14, 106, 144, 147

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Survey*, 28(4es), December 1996. 231

[KQCM09] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009. 160

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, March 1988. 155

[KR91] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. 47

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. 145

[LEH04] Jun Luo, Patrick Th. Eugster, and Jean-Pierre Hubaux. Pilot: Probabilistic lightweight group communication system for ad hoc networks. *IEEE Transactions on Mobile Computing*, 3(2):164–179, April 2004. 25

[Ley95] Frank Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *BTW*, pages 51–70, 1995. 154

[LN86] Behrouz Tork Ladani and Naser Nematbakhsh. Modelling web service composition using Reo coordination language. In *International Conference for Internet Technology and Secured Transactions (ICITST-2007)*, 1386. 2. 163

[Lom11] Andoni Lombide Carreton. *Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications*. PhD thesis, Vrije Universiteit Brussel, Department of Informatics, 2011. 232

[LtW+11] Marcello La Rosa, Arthur H. M. ter Hofstede, Petia Wohed, Hajo A. Reijers, Jan Mendling, and Wil M. P. van der Aalst. Managing process model complexity via concrete syntax modifications. *IEEE Trans. Industrial Informatics*, 7(2):255–265, 2011. 18

[Lut94] Roland Lutz. IBM FlowMark workflow manager: concept and overview. In *Proceedings of the ninth Austrian-informatics conference on Workflow management : challenges, paradigms and products: challenges, paradigms and products*, CON '94, pages 65–68, Munich, Germany, Germany, 1994. R. Oldenbourg Verlag GmbH. 153

[LWM+11] Marcello La Rosa, Petia Wohed, Jan Mendling, Arthur H. M. ter Hofstede, Hajo A. Reijers, and Wil M. P. van der Aalst. Managing process model complexity via abstract syntax modifications. *IEEE Trans. Industrial Informatics*, 7(4):614–629, 2011. 18

[MA07] Sun Meng and Farhad Arbab. Web services choreography and orchestration in Reo and constraint automata. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 346–353, New York, NY, USA, 2007. ACM. 162

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM. 45

[MAGK95] C. Mohan, Gustavo Alonso, Roger Günthör, and Mohan Kamath. Exotica: A research perspective on workflow management systems. *Data Engineering Bulletin*, 18, 1995. 153

[Man00] Dragos Manolescu. *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2000. 144

[MC07] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, 23 Mar 2007. 161

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile Computing Middleware. In Enrico Gregori, Giuseppe Anastasi, and Stefano Basagni, editors, *Advanced Lectures on Networking, NETWORKING 2002*, volume 2497 of *Lecture Notes in Computer Science*, pages 20–58. Springer-Verlag, 2002. 2, 7, 20

[MK09] Andrew Matsuoka and David Kitchin. A semantics for exception handling in Orc. 2009. 161

[MKC07] Kristi Morton, David Kitchin, and William Cook. Orc-X: Combining Orchestrations and XQuery. Technical Report TR-07-63, The University of Texas at Austin, December 2007. 161

[MLM+06] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Technical report, OASIS, 2006. 14

[MRV98] Amy L. Murphy, Gruia-Catalin Roman, and George Varghese. An exercise in formal reasoning about mobile communications. In *In Proc. of the 9th Int. Workshop on Software Specification and Design*, pages 25–33. IEEE Computer Society Press, 1998. 1, 22

[MSB12] Tom Maguire, David Snelling, and Tim Banks. Web Services Service Group - Specification (WS-ServiceGroup), Version 1.2., August 2012. http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf. 90

[MSP+07] Stijn Mostinckx, Christophe Scholliers, Eline Philips, Charlotte Herzeel, and Wolfgang De Meuter. Fact Spaces: Coordination in the face of disconnection. In Amy L. Murphy and Jan Vitek, editors, *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4467 of *Lecture Notes in Computer Science*, pages 268–285. Springer, 2007. 78, 79

[MTS05] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: programming in E as plan coordination. In *Proceedings of the 1st international conference on Trustworthy global computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag. 38

[MVT+09] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in AmbientTalk. *Software - Practice and Experience*, 39(7):661–699, May 2009. 47

[MWW+98] Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelika Kotz Dittrich, and Gerhard Weikum. From centralized workflow specification to distributed workflow execution, 1998. 144

[MZ04] Marco Mamei and Franco Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 479–486, 2004. 158

[NMM+03] Henrik F. Nielsen, Noah Mendelsohn, Jean J. Moreau, Martin Gudgin, and Marc Hadley. SOAP version 1.2 part 1: Messaging framework. W3C recommendation, W3C, June 2003. 14

[Obj11] Object Management Group. Business Process Model and Notation, version 2.0, January 2011. http://www.omg.org/spec/BPMN/2.0/. xi, 16, 82, 94, 105, 163

[Obj12] Object Management Group. BPMN 2.0 by example, august 2012. http://www.bpmn.org. xi, 15, 16

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998. 155

[PCJ+10] Eline Philips, Andoni Lombide Carreton, Niels Joncheere, Wolfgang De Meuter, and Viviane Jonckers. Orchestrating nomadic mashups using workflows. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 1:1–1:7, New York, NY, USA, 2010. ACM. 227

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003. 14, 25, 149

[PG03] Mike P. Papazoglou and Dimitrios Georgakopoulos. Introduction: Service-oriented computing. *Commun. ACM*, 46(10):24–28, oct 2003. 7, 14, 226

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 368–377, New York, NY, USA, 1999. ACM. 156, 157, 187

[PSHM07] Eline Philips, Christophe Scholliers, Charlotte Herzeel, and Stijn Mostinckx. Reasoning about past events in context-aware middleware. In *Proceedings of Object Technology for Ambient Intelligence and Pervasive Computing*, OT4AmI2007, pages 27–32, Berlin Heidelberg, 2007. Springer-Verlag. 232

[PVJ10] Eline Philips, Ragnhild Van Der Straeten, and Viviane Jonckers. NOW: a workflow language for orchestration in nomadic networks. In *Proceedings of the 12th international conference on Coordination Models and Languages*, COORDINATION'10, pages 31–45, Berlin, Heidelberg, 2010. Springer-Verlag. 9, 227, 228, 229

[PVJ13] Eline Philips, Ragnhild Van Der Straeten, and Viviane Jonckers. NOW: Orchestrating services in a nomadic network using a dedicated workflow language. *Science of Computer Programming*, 78(2):168–194, feb 2013. 9, 212, 220, 229

[PVVJ12] Eline Philips, Jorge Vallejos, Ragnhild Van Der Straeten, and Viviane Jonckers. Group orchestration in a mobile environment. In *Proceedings of the 14th international conference on Coordination Models and Languages*, COORDINATION'12, pages 181–195, Berlin, Heidelberg, 2012. Springer-Verlag. 10, 227, 228

[RS95] Andreas Reuter and Friedemann Schwenkreis. ConTracts - a low-level mechanism for building general-purpose workflow management-systems. *IEEE Data Engineering Bulletin*, 18:4–10, 1995. 106

[RtEv05a] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: identification, representation and tool support. In Lois M. L. Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor, editors, *Proceedings of the 24th international conference on Conceptual Modeling*, Lecture Notes in Computer Science, pages 353–368, Berlin, Heidelberg, 2005. Springer-Verlag. xii, 17, 94, 105, 106

[RtEv05b] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: identification, representation and tool support. In *Proceedings of the 24th international conference on Conceptual Modeling*, ER'05, pages 353–368, Berlin, Heidelberg, 2005. Springer-Verlag. 62, 65

[RtHvdAM06] Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPM Center, 2006. `http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf`. 8, 59, 67, 69, 72, 73, 76, 90, 109, 118, 124, 150, 161, 176, 178, 180, 185, 186, 227

[Rvt06] Nick Russell, Wil van der Aalst, and Arthur ter Hofstede. Workflow exception patterns. In Eric Dubois and Klaus Pohl, editors, *Proceedings of the 18th international conference on Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 288–302, Berlin, Heidelberg, 2006. Springer-Verlag. 18

[RvtE05] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Workflow resource patterns: identification, representation and tool support. In *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 216–232, Berlin, Heidelberg, 2005. Springer-Verlag. 17

[Sen08] Rohan Sen. *Supporting collaboration in mobile environments*. PhD thesis, Washington University, St. Louis, MO, USA, 2008. AAI3332131. 149

[SGD09] Christophe Scholliers, Elisa Gonzalez Boix, and Wolfgang De Meuter. TOTAM: Scoped tuples for the ambient. *ECEASST*, 19, 2009. 159

[SMA+97] Remco V. Stiphout, Theo D. Meijler, Ad Aerts, Dieter Hammer, and Riné L. Comte. TREX: Workflow TRansactions by Means of EXceptions. Technical report, Eindhoven University of Technology, 1997. 106

[SOSF04] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Cameron Foulger. Data flow and validation in workflow modelling. In *Proceedings of the 15th Australasian database conference - Volume 27*, ADC '04, pages 207–214, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc. 62, 66

[SP07] Christophe Scholliers and Eline Philips. Coordination in volatile networks. Master's thesis, Vrije Universiteit Brussel, Department of Informatics, 2007. 81

[SR93] Amit P. Sheth and Marek Rusinkiewicz. On transactional workflows. *IEEE Data Eng. Bull.*, 16(2):37–40, 1993. 106

[SRG08] Rohan Sen, Gruia-Catalin Roman, and Christopher D. Gill. CiAN: A workflow engine for MANETs. In Doug Lea and Gianluigi Zavattaro, editors, *Proceedings of the 10th international conference on*

Coordination models and languages, volume 5052 of *Lecture Notes in Computer Science*, pages 280–295, Berlin, Heidelberg, 2008. Springer-Verlag. 149

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 155

[The99] The Workflow Management Coalition. *Workflow Management Coalition, Terminology & Glossary (Document No. WFMC-TC-1011)*. Workflow Management Coalition Specification, feb 1999. 19

[TWRG09] Louis Thomas, Justin Wilson, Gruia-Catalin Roman, and Christopher Gill. Achieving coordination through dynamic construction of open workflows. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 14:1–14:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc. 151

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. In *Lisp Symbolic Compututation*, volume 4, pages 223–242, Hingham, MA, USA, jul 1991. Kluwer Academic Publishers. 34

[Vas07] Mariana Hernandez Vasquez. User-adaptable context-aware applications in a mobile environment. Master's thesis, Vrije Universiteit Brussel, Écoles des Mines de Nantes, 2007. 78

[VC08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Department of Informatics, 2008. 20, 21, 22, 28, 33, 50, 84

[vdAtH05] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005. 7, 90, 106, 144, 145

[VDM+06] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez Boix, Theo D'Hondt, and Wolfgang De Meuter. Ambient references: addressing objects in mobile networks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 986–997, New York, NY, USA, 2006. ACM. 92

[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 *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society. 55, 92

[VSHD12] Tom Van Cutsem, Christophe Scholliers, Dries Harnie, and Wolfgang De Meuter. An operational semantics of event loop concurrency in AmbientTalk. Technical report, Vrije Universiteit Brussel, 2012. http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-04.pdf. 230

[vtea12] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and et al. The workflow patterns initiative. http://www.workflowpatterns.com, april 2012. 15, 17

[vv02] Wil M. P. van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002. 145, 151

[vvH94] W. M. P. van der Aalst, Kees van Hee, and Geert-Jan Houben. Modelling and analysing workflow using a Petri-net based approach. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proceedings of the 2nd Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, 1994. 145, 147, 149

[Wei91] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, 1991. 1

[Wei93] Mark Weiser. Ubiquitous computing. *Computer*, 26:71–72, 1993. 1

[Wei99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999. 1

[WV98] Mathias Weske and Gottfried Vossen. Workflow languages. *Handbook on Architectures of Information Systems (International Handbooks on Information Systems)*, pages 359–379, 1998. 14

INDEX

SWOOP, 139, 207
SURA, 131, 204
iMPASSE, 122, 204

Abstract Process
    in BPEL, 147
Active Tuple
    in Linda, 156
Activity, 15, 19, 60, 113, 168
Actor, 38
    in WORKPAD, 151
`actor:`, 38
Actor Model, 38
ADOMA-WFMS, 107
`@All`, 51
Alternative, 95, 199
`ambient:`, 50
Ambient Devices, 1
Ambient Reference, 50, 92
Ambient-Oriented Programming, 2
AmbientTalk, 31, 33
AmbientTalk/2, 31
AmOP, 20
`@Any`, 51
Application, 19
Arity Decoupling, 22, 28
Asynchronous Channel
    in Reo, 162
Asynchronous Message, 40
Atomic Task, 15

Barrier, 86
Basic Activity
    in BPEL, 148
Block Closure, 41
BPEL, 147
BPMN, 15
Business Process, 19

Cancelling Barrier, 86, 133
Cancelling Discriminator, 124
Case, 105
Chimera-Exc, 107
Chromatic Orc, 161
CiAN, 149
`clone:`, 35
Closure, 34
Communicating Event Loops
    in E, 38
Compensation, 141
Compensations, 135
`component`, 175
Concurrent Combinator
    in Orc, 160
Connection, 121
`Connection`, 122
Connector
    in Reo, 162

Contract
in ConTracts, 106
ConTracts, 106
Control Connector
in FlowMark, 153
Control Flow, 120
Control Flow Perspective, 17
Coordination Languages, 155
Coordination Model, 155
Crime, 78, 79, 128, 187

Data Connector
in FlowMark, 153
Data Environment, 63
Data Flow, 116, 170
Data Flow Perspective, 17
Decoupled Communication, 26
`deftype`, 43
Delegation, 35
Description, 135
Disconnected Client
in FlowMark, 153
Distributed Naming, 32
Drop, 101
`@Due`, 45
Dynamic Modification, 28

E, 38
`Env`, 113
Exception Handling Perspective, 18
Exclusive Choice, 17, 124, 141
Executable Process
in BPEL, 147
Exotica/FDMC, 153
Explicit Control Flow, 27
`export: as:`, 42
`extend: with:`, 35

Fact Space Model, 79, 128
Failure, 139, 195
Failure Description, 136
Failure Events, 135
Failure Handling, 6, 29, 135, 195, 207
Automatic, 29, 196
Explicit, 29
For Groups, 30
Individual, 30
Far Reference, 39
Fault Handler
in BPEL, 106
Federated Fact Space, 79, 128
Federated Tuple Space
in LIME, 156
Filter, 84, 131, 191
FlowMark, 153
Forward Chainer, 187
Function, 37
Future, 40

Gateway, 17
Generative Communication, 156
Group, 82, 131, 141
in LIME, 157
Group Behaviour, 25
Group Communication, 25
Group Join, 88
Group Membership, 28
Group Orchestration, 5, 26, 28, 127, 187, 204
Group Patterns, 128
Group Synchronisation Patterns, 128, 190, 192

Host
in CiAN, 149
Host Level Tuple Space
in LIME, 156
Host Listener
in CiAN, 150

iMPASSE, 23
Instance, 19
Intensional Definition, 28, 128, 187
Intercession, 45
Interface Tuple Space
    in LIME, 157
Interleaved Routing, 180
Intermittent Connections, 20
Introspection, 45
Inversion of Control, 55
Isolate, 40
`isolate:`, 40

Kepler, 145
Keyword
    in Orc, 160

Leased Object Reference, 45
Lexical Scope, 36
LIME, 156
Linda, 156
Lossy Channel
    in Reo, 162

MANET, 22
`merge`, 173
Merging Strategy, 117
Meta-Object Protocol, 47
Micro-Workflow, 144
Mirage, 48, 49
Mirror, 45, 46
`mirror:`, 48
Mirror Construction Closure, 49
`mirroredBy:`, 49
Mobile Ad Hoc Network, 1, 22
Mobile Channel
    in Reo, 162
Mobile Service, 3
MOP, 47
Multi-Choice, 178
Multifuture, 51

Net Variable
    in YAWL, 145
Nomadic Application, 2
Nomadic Network, 1, 22

`object:`, 34, 40
Object Scope, 36
`@One`, 51
Open Workflow, 151
OPERA, 107
Orc, 160
Orchestration, 13, 26

Parallel Split, 69, 124, 141
Partial Failure, 20
Partner Link
    in BPEL, 148
Passive Tuple
    in Linda, 156
Persistent Trigger, 76, 125, 185
Planned Disconnection
    in FlowMark, 153
Presentation Perspective, 18
`prioritise`, 172
Process, 19
Process Definition, 20
Propagation Rule
    in TOTA, 158
Prototype, 34

Quota Constraint, 85

`random`, 173
Reaction
    in LIME, 157
    in TOTAM, 160
`reflect:`, 46
Reflectee, 46

Reflection, 45
Registered Service, 3
Registry Service
    in Reo, 163
Reo, 162
Replace, 95
RESCUE, 152
Resource Perspective, 17
Restart, 93
RestartAll, 94
`restore`, 173
Retry, 93
Router
    in Reo, 162

Scope Activity
    in BPEL, 148
`self`, 36
Self-send, 36
Sequence, 17, 69, 125, 133, 176
Service, 3, 114
Service Description, 113
Service Exception, 93
Service Group, 90
Service Orchestration, 5, 14, 25, 26, 118, 174, 204
Service Timeout, 92
Service Unresponsive, 92
Shared Fact Space, 82
Site
    in Orc, 160
Skip, 95, 199
SkipAll, 95
Sliver, 150, 152
Slot Object, 47
Slots, 47
Snapshot, 50
Snapshot Group, 84
Space Decoupling, 22, 27
Sphere
    in OPERA, 107
Spheres of Compensation, 154
Standard Patterns, 69, 176
Stationary Service, 3
Structured Activity
    in BPEL, 148
Sub Process, 15, 19
Sub Workflow, 19
`super`, 35
SURA, 24
SWOOP, 24
Synchronisation Decoupling, 22, 27
Synchronisation Patterns, 72, 121, 182
Synchronised Task, 89, 131, 192
Synchronization, 73, 124, 184
Synchronous Channel
    in Reo, 162

Task, 19
Task Variable
    in YAWL, 145
Time Constraint, 85
Time Decoupling, 22, 26
`TimeoutException`, 45
TOTA, 158
TOTAM, 159
Trigger Patterns, 76, 185
Tuple, 66
    in Linda, 156
Tuple Space
    in Linda, 156
Tuple Space Descriptor
    in TOTAM, 159
Type Tag, 43

Ubiquitous Computing, 1
User, 20
User Service, 3

Vat
in E, 38
Volatile Connections, 20

Wait, 95
Wait-and-Resume, 101
Web Service, 14
`whenAll: becomes:`, 51
`when:becomes:`, 41
`when: becomes: catch:`, 198
`when: disconnected:`, 198
`when: discovered:`, 43, 198
`whenEach: becomes:`, 51
`when: elapsed:`, 198
`whenever: disconnected:`, 44
`whenever: discovered:`, 43
`whenever: reconnected:`, 44
Work Item, 105
in YAWL, 146
Workflow, 14, 19
Workflow Engine, 20
Workflow Language, 15
Workflow Management System, 14, 19
Workflow Participant, 20
Worklet
in YAWL, 146
Worklist
in FlowMark, 153
Worklist Handler
in WORKPAD, 151
inYAWL, 146
WORKPAD, 151
WS-BPEL, 106, 144

X-Orc, 161

YAWL, 90, 106, 144, 145

Zero Infrastructure, 20