Vrije Universiteit Brussel

Faculteit van de Wetenschappen
Vakgroep Computerwetenschappen
Software Languages Lab

Ambient Contracts

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

Christophe Scholliers

Promotoren: Prof. Dr. Wolfgang De Meuter en Prof. Dr. Éric Tanter

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 277 8
NUR 989
Legal deposit D/2013/11.161/039

Samenvatting

Vooruitgang in draadloze netwerktechnologie heeft de realisatie van een nieuw soort applicaties mogelijk gemaakt. Zogenaamde ambiente applicaties bestaan uit mobiele apparaten die spontaan met elkaar communiceren zonder de noodzaak van een centrale server. De netwerktechnologie die dergelijke spontane interacties mogelijk maakt noemt men *mobiele ad hoc netwerken*. De hardware kenmerken van deze nieuwe netwerken — vooral het feit dat de communicatie erg instabiel is en dat de apparaten niet kunnen beroepen op een centrale server — maken dat vorige software-oplossingen voor gedistribueerde systemen onbruikbaar zijn voor de ontwikkeling van ambiente applicaties. Software-technologie heeft zich tot nu toe gericht op het omgaan met de hardware kenmerken van ambiente applicaties, maar geven geen antwoorden op hoe grote en robuuste ambiente software kan gebouwd worden.

Software contracten hebben een belangrijke rol gespeeld bij de ontwikkeling van grote en robuuste software in het algemeen. Software contracten worden op brede schaal toegepast in statisch getypeerde talen en maken momenteel hun intrede in dynamisch getypeerde programmeertalen. Traditionele contract systemen kunnen echter niet worden toegepast op de constructie van ambiente applicaties omdat de wijze waarop deze gevalideerd worden er voor kan zorgen dat er fouten geïntroduceerd worden die niet aanwezig zijn in de oorspronkelijke ambiente applicatie. Daarnaast berusten veel van de huidige contract systemen voor gedistribueerde systemen op een *centrale* server, dewelke niet verenigbaar is met het ad-hoc karakter van ambiente toepassingen. Het werk in dit proefschrift wordt gemotiveerd door het ontbreken van een passend contract systeem voor ambiente toepassingen. Dit belemmert de exploratie naar complexere en meer grootschalige ambiente toepassingen.

In dit proefschrift stellen we een nieuw contract systeem voor dat speciaal ontworpen is voor de ontwikkeling van ambiente toepassingen. Met deze ambiente contracten kan de programmeur uitdrukken welke verplichtingen moeten nageleefd worden tussen de gedistribueerde modules van een ambiente applicatie. Het belangrijkste onderzoeksthema dat we hebben aangekaart is de ontwikkeling van een contract systeem dat rekening houdt met de specifieke hardware-eisen van ambiente toepassingen. We tonen verschillende toepassingen van ambiente contracten en leggen uit hoe de schuld moet toegekend worden in geval van een contractbreuk.

Ambiente contracten zijn geformuleerd als een uitbreiding van AmbientTalk. Naast een praktische implementatie is er ook een formele basis ontwikkeld in PLT-Redex. We hebben het ambient contract systeem gevalideerd door het toe te pasen in het kader van een groot applicatie raamwerk genaamd UrbiFlock. Door onze contracten toe te passen in praktische applicaties hebben we geconstateerd dat vele informele specificaties makkelijk gecodeerd kunnen worden met behulp van het ontwikkelde ambient contract systeem.

Abstract

Advances in wireless technology have enabled the realization of a new brand of distributed applications, called ambient applications. Such ambient applications consists of mobile devices that spontaneously interact with each other without the need for a centralized server. The network technology that enables such spontaneous interactions is called a *mobile ad hoc network*. The hardware characteristics of these new kinds of networks — mainly the fact that the communication is highly volatile and that devices cannot rely on a centralized server — have rendered previous software solutions for distributed systems obsolete. To date, software technology has focused on dealing with these hardware characteristics of ambient applications but has failed to give answers on how to build big and robust ambient software.

A proven software methodology called software contracts has played an important role in the development of big and robust software in general. Software contracts are widely adopted in statically typed languages and are currently finding their entrance in dynamically-typed programming languages. Traditional contract systems however, cannot be applied to the construction of ambient applications because the way in which they are validated can introduce bugs that are not present in the original ambient application. In addition, many of the current contract systems for distributed systems rely on a *centralized* server which is not reconcilable with the ad-hoc nature of the ambient applications. Our work is motivated by the lack of an appropriate contract system for ambient applications which hampers the exploration of more complex and more large scale ambient applications.

In this dissertation, we propose a new contract system, dubbed *ambient contracts* which is specifically designed for the development of ambient applications. With ambient contracts the programmer can express the obligations and promises between the distributed modules of an ambient system. The main research challenge that we address is the adaptation of current contract systems to take into account the specific hardware requirements of ambient applications. We show various applications of ambient contracts, and explain how to assign blame in case of a violation.

Ambient contracts are formulated as an extension of AmbientTalk, a dynamically-typed object-oriented language built upon the principles of the actor model. Next to a practical implementation ambient contracts also have a formal specification which is validated by making use of the PLT-Redex tool. We have validated the ambient contract system, called AmbientTalk/C, by employing it in the context of a large application framework called UrbiFlock. From this use case, we found that many of its previous informal specifications could be easily encoded with the developed ambient contracts.

Acknowledgements

This dissertation would not have been what it is today without the tremendous support from my colleagues, friends and family. With these acknowledgements I would like to thank everybody who has helped me directly or indirectly with obtaining my Phd.

From all these people there are a number of persons who I would like to thank in particular for their impact on my academic life. I thank professors Theo D'Hondt, Viviane Jonckers and Wolfgang De Meuter for teaching and inspiring me during my studies. Their classes were a joy to follow and I can only hope that one day I can inspire a new generation of students just like they inspired the students in my year.

I thank Stijn Mostinckx and Charlotte Herzeel for guiding Eline Philips and me with our master thesis. I also thank all the members of the former PROG lab who gave us brutally honest comments on our work. It was through Stijn and the PROG lab that I became interested in conducting research.

After finishing my master studies many, if not all, members of the lab contributed one way or another to prepare my IWT proposal and prepping me for the famous "minute". Getting my funding from the IWT is truly the merit of all these people together: I will always be grateful for the help I got back then.

I am very thankful to Wolfgang De Meuter for believing in me and giving me the opportunity to start a Phd under his supervision. At a certain moment he even publicly called me the best Phd student he every had. I must admit that at that time I was his first and only Phd student. On a more serious note, I thank Wolf for guiding me when I needed guidance the most. His support ranged from all practical matters up to hard-core research brainstorms. When I could not see the wood for the trees Wolf was there to help me see the bigger picture. I am also thankful that Wolf has given me the freedom to conduct research and explore all paths in research I wanted to explore. Finally, I thank Wolf for bringing me in contact with my co-promotor Éric Tanter.

Being able to work with Éric has taught me a lot of important lessons in my academic life. I especially liked the many discussions over lunch and the fruitful collaboration that flew forth from these discussions. Thank you Éric! I also thank Éric and his students (at that time), Paul, Ismael, Oscar, Guillaume, David and Victor for welcoming me to the Pleiad lab. During the weekends they were there to, take me to concerts, theatre, play football, etc.

Many thanks go out to all the students that I (co)guided in their bachelor and master thesis. Their vivid imagination and their ability to question *everything* has certainly made me a better researcher. To name a few, Jasper Maes did a terribly good job in making an award winning remote controlled R2D2. Ruben Vandamme implemented an award winning multicore-embedded scheme. Wouter Amerijckx ported and extended the Crime language to the sunspot platform, leading to a number of publications. Nick De Cooman did an amazing job on working out alias and ownership control in ambient applications which hopefully will lead to a publication very soon. And finally, Lode Hoste, a special thank you for all the academic fights we had over how an event-driven application should really be programmed [1].

I thank Coen De Roover for guiding students together (for example Wouter) and working together on the Stadium project. If anything the collaboration has been more than fruitful. I can only hope that we can repeat such a fruitful collaboration in the upcoming CHAQ project. I thank Elisa Gonzalez Boix for guiding Nick together and for the work on tuple spaces, taking Crime and the fact space model to the next level.

[1] Yes you also won awards with your thesis (Alcatel, ICSE) but everybody knows that Lode ;)

Working with you has been a joy and I hope that in the future we can find a new baby [2] to work on together. I specially thank Tom Van Cutsem and Dries Harnie for the hours of discussion about the operational semantics of AmbientTalk. Without this collaboration there would not be a formal basis for ambient contracts in AmbientTalk/C. Thanks to the reading committee, Jens Nicolay, Elisa Gonzalez Boix, Dries Harnie and Carlos Noguera for giving corrections on my dissertation text. I would like to thank all the members of the Software Languages Lab of which some have become more than colleagues. Thanks for all the priceless parties we went to, a person mentioned before, Coen, Andoni and Stefan. Thanks also to the people who I had the pleasure sharing an office with Dirk Van Deun, Stijn Timbermont and Lode Hoste.

I thank Yves Vandriessche and Coen for the hacking sessions on the Arduino boards which have led to the now infamous Wii workshops. I hope that these workshops can inspire a new generation of students to start computer science. In the same vain, I thank Yves for the many hours of assembler hacking we did in order to get the Armpit Scheme interpreter working with the display and other hardware. Five years and many first bachelor projects later I can only conclude that it was worth the effort.

I thank all the members of the jury Prof. Dr. Matthew Flatt, Prof. Dr. Erik Ernst, Prof. Dr. Theo D'Hondt, Prof. Dr. Viviane Jonckers and Prof. Dr. Eva Colebunders for taking their time to read my dissertation and challenge me with insightful questions during the defences.

I thank the secretaries of the department which were always ready to help me. Finally, my bank account very much appreciates that (I *have* to mention this): The work in this dissertation has been partially funded by the institute for the Promotion through Science and Technology in Flanders (IWT-Vlaanderen).

Apart from all the academic support my research would not have been possible without the personal support from my friends and family. Thank you (and sorry) to all my friends (too dangerous to enumerate) who had to endure me talking about topics such as meta-circular interpreters, context-awareness, and other light topics such as the bifid encryption mechanism. I hope that the long running inside jokes such as *de boze kaaskroket*, *zijt gij toevallig niet Ellen?*, and wait for it .... *Cola*, at least partially made up for the brainpower I sucked from you guys.

Many thanks goes to Kim Heymans which is a true friend on which I can always count. Thanks for all the times we were together having the time of our life playing foosball, spinning records and playing playstation (thanks Stefanie). All these moments were a great distraction and recharged me so that I could work hard to finishing my Phd.

Bedankt moeke en vake jullie zijn er altijd voor mij geweest. Jullie hebben mij de kans gegeven om naar de universiteit te gaan op een moment dat het er naar uitzag dat studeren niet voor mij was weggelegd. Bedankt voor *al* jullie steun! Bedankt ook aan mijn allerbeste zus Nathalie om Twister in ons leven te brengen, de lekkere cup-cakes en de crazy hard-core feestjes! Ook hartelijk bedankt bonnenke en bompa!

And finally, the moment everybody has been waiting for: A big thank you for my partner in crime, my girlfriend and future wife Elisa Gonzalez Boix. This thesis would not have been what it is today without Elisa. Being together with Elisa has a positive impact on my life in all aspects. T'estimo Elisa!

Thanks for helping, everyone!

– Christophe

[2] Academic, an academic baby !

Contents

1 Introduction **1**
1.1 Research Context . . . 3
1.2 Design by Contract . . . 4
1.3 Problem Statement . . . 5
1.4 Research Approach . . . 6
1.4.1 Language Oriented Approach . . . 7
1.4.2 Runtime Verification . . . 7
1.5 Contributions . . . 8
1.6 Supporting Publications . . . 9
1.7 Dissertation Roadmap . . . 9
1.8 Summary . . . 11

2 Ambient-Oriented Programming **13**
2.1 Middleware for Ambient-Oriented Programming . . . 13
2.2 Wireless Network Applications . . . 15
2.2.1 Technological Advances . . . 15
2.2.2 Integration at the Application Level . . . 17
2.2.3 Ambient-Oriented Programming . . . 18
2.3 Ambient-Oriented Programming in AmbientTalk . . . 19
2.3.1 A Prototype-Based Language . . . 19
2.3.2 A Concurrent Language . . . 21
2.3.3 A Distributed Language . . . 23
2.4 An Early Preview of AmbientTalk/C . . . 25
2.4.1 Ambient Contracts Validation Strategy . . . 25
2.4.2 Behavioral Validation: Computational Contracts . . . 27
2.4.3 Validation of Event-Driven Applications . . . 29
2.4.4 Validation of Distributed Applications . . . 29
2.4.5 Ambient Contracts Overview . . . 31
2.5 Conclusion . . . 32

3 Related Work: Software Verification Techniques **33**
3.1 Categorization of Software Contracts . . . 33
3.1.1 Design considerations for AmbientTalk/C . . . 35
3.1.2 Conclusion . . . 37
3.2 Survey of Related Work . . . 37
3.2.1 Higher-Order Behavioral Verification Techniques . . . 37
3.2.2 Behavioral Contracts for Concurrent and Distributed Systems 43
3.2.3 Systems for Synchronization Verification . . . 44

3.2.4 QoS Contracts . . . 48
3.2.5 Aspect-Oriented Programming and Contracts . . . 48
3.3 Conclusion . . . 49

4 Communicating Event-Loops: Formal Specification 53
4.1 AmbientTalk Operational Semantics . . . 53
4.1.1 Syntax . . . 53
4.1.2 Reduction Rules . . . 57
4.1.3 Service Discovery . . . 62
4.1.4 Robust time-decoupled message transmission . . . 64
4.2 Conclusion . . . 66

5 Computational Contracts for Functions 67
5.1 Higher-Order Contracts in a Nutshell . . . 69
5.1.1 First-Order Function Contracts . . . 69
5.1.2 Higher-Order Pre/Post Contracts . . . 70
5.2 Computational Contracts . . . 71
5.2.1 Prohibit Contracts . . . 72
5.2.2 Ensure Contracts . . . 74
5.2.3 Usage Protocols . . . 75
5.3 Contract Verification and Blame Assignment . . . 77
5.3.1 Flat and Higher-Order Pre/Post Contracts . . . 77
5.3.2 Aspect Extensions for Computational Contracts . . . 79
5.3.3 Verification and Blame Assignment of Computational Contracts 81
5.3.4 Computational Contracts: Step by Step Example . . . 82
5.4 Operational Semantics of Computational Contracts . . . 83
5.4.1 CEK Model and Syntax Definition . . . 83
5.4.2 Higher-Order Aspect Language . . . 86
5.4.3 Computational Contracts . . . 89
5.5 Discussion . . . 90
5.5.1 Identity, Sameness, and Difference . . . 90
5.5.2 Who will Guard the Guards? . . . 92
5.6 Conclusion . . . 92

6 Server Side Ambient Contracts 95
6.1 Object-Level Contracts . . . 97
6.1.1 Flat Object Contracts . . . 97
6.1.2 Higher-Order Object Contracts . . . 98
6.2 Computational Contracts over Objects . . . 100
6.2.1 Method-level Computational Contracts . . . 100
6.2.2 Computational Contracts for Object Protocols . . . 101
6.3 Parametrized and Parametric Polymorphic Contracts for Objects . . . 105
6.3.1 Parameterized Contracts . . . 105
6.3.2 Relationally-Parametric Polymorphic Contract Inference . . . 106
6.3.3 Executable Semantics . . . 108
6.4 Blame Assignment in OO Higher-Order Contracts . . . 111
6.4.1 Recursive Higher-Order Object Contacts . . . 111
6.4.2 Blame Assignment for Recursive Higher-Order Contracts . . 112
6.5 Operational Semantics of Object Contracts . . . 113
6.6 Future Contracts for Event Loop Concurrency . . . 114

6.6.1 Contracts for Futures . . . 115
6.6.2 Callbacks and Computational Contracts . . . 117
6.7 Contract Validation and Blame Assignment in Event Loop Contracts . 117
6.7.1 Future Type Message Contract: Blame Inversion . . . 117
6.7.2 Executable Semantics of Future Contracts . . . 118
6.7.3 First-Class Aspect Environments for Validating Callbacks . . 119
6.8 Operational Semantics of Higher-Order Event Loop Contracts . . . 120
6.9 Conclusion . . . 122

7 Client Side Ambient Contracts 125
7.1 Flat Far Reference Contracts . . . 126
7.1.1 Flat Far Reference Contracts in Action . . . 126
7.1.2 The lifetime of a Far Reference Contract . . . 128
7.1.3 Atomicity . . . 129
7.1.4 Flat Far Reference Contracts Overview . . . 130
7.2 Far Reference Contracts: The Problems . . . 131
7.2.1 Running Example . . . 131
7.2.2 Coarse-Grained Blocking: Delay all Message Processing . . . 131
7.2.3 Fine-Grained Blocking: Delay the Contracted Message . . . 132
7.2.4 Contract Decomposition . . . 133
7.2.5 Conclusion . . . 134
7.3 Validation and Blame Assignment of Flat Far Reference Contracts . . 134
7.3.1 Far Reference Contracts Overview . . . 134
7.3.2 Flat Far Reference Contracts Implementation . . . 135
7.4 Higher-Order Far Reference Contracts . . . 137
7.4.1 Higher-Order Far Reference Example . . . 137
7.4.2 Computational Far Reference Contracts . . . 138
7.5 Validation and Blame Assignment of Far Reference Contracts . . . 140
7.6 Computational Message Contracts . . . 141
7.6.1 Message Contracts . . . 141
7.6.2 Prohibit Message Contract . . . 142
7.6.3 Ensure Message Contracts . . . 142
7.6.4 Protocol Message Contracts . . . 143
7.7 Validation of Messages Contracts . . . 144
7.8 Conclusion . . . 145

8 Ambient Contracts at Work 147
8.1 UrbiFlock Framework . . . 148
8.1.1 Architecture . . . 148
8.1.2 Software Specifications in UrbiFlock before AmbientTalk/C . 150
8.2 Contracts over the Graphical User Interface Module . . . 151
8.2.1 Experiences . . . 153
8.3 Contracts over the Application Module . . . 154
8.3.1 Experience . . . 156
8.4 Contracts over the Application IR8U Module . . . 157
8.4.1 IR8U Overview . . . 158
8.4.2 Contracts . . . 160
8.4.3 Experience . . . 162
8.5 Conclusion . . . 163

9 Conclusion **165**
9.1 Summary and Contributions 165
9.2 Shortcomings and Future Work 168
9.3 Conclusion 169

A Communicating Event Loop Calculus in PLT-Redex **171**
A.1 Basic Example of the PLT-Redex Semantics 171
A.2 Syntactic Sugar 172
A.3 Advanced Example of the PLT-Redex Semantics 173
A.4 Reduction Rules 174

Bibliography **176**

List of Figures

2.1 Comparison of wireless communication technology [BC06]. 16
2.2 Object definition in AmbientTalk. 20
2.3 Extending an object in AmbientTalk. 20
2.4 Defining and applying a block. 21
2.5 Keyworded function definition in AmbientTalk. 21
2.6 Communicating event loop model. 22
2.7 Future type message passing in AmbientTalk. 23
2.8 Publishing objects in the Ambient for remote discovery. 24
2.9 Object discovery in AmbientTalk. 24
2.10 Dealing with disconnections in AmbientTalk. 25
2.11 Errors signaling without contracts (left) and with Contracts (right) . . 26
2.12 Contracts in a system with first-class functions. 27
2.13 A computational contract example. 28
2.14 Supplier Module 29
2.15 Server side and client side contracts over objects. 30
2.16 Distributed ambient contracts 30
2.17 The ambient contracts design space. 31

3.1 Overview of the categorizations of contracts by Beugnard et al [BJPW99] 34
3.2 Example: Interface Contract in a Java Like Language. 34
3.3 Example: Behavioral Contract in a Typed Language. 35
3.4 Example: Synchronization Contract in a Java like Language. 35
3.5 Overview of the design space of ambient contracts explored by the related work. 51

4.1 Semantic entities of AT-LITE. 54
4.2 AT-LITE Syntax 56
4.3 Substitution rules: x denotes a variable name or the pseudovariable this. 57
4.4 Actor-local reduction rules. 61
4.5 Actor-global reduction rules. 62
4.6 Auxiliary functions and predicates 63
4.7 Reduction rules for service discovery 64
4.8 Reduction rules for time-decoupled message transmission 65

5.1 Situation of Chapter 5 in the ambient contract design space for functions. 68
5.2 Example math module providing a contracted function. 69
5.3 The AmbientTalk prompt: signaling a contract violation. 70
5.4 Higher-order pre/post contract over the function 70

5.5 The AmbientTalk prompt: signaling a higher-order contract violation. 71
5.6 Using a prohibit contract to prevent the `sqrt` to display text. 73
5.7 Defining a prohibit contract over the argument of the `map_pos` function. 73
5.8 Using the contracted `map_pos` function (from uses.at). 73
5.9 Exporting the function `display-average` with a promise contract. . 74
5.10 Ensure protocol contract over the `readCharFromFile` function. . . . 75
5.11 Defining a prohibit protocol contract over the `readAndShow` function. 77
5.12 Higher-Order pre/post contract constructors. 78
5.13 A higher-order contract definition over the function fixTen. 78
5.14 Aspect to validate the argument of the `sqrt` function. 81
5.15 Computational contract constructor. 81
5.16 High-Level Abstraction for Computational Contracts. 82
5.17 Computational contract implementation in use. 82
5.18 Core syntax of the λ_c language 84
5.19 Basic reduction rules of the CEK machine for higher-order contracts. . 85
5.20 Syntax of the λ_c language 87
5.21 Reduction rules for aspects. 88
5.22 Meta functions to weave aspects. 89
5.23 Reduction rule for computational contracts. 90
5.24 Syntactic sugar for the definition of Computational Contracts. 90
5.25 Example where a computational contract and a function contract can be active over the *same* function. 91

6.1 Situation in the ambient contracts design space for objects. 96
6.2 Situation in the ambient contracts design space for futures. 96
6.3 Flat Object Contract. 97
6.4 Flat Object Contract Usage. 98
6.5 Object contract definition and deployment. 99
6.6 Violation of the stack contract by the module uses.at. 99
6.7 Computational contract over methods 100
6.8 Methods with a computational contract blame assignment. 100
6.9 Initialization Protocol. 102
6.10 Deactivation Protocol. 102
6.11 TypeQualifierProtocol. 103
6.12 Dynamic Preparation Protocol. 104
6.13 Parametrized contract constructor function for stack contracts. 105
6.14 Stack module making use of a parametric contract. 106
6.15 Parametric object contract violation. 106
6.16 Parametric polymorphic identity function. 107
6.17 Applying the parametric polymorphic identity function. 107
6.18 Parametric polymorphic contract for the function. 107
6.19 Applying Parametric Polymorphic Contract Functions 108
6.20 Applying Parametric Polymorphic Contract with Objects 108
6.21 Parametric Polymorphic Contract Inference 110
6.22 Data structure under contract. 111
6.23 Unfair contract validation. 111
6.24 Fair contract validation. 112
6.25 Object contract definition. 113
6.26 Blame order in the math example. 113
6.27 Extension for Contract Syntax. 114

6.28 Recursive Contract Semantics 115
6.29 Server providing a function with a future contract. 115
6.30 Client using a future contract. 116
6.31 Future blame. 116
6.32 Example of callbacks in combination with a computational contract. . 117
6.33 Passing (left) and resolving (right) a future from the client module to the server module. 118
6.34 Future contract constructor. 119
6.35 Definition of a variant on the block. 120
6.36 Aspect capturing registration construct. 121
6.37 Extension for Event Loop Contracts Syntax. 121
6.38 Object-Oriented Aspect Language 122
6.39 Extensions for reifying the aspect environment. 122

7.1 Situation in the ambient contracts design space. 126
7.2 Defining and exporting the coordinator object on the server actor. . . . 127
7.3 Discovering and registering with the coordinator by the client actor. . 127
7.4 Asynchronous contract violation. 128
7.5 Specifying remote-object assertion constructors with a custom validation lifetime. 129
7.6 Specifying a contract with an atomically executed assertion. 130
7.7 Assertion variations supported by flat far reference contracts. 131
7.8 Coordinator (Actor) 131
7.9 Deadlock Example: Client (Actor) 132
7.10 Transmission ordering: Server side (Actor) 133
7.11 Transmission ordering: Client side (Actor) 133
7.12 Transitively passing far references. 133
7.13 Validating asynchronous contracts *before* processing a contracted message. 134
7.14 Asynchronous Contract: Sequence Diagram 135
7.15 Flat far reference contract constructor. 136
7.16 AddPlayer contract validation example. 137
7.17 Far Reference Contracts: Stack Example (Uses) 139
7.18 Far Reference Contracts: Stack Example (Defs) 139
7.19 Computational Far Reference Contract: Stack Example (Defs) 139
7.20 Computational far reference contract violation. 140
7.21 Implementation of Far Reference Contracts 140
7.22 Providing a function with a prohibit computational contract for messages. 142
7.23 Computational message contract violation. 142
7.24 Providing a function with an ensure computational contract for messages. 143
7.25 Computational message contract violation. 143
7.26 Message ordering contract example. 143
7.27 Message ordering contract example. 144
7.28 Aspect example of intercepting message sends 144
7.29 Prohibit message contract constructor. 145

8.1 UrbiFlock Design Diagram 148
8.2 Comments in the Application Module. 151
8.3 Graphical User Interface, Hello World example. 151
8.4 Graphical User Interface, Hello World example screenshot. 151

8.5 Contracts defined over the graphical user interface module. 152
8.6 State diagram of an UrbiFlock application. 154
8.7 Application contracts protocol implementation. 155
8.8 Contracts defined over the application listener. 156
8.9 Contracts defined over the application module. 157
8.10 Screenshot of IR8U application in Urbiflock 158
8.11 Overview of contractual bindings of the IR8U application. 160
8.12 Exporting an application. 161
8.13 Ambient contracts over the remote application interface of IR8U. . . . 162
8.14 Computational contracts for mandatory super calls. 162
8.15 Message contract for asking ratings. 163

A.1 Example AmbientTalk code snippet. 171
A.2 Plt-Redex: AmbientTalk code snippet. 171
A.3 Plt-Redex: Actor code snippet. 172
A.4 Tracing the execution of an AmbientTalk program 172
A.5 Plt-Redex Reduction graph. 172
A.6 Syntactic sugar 173
A.7 AmbientTalk, sending an asynchronous message 173
A.8 Plt-Redex: sending an asynchronous message 173
A.9 Plt-Redex Reduction graph of an asynchronous message. 174
A.10 Plt-Redex reduction rules for the AmbientTalk language. 175

List of Tables

3.1 Higher-Order Contract Validation. 42

5.1 Overview of the high-level functional computational contracts provided in AmbientTalk. 72

6.1 Aspect Extensions for First Class Aspect Environments. 120

7.1 Computational contracts over messages. 141

Chapter 1

Introduction

It is now more than 20 years ago that M. Weiser postulated his vision for *"The computer of the 21st Century"* [Wei91]. In his vision, computer interactions are no longer tied to a single location which usually involve a chair, a screen, a keyboard and a mouse. Instead computing is envisioned to become an unobtrusive and distraction-free activity in a technology-rich environment. In such environments where computing is omnipresent, personal computers are shifted to the background and controlled through the set of all physical objects in the environment.

Moving closer to realizing Weiser's vision is not possible without improvements in *both* hardware and software technology. Current day mobile devices are characterized by their support for advanced networking technologies like Wi-Fi, Bluetooth and more recently Near Field Communication (NFC). Such networking technologies enable mobile devices to spontaneously interact with each other and thus form an enabling technology towards Weiser's vision. The hardware characteristics of these new kinds of distributed hardware — mainly the fact that the communication is highly volatile and that devices can not rely on a centralized server — have rendered previous software solutions for distributed systems obsolete.

In order to overcome the software problems introduced by this new hardware, the ubiquitous computing field has been undergoing three main generations of research challenges [Com11]. The first generation of ubiquitous computing systems was mainly driven by technological advances in wireless communication which enable collocated devices to spontaneously interact without the need for a centralized infrastructure. The research goal was to literally connect everything to everything and has been termed *connectedness* (1991-2005). The second generation of systems was driven by the need to adapt running systems according to their context and has been termed *awareness* (2000-2007). At the heart of many of these systems lies a form of reasoning engine which is fed by data from sensors that record the current context. Once this context is processed the application dynamically adapts itself according to the available information. These challenges are addressed in research areas such as context-oriented programming [HCN08, CH05], context awareness [DSA01, ADB^{+}99], resource-awareness, etc. Finally, the third generation ubiquitous computing research, building upon awareness and connectedness, has focused on giving meaning to services and interactions in order to make them smarter, leading to a research field dubbed *smartness* (2004-). Such systems can be described as highly complex dynamically orchestrated and coordinated groups of interacting nodes. A prominent factor in such systems is the scale and complexity of the developed applications.

While the problems of connectedness and awareness are mainly understood, dealing with the challenges of smartness is still an ongoing research activity. It is our vision that the development of large scale complex ubiquitous systems can only be realized when the programmer is equipped with *appropriate software engineering abstractions*. Previous efforts to aid the developer into writing ubiquitous applications already had a significant impact on the software development process. However, they focus on abstracting away from the hardware characteristics of the devices on which ambient applications are deployed in order to provide *connectedness*.

When developing (large scale) ubiquitous applications, the non-functional requirements become prevalent over the pure functional requirements. One particular successful way to deal with these requirements in sequential software is by making use of *contracts* [Mey92]. The contracts defined over a module stipulate both the obligations and promises of clients and suppliers so that the programmer does not need to be concerned about the internal details. Recently the work on contracts systems have been revived with work on higher-order contracts [FF02, FB06, GPW10] and distribution [CP09, ABZ10]. Contract systems allow the programmer to abstract away from the internal details of modules only when they are powerful enough to express the functionality of the abstractions that they monitor. We argue that current contract systems fall short in expressing the (temporal) behavior of ubiquitous applications. Moreover, in certain situations when a contract is violated they fail to correctly point out the responsible of the violation.

In this dissertation, we focus on the formulation of a contract system that can capture ubiquitous requirements so that the programmer can express the obligations and promises of his modules. The main research challenge that we address is the adaptation of current contract systems to take into account the specific requirements of ubiquitous applications. Many of the current contract systems for distributed systems rely on a *centralized* server which is not reconcilable with the ad-hoc nature of the targeted applications. Moreover, many of the standard ways to validate contracts are not applicable in a distributed context. The reason is that they change the semantics of the underlying program in order to verify the contract. These kind of contract systems are not satisfactory as their usage might introduce deadlocks which are not present in the original software. *The work in this dissertation is thus motivated by the quest for a contract system that enables the programmer to write robust software for distributed applications that communicate over unreliable networks.* This is also the main research challenge addressed in this dissertation: to bridge the gap that makes it impossible to apply current day contract systems for the development of robust ubiquitous applications.

A large body of software engineering research has been conducted in order to overcome the limitations of traditional tools for the development of ubiquitous applications. Also at the Software Language Lab (SOFT), where this research has been conducted, a great effort has been made in order to deal with the new generation of ubiquitous hardware. Our lab has formulated answers towards connectedness under the form of a new programming paradigm dubbed ambient-oriented programming [VMG$^+$07]. At the heart of this paradigm lies a set of programming language abstractions that deal with the volatility of the connection over which the distributed nodes communicate. Many research efforts have also focussed on awareness with research artifacts such as Fact Spaces [MSP$^+$07], TOTAM [SGD09] and Context L [CH05]. Finally, in recent years these research artifacts have been combined in order to realize more complex dynamically orchestrated applications. A prominent example of the later can be found in the development of UrbiFlock [GBLCS+11], a ubiquitous social networking application that builds upon both the research efforts coming from AmbientTalk and TOTAM. Naturally, the work presented in this dissertation is an extension of the research that we previously conducted at the Software Languages Lab.

In summary, current day ambient software engineering tools fall short for the development of large scale ubiquitous applications. Traditional contract systems can not be applied for the construction of such applications as they rely on a centralized server or might alter the semantics of the original software. The lack of an appropriate contract system for ubiquitous applications hampers the exploration of the third research track in order to fulfill the vision of Weiser. In this dissertation, we propose a new contract system, dubbed *ambient contracts*, which adheres to the characteristics of ubiquitous applications. With this contract system the programmer can express the obligations and promises between the distributed modules of a ubiquitous system.

1.1 Research Context

The research conducted in this dissertation spans a number of research fields from different branches of computer science. We describe each of the domains and sketch why each of the research domains are relevant for the formulation of our solution.

Ubiquitous Computing is the research vision postulated by M. Weiser as explained in the introduction. It has spawned a tremendous amount of research over the different areas of computer science. The vision now postulated more than two decades ago is slowly becoming a reality. While many individual research challenges are solved, it remains to be seen how all these research efforts can be combined in order to develop large scale applications. It is this observation that has led us to develop software engineering abstractions under the form of software contracts in order to aid the developer in building large scale ubiquitous applications.

Mobile Ad Hoc Networks are the enabling technology that allows the concretization of Weiser's vision. While the hardware technology of mobile devices is constantly improving, the characteristics of mobile ad hoc networks are significantly different from traditional distributed networks. The programmer needs to take these differences into account when writing ubiquitous applications, and has to do this for every mobile application that he/she writes. This observation has led to the development of software abstractions specifically designed for the development of applications that make use of mobile ad hoc networks. The characteristics of mobile ad hoc networks naturally have to be taken into account for the abstractions advertised in this dissertation.

Software Verification is the process of checking that a software system adheres to a set of initial design requirements. Verification can be used for many purposes, such as debugging, testing, validation, profiling, fault protection, behavior modification (e.g. recovery), etc. The term *verification* is mostly associated with static verification. However, there are two types of verification, runtime verification and static verification. We argue that the dynamic and the ad-hoc nature of mobile ubiquitous applications are better suited towards runtime verification. In this dissertation, contracts are verified during the execution of ubiquitous applications. Moreover, through reflective capabilities, runtime verification can be made an integral part of the underlying language.

1.2 Design by Contract

Contract frameworks have been around for a very long time but the first appearance of pre and post-conditions can be found in the work of Tony Hoare [Hoa72]. Tony Hoare observed that the validity of a program result can often be determined solely based on the input and the output of the program. Therefore, validating the input of a program is as important as validating the output of the program. In his work he developed a formal model under the form of what is now known as a Hoare triple [Hoa72]. A Hoare triple $P\{Q\}R$ consists of a precondition P, a subprogram Q and a postcondition over the result R. The precondition of a Hoare triple validates the input of the subprogram Q while the postcondition validates its result. By chaining such subprograms together certain properties about the composition of the program can be proved.

The ideas of Hoare were adopted and implemented in the programming language CLU by Barbara Liskov [LSAS77]. Afterwards, they were adopted in a wide range of programming languages but were mostly influenced by the programming language Eiffel [Mey88]. Together with the Eiffel language many terms used to describe properties of a contract systems were introduced, most notably the term Design by Contract (DbC). Meyer was inspired by contracts in the business world where the parties of a contract are called the *client* and the *supplier*. Similarly to a business contract, a software contract specifies obligations that must be fulfilled by the involved parties. In DbC the obligation that must be fulfilled by are:

- Preconditions, which state the conditions the client of a supplier must meet.

- Postconditions, state the guarantees that the supplier fulfills given that the client meets the preconditions.

Many times the contractual obligations are specified on the module boundaries. When a module imports another module, the two modules enter into a contractual agreement. The contract system keeps tracks of the obligations and promises of the supplier and client of the values that flow between the modules. When a violation against a contract is detected, the contract system points out the responsible module. The process of keeping track of the obligations and promises of the values in order to point out the responsible module when a violation is detected is called *blame assignment*.

The large proliferation of external frameworks which provide support for DbC [LCC$^+$05, FF01, DH98, LKP02, KHB99] is a proof of the wide adoption of the design by contract methodology in current programming languages. However, all these contract frameworks are incarnations of basic pre/post contract systems. They are not directly applicable for use in an ambient-oriented programming language because they do not integrate well with the basic programming language constructs for programming mobile distributed application. One of the observations is that basic pre/post contract systems do not monitor the subprogram Q. In an ambient-oriented programming language this subprogram Q can send messages or change the state of the program by means of side effects. Such concerns are important in the context of ubiquitous systems but they are hard to express with current contract systems. In the following section we give an overview of the limitations of current contract systems.

1.3 Problem Statement

From previous research efforts it has become apparent that developing applications for mobile ad hoc networks is substantially different from developing applications for fixed computer networks because of two differentiating characteristics [VMG$^+$07]: nodes in the network only have intermittent connectivity (due to the limited communication range of wireless technology combined with the mobility of the devices) and applications need to discover and compose services without relying on additional infrastructure such as a centralized server. These properties do not map well onto regular contract systems for general purpose programming languages [DVM$^+$05] which treat disconnections as fatal errors and assume that all communication is stable. As a result, it is currently extremely hard to program and debug applications that are deployed in such a highly dynamic environment.

The lack of abstractions dedicated to *verify* ubiquitous or ambient applications [1] results in complex and unmaintainable code [UBPU08]. In this dissertation, we present a novel programming abstraction called *ambient contracts* to deal with these difficulties. In the development of any contract system it is important that the contracted system behaves exactly as the original program in case the program does not violate any contract. In case the program does violate a contract there should be a meaningful report about the cause of the problem and which module is the responsible for the violation. In the following paragraphs we give more detail of the different problems that a programmer has to face when using current contract systems for developing ambient applications.

Complex First-Class Values and Blame Assignment. Prominent programming paradigms today all have support to build complex first-class values. This is no different for the language constructs that have been developed for the ambient-oriented paradigm. Programmers exchange objects, functions, futures, and remote references between distributed communication partners without having to worry about low-level serialization concerns. While contracts for higher-order functions have been investigated, the use of complex first-class values such as remote references or future values poses new challenges which existing contract systems can not deal with. The use of these complex first-class values is so prominent that a contract system that does not support such values would be very limited in expressiveness. Therefore, the use of complex first-class values should be natively supported. From previous research it has become clear that when supporting higher-order values at runtime determining who is the responsible for a violation is no longer trivial. In short, an ambient contract system should keep track of who to assign blame in case a violation against a contract is detected, even in the presence of complex first-class values.

Capturing Ubiquitous Behavioral Properties. One of the main advantages of a contract system is that it provides the programmer with an enforceable specification of the obligations and promises of using certain values of his program. To this end, the contract system must be sufficiently expressive for the target domain. In the case of ubiquitous applications this means that the contract systems should be able to express aspects such as prohibiting or enforcing sending messages over the network, mandatory method invocations, access permission, memory constraints etc. Current contract systems, however, have mainly focused on pre- and post-conditions. They can not

[1] We use the terms ubiquitous and ambient application interchangeably.

validate that certain events happen or should not happen during the execution of a contracted value in a ubiquitous setting. In this dissertation, we investigate how we can overcome the problems of current higher-order contract systems to allow validation of the computation of a function. As these concerns could not be expressed even on a single node, a first research problem was the formulation of a contract system that can validate concerns over the internal working of a function.

Scoping of Behavioral Contracts. Ubiquitous applications are inherently concurrent. The programming paradigms for ubiquitous computing have addressed the complexity of writing concurrent applications by a model called *communicating event-loops*. This model provides the user with language constructs that mitigate the problems of dealing with concurrency and volatile connections. For example, language constructs such as non-blocking futures provide the programmer with a way to simulate synchronous communication in an asynchronous communication model. However, these language constructs also introduce the notion of event handlers which are triggered when the results of sending an asynchronous message becomes available. The scope and the blame assignment of behavioral contracts in combination with such callback mechanisms — that can trigger at any moment in time — is not investigated by current contract systems.

Non-Interference. When naively inserting contract validation code into a running program the semantics of the underlying program might be altered. Of course this is an unwanted property and should be avoided as much as possible. In the context of ubiquitous applications maintaining interference-free contract validation is not as trivial as it might seem. First, the programmer can not assume that a centralized server is always available because communication is set up in an ad-hoc fashion. This complicates contract validation as there is no longer a centralized point where the contract can be verified. Second, connections are volatile and can be broken at any moment in time. Therefore, a contract system should never block the computation, in order to avoid introducing deadlocks which were not present in the original program. This observation is in line with previous research effort that showed that blocking programming language abstractions can not be reconciled with the development of ubiquitous applications. In short, an ambient contract should not interfere with the distributed semantics of the program even in the presence of volatile connections.

In conclusion, the main research problem that we address in this dissertation is the formulation of an *expressive* contract system that can deal with *complex first-class values* in the presence of highly *volatile connections*.

1.4 Research Approach

A large part of the work described in this dissertation deals with programming language design. This choice is motivated by a number of practical and cultural considerations. Language research at the Software Languages Lab (formerly known as Programming Technology Lab) has a long tradition of conducting research with so called *little languages* [Ben86]. Examples include Agora [CDDS94] and AmbientTalk [VMG$^+$07]. In this section we motivate the choice for a language-oriented approach. The second major design decision for the definition of an ambient contract system is whether the contracts are verified statically or at runtime. In this section we argue for runtime verification as opposed to static verification.

1.4.1 Language Oriented Approach

The main advantage of a language-oriented approach is the ability to define the system as a whole rather than a composition between the framework and the underlying language in which the framework is defined. It is well known from prior work that the composition of certain software systems that foster their own rules can lead to an "impedance" mismatch. A prevalent example can be found in the database world where relational databases are mapped onto objects and vice versa [CD96]. Another example closer to ubiquitous computing can be found in the mismatch between an event-driven system and a multithreaded program. Composing those two leads to a system where the invariants governing the event-driven system can be easily broken by the multithreaded program. Thus, the main problem is to compose these systems into one coherent system that maintains the global properties, such as deadlock freedom, despite the composition. Maintaining such properties is much harder with a library or a middleware than with a programming language. The reason is that a middleware or a framework might not always be able to control parts of the underlying language needed to truly enforce certain properties. On the contrary, when designing language abstractions this composition can be avoided as the language designer has complete control over the implementation of every aspect of the language. This makes a language-oriented approach more suitable for the development of a contract system than the design of a middleware or a framework. A good language design can truly *enforce* certain program properties which have been specifically designed to address the problems at hand.

1.4.2 Runtime Verification

Two main approaches exist for the verification of contracts: runtime assertion checking (RAC) and static verification [CR06]. Both approaches have their advantages and disadvantages.

Runtime verification consists of monitoring the application while it is executing. The programmer defines a set of assertions that define correct executions of the program and the contract system verifies these assertions at runtime. When the contract system discovers a violation of one of those tests, it reports the violation. Runtime verification of contracts was popularized by the contract system in Eiffel [Mey91].

Static verification consists of a logical proof system that, given a set of specifications and the program code, verifies that no violations of the specifications can occur at runtime. Static verification implies that the verification is based on the program text and that the program is not executed during the verification phase. To be sound, static approaches need to be conservative. This means that they may report a violation even though the program actually does not violate the specification at runtime.

Both approaches have their merits. Runtime assertion checking systems often allow the programmer to express more than their static verification counterparts. Static verification techniques, on the other hand, allow the programmer to catch violations earlier, i.e. even before the program is executed. This advantage does not come without a cost. Generally, static verification tools require fairly elaborate specifications of the modules being checked. Furthermore, in order to be effective, static verification tools require specifications that are usually more complex than the specifications needed for runtime verification. And finally, current complex static verification mechanism do not scale very well for very large software projects. As we argue in this dissertation, the use of static verification is hard to combine with the dynamic nature of mobile ad-hoc network applications where the communication partners during a program execution might not be known or available at deployment time.

1.5 Contributions

This dissertation makes the following contributions in the intersection of programming language design, mobile ad hoc networks and runtime verification:

Survey of Related Work. We give an extensive survey of related work on program validation. Many contract monitoring and verification mechanisms have been designed. However, only few take into account both the dynamic nature of the programming language and the dynamic nature of ambient applications. From this survey of related work we conclude that the higher-order contract mechanisms [FF02] developed in the context of the dynamic language Scheme [KCR98] form the most suitable foundation for the specification of an ambient contract system.

Communicating Event Loop Calculus. Together with Tom Van Cutsem and Dries Harnie, we defined an operational semantics of the AmbientTalk programming language [VCSHDM]. We believe this is the first formal account of an actor-based language whose concurrency model is based on communicating event loops. The usability of the semantics has been validated by a direct translation of the reduction rules into an executable program by using the PLT-Redex [FFF09] tool. The advantage of having an executable semantics is that it enables us to try out example programs and validate their operation. Because of the non-deterministic nature of our semantics, writing out the evaluation steps for the execution of even a small program would require a big effort. Moreover, due to the sheer amount of evaluation steps involved in the execution of a simple program executing it by hand would be very prone to error.

Computational Contracts. We have developed the concept of computational contracts. A computational contract is a contract over the execution of a contracted entity. One example of a computational contract is a contract over a function to ensure that this function updates the GUI when applied. Computational contracts can verify well-defined execution points during the execution of a contracted entity. With computational contracts the developer can define a function contract that verifies a single event or a sequence of events during the execution of the contracted function. We have defined computational contracts that can be used in combination with object-oriented and event-loop concurrency language constructs.

Ambient Contracts. Building further on the concepts of computational contracts we have developed the concept of ambient contracts in a contract framework called AmbientTalk/C. Ambient contracts allow the specification of behavioral constraints over distributed objects which communicate over an unreliable network. Ambient contracts allow assertions over remote references to be validated without interfering with the messages ordering semantics of the program. We have defined ambient contracts that can be used in combination with object-oriented language concepts and event-loop language constructs such as futures.

Validation. We have applied the AmbientTalk/C framework to a framework for the rapid prototyping of context aware social networking applications called *UrbiFlock*. This framework constitutes of almost 4000 lines of AmbientTalk code and another 3000 lines of Java GUI code. We give an overview of the most important ambient contracts applied throughout this framework and report on our experience of using the AmbientTalk/C contract framework.

1.6 Supporting Publications

Of the (co-)authored publications that are related to connectedness [RSA+12, GSL+10], awareness [MSP+07, SBMD10], and smartness [SHSDM11, STM11, SHT+11, GBLCS+11], the following introduce the key ideas of this dissertation:

Ambient Contracts: Verifying and Enforcing Ambient Object Compositions a La Carte *Personal Ubiquitous Computing. Christophe Scholliers, Dries Harnie, Éric Tanter and Wolfgang de Meuter.* [SHT+11] The work described in this paper is an early version of what we describe in this dissertation. It defines contracts over a group of remote references. In this dissertation, we have elaborated on these ideas and extended them with computational contracts. For the later we also have incorporated blame assignment.

Computational Contracts: Grey Box Verification and Blame Assignment in a Higher-Order Setting *Proceedings of the 2011 Workshop on Scheme and Functional Programming. Christophe Scholliers, Éric Tanter and Wolfgang De Meuter.* [STM11] The work in this paper lays the foundations on which we build ambient contracts. More specifically, it gives an overview of how higher-order values can be combined with behavioral contracts.

Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications *Symposium on Applied Computing. Elisa Gonzalez Boix, Andoni Lombide Carreton, Christophe Scholliers, Tom Van Cutsem, Wolfgang De Meuter and Theo D'Hondt.* [GBLCS+11] In this paper the UrbiFlock framework is explained which forms the basis of the validation chapter.

1.7 Dissertation Roadmap

Chapter 2. Ambient-Oriented Programming provides the reader with an introduction to the main requirements for the ambient-oriented paradigm (AmoP). We describe the paradigm as proposed in previous work by Dedecker [DVM+05] and Van Cutsem [VMG+07]. A thorough overview of these requirements deduced from the hardware layer is very important not only for the definition of language constructs for ambient-oriented programming but also for the software engineering support as proposed in this dissertation. At the end of chapter 2 we give a sneak preview of the AmbientTalk/C framework.

Chapter 3. Related Work: Software Verification Techniques gives an overview of the landscape of contract systems. Many available contract systems provide powerful specifications but are not always applicable in an ambient environment. We conclude that the flexibility and the dynamic nature of ambient oriented systems are best matched with current higher-order contract systems. However, not all problems encountered in a ubiquitous setting can be tackled by these systems.

Chapter 4. Communicating Event-Loops: formal Specification gives a formal overview of the underlying event-loop concurrency model [VCSHDM], the computational model for concurrency and distribution, that adheres to the AmOP criteria presented in chapter 2. We start with a quick overview of the basic language constructs of the AmOP paradigm followed by the operational semantics. The communicating event loops calculus we present in this chapter is used in the remainder of the dissertation to define the language constructs that constitute ambient contracts. The formulation of this operational semantics is used in later chapters in order to define the semantics of ambient contracts.

Chapter 5. Computational Contracts for Functions This chapter investigates the first step towards ambient contracts. We define a novel contract system to validate concerns over the internal working of functions. In this chapter we limit ourself to the explanation of computational contracts for functions.

Chapter 6. Server-Side Ambient Contracts gives an overview of how computational contracts are combined with language constructs specifically for the ambient-oriented paradigm. It shows how higher-order contract systems can be unified with a prototype-based object model, event loop concurrency and futures values. In this chapter we define contract that are defined over objects hosted by the same actor. These contracts can be used in order to protect the server from misbehaving clients.

Chapter 7. Client-Side Ambient Contracts In this chapter, we propose an novel asynchronous contract system. In this contract system, the validation of contracts over remote references is postponed until a message is sent to the contracted (remote) value. By postponing the verification, non-interference of the contract with respect to deadlocks and message ordering can be ensured. We describe an expressive model to specify and verify asynchronous contracts over remote objects. Subsequently, we give an overview of how to validate and assign blame for higher-order far-reference contracts. Finally, we give an overview of how computational contracts can be used in order to specify contracts over the outgoing messages of a contracted entity. The contracts defined in this chapter can be used in order to protect the client from misbehaving servers.

Chapter 8. Ambient Contracts at Work presents the validation of the developed contract system by means of a concrete use case. We apply ambient contracts to a context-aware social networking application called UrbiFlock [GBLCS$^+$11]. Many of the contracts defined in the earlier chapter are directly influenced by their application in this framework.

Chapter 9. Conclusion gives an overview of the contributions made in this dissertation. This naturally leads to a discussion of the current limitations and possible directions for future work.

1.8 Summary

In this dissertation, we present a novel contract system specifically designed for the development of large scale ambient applications. We have focused our research efforts on such systems in order to facilitate the development of third generation ambient applications. Starting from the hardware characteristics of mobile devices we will deduce the desirable properties a contract system for ambient applications needs to have. From these properties we develop a contract system that supports the specification and verification of local behavioral, event-driven abstractions, and asynchronous messages passing. As we show in the related work no other contract system can deal with the challenges of an ambient environment. Moreover, our contract system provides the programmer with specialized reports that point out the parties to blame in case of a violation.

Chapter 2

Ambient-Oriented Programming

The main contribution of this dissertation is the definition of a contract system applicable for the development of smart ambient applications. In this chapter we give an overview of the ambient-oriented programming paradigm. The ambient-oriented programming paradigm forms the foundation on which the work presented in this dissertation has been built. We start this chapter with a short overview of the technologies that enable the development of ambient applications. Then we show the challenges where every ambient application has to deal with, independently of the programming language used. Starting from those general challenges we give an overview of the core ambient-oriented programming paradigm. This paradigm has been concretized in a programming language called *AmbientTalk*. This is also the main programming language that has been used to prototype the AmbientTalk/C contract framework. Once we have shown how to deal with the general challenges when writing ambient-oriented applications we show the specific challenges for the definition of an ambient contract system. The work described in this dissertation builds upon both the concepts behind ambient-oriented programming as well as its technical foundation laid by the AmbientTalk language. Therefore, it is crucial to explain both the paradigm and the language before presenting ambient contracts in the later chapters of this dissertation.

2.1 Middleware for Ambient-Oriented Programming

In this section, we give a bird's-eye view of current middleware solutions for the development of distributed and ambient applications. We argue that most middleware solutions are not ideal as the starting point for the development of an ambient contract system. Most middleware solutions either introduce a significant overhead for the programmer or are not flexible enough to express common ambient-oriented concepts. We argue that these limitations are not present in the AmbientTalk language which has been used as a research vehicle for the development of ambient contracts. The AmbientTalk language allows us to focus on the essential parts of developing a contract system for ambient applications while minimizing the overhead of dealing with low-level concerns.

To understand why we opted to use the AmbientTalk language, it is important to know that in software engineering a problem can be divided into its *accidental com-*

plexity and its *essential complexity* [Bro87]. Accidental complexity relates to the difficulties a programmer faces because of their choice of software engineering tools. This complexity can be decreased by selecting or developing better tools. On the other hand, essential complexity is caused by the characteristics of the problem to be solved. Such essential complexity is intrinsic and thus cannot be reduced. While the accidental complexity encountered when writing today's desktop applications is being addressed by the use of high-level programming languages such as Java or C#, we have not witnessed the same software engineering support for the development of ambient applications in mainstream programming languages.

Mainstream solutions for the development of distributed applications can be roughly divided into explicit communication abstractions and transparent communication abstractions. Explicit communication abstractions can be found in the use of sockets, or channels as adopted in the .Net languages. Such abstractions allow programmers to set up a communication link with other distributed components by means of an explicit communication address (i.e. IP address) or by making use of a centralized naming server (i.e. a DNS server). Unfortunately, as the exact communication partners of an ambient interaction are not known beforehand explicit addressing is rendered useless in an ambient context. Moreover, sockets are fragile and even common ambient-oriented operations like discovering other communication partners on the network requires a significant amount of complex code. Writing ambient applications in such a framework thus introduces a significant amount of *accidental* complexity that has to be handled by the programmer. A first prominent example of such accidental complexity can be found in the amount of exception handling code that is required to appropriately deal with temporary disconnections. A second difficulty for ambient programmers is to manually serialize and deserialize the interchanged values.

Implicit communication abstractions, on the other hand, try to hide the distributed nature of the application by making communication with remote components transparent. By far the most adopted technique for distribution transparency are systems based on the principle of remote procedure calls (RPC) such as Java RMI. A remote procedure call is similar to a normal procedure call but the arguments passed to a remote procedure call are marshaled and sent to a remote device. This remote device then executes a piece of code and sends back the result. All this is made completely transparent for the programmer. In the *myth of transparent distribution* [GF99], Gerraoui and Fayad point out that the illusion of distribution transparency is both impractical and dangerous. They point out that such an illusion introduces a wide range of problems that can only be solved by breaking distribution transparency. One example they describe is caused by latency. Distribution transparency can only be achieved when the processing time for distributed operations is similar to the processing time for local operations. Such a requirement is hard to amalgamate with the volatile nature of wireless connections.

In summary, when using low-level communication links to develop ambient-oriented applications the programmer is confronted with a lot of accidental complexly. On the other hand, systems trying to hide the communication in favor of distribution transparency have been shown to be impractical and dangerous. Our approach for programming ambient applications consists of language abstractions that address challenges that are directly derived from the hardware characteristics of mobile ad hoc networks. Programmers have to deal with these challenges independently of the ambient application they write. We built further on the ambient-oriented programming model that has focused on high-level language abstractions that tackle the accidental complexity (due to the hardware) of writing ambient applications so that the programmer can focus on the essential complexity. In particular it provides a set of abstractions that allows the programmer to reason at a high level about the distributed concepts of an ambient application. Infrastructure-less object discovery, buffered asynchronous communication and marshaling of objects are all primitive abstractions in an ambient-oriented programming language.

As the problems of dealing with the accidental complexity of mobile ad hoc networks are now well understood bringing the developed abstractions to a mainstream programming language would be a matter of manpower rather than brainpower. To avoid reinventing the wheel while formulating an ambient contract system we have chosen to make use of the AmbientTalk language which already embodies ambient-oriented abstractions. It is important though, to realize that the use of the AmbientTalk language as a research vehicle for the development of ambient contracts is a means to an end, not the end itself.

2.2 Wireless Network Applications

Advances in wireless technology have enabled the realization of a new brand of distributed applications in which mobile devices spontaneously interact with each other without the need of a centralized server. The network technology that enables such spontaneous interactions is called *mobile ad hoc networks*. Such mobile ad hoc networks in combination with the miniaturization of mobile devices are the key enablers for the development of ambient applications. Recent technological advances such as personal area networks (bluetooth), wireless LAN networks (Wi-Fi & Wi-Fi Direct) and digital short range communication (UWB) only confirm the need for specialized network infrastructure to connect distributed devices wirelessly.

While technological advances at the network layer are spectacular, a similar evolution at the software level is required to take full advantage of this new technology. The change from wired to wireless ad hoc communication has repercussions that percolate up to the level of the programming language [DVM$^+$05]. The hardware characteristics of mobile ad hoc networks render main stream programming languages unsuited for the development of ambient applications. For example, traditional distributed programming languages consider network disconnections as an exception rather than the rule. Therefore, the programmer has to constantly deal with intermitted network disconnections which leads to code that is polluted with if-tests and/or exception handling code.

In the following section, we give an overview of current wireless technology and show that the hardware characteristics of this mobile ad hoc networks must be handled at the software level.

2.2.1 Technological Advances

In the last years, wireless technology has been embedded in a wide range of devices and applications. Wireless technology is embedded in numerous mobile phones, music devices, e-readers, home automation etc. Not all these devices use the same wireless technology. For a particular application domain, certain wireless technology is better suited than other. For example, the requirements for wireless network technology of a mobile phone is very different form the technology needed for a smart fridge. While it is very likely that a user moves with his mobile phone it is rather unlikely that a smart fridge needs to communicate with the rest of the house while being moved. Therefore, the technology used for wireless phones differs from those used for home-automation. Many mobile phones today embed a multitude of wireless technology to better suit the needs of the user depending on what he/she is doing.

To give an intuitive feeling of the landscape of wireless technology we present an overview of the wireless communication technologies that are commercially available today[1]. An overview of mainstream wireless network technology is shown in figure 2.1. The y-axis shows the bandwidth while the x-axis indicates the mobility of the users in terms of connection speed. As can be expected, when the user is very mobile the bandwidth is limited (1 Mbits/s). When the user is rather stationary by making use of Ultra-wide-band technology (UWB) the data-rate can be as high as 100 Mbits/s. UWB technology was developed during the second world war and initially it has been mainly used by the US army. Nowadays, commercial applications include streaming of high definition video streams.

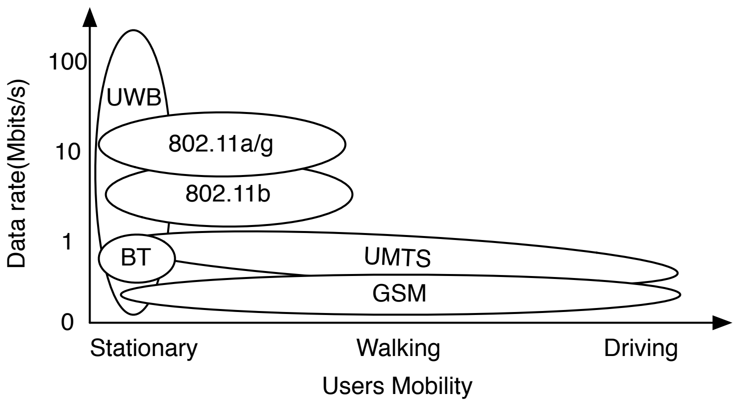


Figure 2.1: Comparison of wireless communication technology [BC06].

The use of the ether as communication medium brings a number of complications that are not present in wired communication technology. Examples of such complications include thermal noise, signal power attenuation (due to variations in the distance between transmitter and receiver), multi-path propagation, and interfering signals (multiple users using the same technology). Such channel characteristics are inherent to wireless communication and due to physical limitations will never change. Nevertheless the problems introduced by these characteristics can be minimized by using more advanced technology. For example, to minimize the problems of saturation, one promising approach is the use of *adaptive* radio technology [BC06]. Current mainstream wireless technology only saturates the ether at certain highly utilized frequencies. At the same time certain frequencies are used less frequently. To improve this situation and use all available frequency bands, adaptive radio technology adapts its communication frequency depending on the saturation level of the available frequencies.

While the utilization of the frequency band, the data rate, and the area coverage are expected to increase in the future, the inherit problems of wireless communication channels are not likely to be solved any time soon. Whenever a user communicates with another user over a wireless connection, at the application level, programmers have to deal with communication links that can disconnect at any moment in time. Therefore,

[1]December 2012

the challenges introduced by the hardware characteristics of mobile ad hoc networks must be handled at the software level. In the next section, we give a detailed overview of those challenges.

2.2.2 Integration at the Application Level

The hardware characteristics that influence the development of ambient applications at a software level have been identified and refined by a number of researchers including, Dedecker [Ded06], Van Cutsem [Van08] and Lombide Carreton [Lom11]. In the following sections we give a summary of the hardware characteristics and which challenges they induce. We mostly follow the definition as given by Van Cutsem [Van08].

Zero Infrastructure. Mobile ad hoc networks are formed by mere co-location of mobile devices without the need for any preexisting infrastructure [Ded06]. In contrast, WANS like internet employ routing devices and naming servers to make sure that services can be reached with a predetermined address (e.g. URL). In a mobile ad hoc network, devices can not know the location of any service beforehand. It is up to the devices in the network to autonomously determine their own routing, addressing, clustering and power control. Moreover, due to device mobility, the topology of the network is constantly fluctuating. At the application level, programmers can not assume that a service is available at any moment in time.

Volatile Connections. Mobile devices equipped with wireless communication technology only have a limited communication range. Because of the limited communication range and the mobility of the devices, the wireless connections between mobile devices can become disconnected at any moment in time. Such disconnection can be either permanent or transient. Quite often, a transient disconnection should not have a significant impact on the distributed interaction, and the application should simply continue whenever the connection is reestablished. At the application level, users expect their applications to be responsive and continue to work in the presence of transient failures. Dealing with failures due to volatile connections is not a new concept in distributed networks. However, the rate of disconnection to which mobile ad hoc networks are exposed is much higher than in traditional distributed systems. As such, the volatility of the connection should be considered the rule, rather than the exception.

Other challenges at the application level introduced by the hardware characteristics of mobile ad hoc networks are autonomy and concurrency. Dedecker [Ded06] considers these challenges to be at the same level as zero infrastructure and volatile connections. Van Cutsem [Van08] on the other hand, argues that they can be deduced from the two previous ones.

Autonomy Most distributed applications follow a client-server approach. In this approach a server has the responsibility to coordinate the clients which connect to it. However, in a mobile ad hoc network such a server might not be available. Therefore, every device should act autonomously. Van Cutsem argues that autonomy follows directly from the lack of shared infrastructure [Van08]: when the application programmer cannot rely on any shared infrastructure, applications necessarily need to be autonomous. We follow the view of Van Cutsem.

Concurrency Ad hoc networking applications have to deal with concurrency at a variety of levels. First, as Van Cutsem pointed out MANET applications are naturally concurrent, simply because they are deployed on a *distributed* network. Second, they have to be able to be responsive for the user while communicating with other clients in the network. Therefore, the programmer of an ad hoc network application is also confronted with the difficulties of writing a concurrent application.

2.2.3 Ambient-Oriented Programming

The hardware characteristics of mobile ad hoc networks affect the application components in such a profound way that they cannot be hidden from the application programmer [Van08]. The programmer is confronted with these effects by the way the application has to be structured, on how distributed application components interact and on how application data needs to be represented. As a result, software technologies for mobile ad hoc networks need to support application programmers by intrinsically offering support for the hardware characteristics of mobile ad hoc networks. In this section we give an overview of the three main software engineering principles as proposed by the ambient-oriented programming paradigm to address these hardware characteristics, as given by Van Cutsem [Van08].

Network Resilient Communication. The fact that there are disconnections should not have a high impact on the application components that use these connections, i.e. the application components should be resilient to frequent disconnections. This does not imply that the programmer should not be aware of changes in the network connectivity. The connections in a mobile ad hoc network signify a physical connection between colocation of mobile devices and whether there is a communicant link or not, can be important for the application logic. Therefore, while most of the application components should be ignorant for disconnections, there should still be abstractions available for the programmer to be notified of changes in the network connection.

Loosely-Coupled Application Components. Applications for mobile ad hoc networks can not assume any infrastructure. This implies that mobile ad hoc network applications cannot designate certain services to an a-priori set of known services because the presence of these services cannot be guaranteed. In order to deal with the uncertainty of the services that will be available at runtime, mobile ad hoc network applications need to be structured in a loosely-coupled way both at the architectural level and at the execution level. A loose coupling at the architectural level, means that the application should be composed in such a way that services can be spontaneously discovered without relying on any infrastructure. Moreover, the discovered services must be able to be discarded and replaced by other (equivalent) services at runtime. On the execution level, a loose coupling is needed to guarantee responsiveness. Applications that lose connectivity with some required service should not be blocked. All distributed application components in the mobile ad hoc network should run autonomously and only communicate using non-blocking communication primitives. Blocking and waiting for the reappearance of an unavailable service would halt the entire application.

Decentralized Discovery. Since mobile ad hoc network application cannot assume any infrastructure, the components themselves are responsible to announce which service they offer. Similarly, it is the responsibility of the clients to listen to these announcements and in response trigger the appropriate actions. In an infrastructure-less network setup, services have to be announced to interested parties by means of broadcasting advertisements.

In this section, we gave an overview of the hardware characteristics of mobile ad hoc networks and their implications on software engineering. This overview did not commit to any specific implementation technique or programming model. Nevertheless, there are general abstractions that a suitable programming model should have: communication abstractions that are resilient to volatile connections, loosely-coupled components, and decentralized discovery.

2.3 Ambient-Oriented Programming in AmbientTalk

The work on ambient contracts builds upon the principles of the ambient-oriented programming paradigm (described in section 2.2.3) and its technical realization: the AmbientTalk language. The AmbientTalk language has been the main research vehicle in which ambient contracts have been designed and implemented. Although our proposed abstractions could have been realized in other middleware, the use of AmbientTalk enables us to minimize the accidental complexity of dealing with volatile connections. In this section we give an overview of the most important AmbientTalk language features. We explain its distribution and communication model and give a short overview of its reflective capabilities. While this overview gives an introduction to the main principles of the AmbientTalk language we limit our explanation to those features of the language that are necessary in order to understand the code excerpts and technical contributions shown in the following chapters. A complete introduction to the AmbientTalk language can be found online [DGM+07]. In this section we provide an informal introduction to the AmbientTalk language, the operational semantics of a core subset of the AmbientTalk language is presented in chapter 4.

2.3.1 A Prototype-Based Language

The object model of AmbientTalk is inspired by prototype-based programming languages such as SELF [US87]. In AmbientTalk, objects are not instantiated from classes. Instead, they are created *ex-nihilo* or by cloning and adapting existing objects. In prototype-based languages, objects are reused and extended by means of *delegation* [Lie86] (also known as object-based inheritance).

An example of defining an object in AmbientTalk is shown in Figure 2.2. The object is created ex-nihilo and bound to the `Point` variable (line 1). The definition of this prototypical object contains a number of fields and methods that represent the object's state and behavior respectively. Cloning an object is done by sending the message **`new`** to an existing object. The receiving object creates a clone of itself and initializes the clone by invoking its `init` method. The `init` method thus has a similar purpose as the “constructor” methods known from class-based languages.

Message Sending and Inheritance. Messages are used both to invoke fields and methods of an object. AmbientTalk treats field access as a method invocation with zero arguments, also known as the *uniform access principle* [Mey00]. Looking up the field `x` in the `Point` object is syntactic sugar for `Point.x()`. Similarly, assigning the

```
def Point := object: {
  def x := 0;
  def y := 0;
  def init(aX,aY) {
    x := aX;
    y := aY;
  };
  def distance(p) {
    sqrt( square(x - p.x) +  square(y - p.y) )
  };
}

def Point2 = Point.new(4,0);
Point2.x;
Point2.distance(Point);
Point2.x := 5;
```

Figure 2.2: Object definition in AmbientTalk.

x field to the value 5 is syntactic sugar for `Point.x:=(5)`. Because `Point.x` applies the method x, different syntax is needed in order to select the method. In AmbientTalk the & operator can be used to select a method from an object, i.e. `Point.&x` selects the method x from the object `Point`.

An object that receives a message looks up the corresponding method and applies it. In case a message is sent to an object that does not understand this message, the message is *transitively* and *automatically* delegated to its parent object. Each AmbientTalk object has a special slot called **super** that represents the parent. The default value of **super** is bound to **nil**. When a message is sent to the value **nil** an exception is thrown. Important to note is that the **super** field of an object is a regular field. Therefore, the parent field of an object can be changed at runtime, this enables *dynamic inheritance* reminiscent to inheritance in Self [US87]. Creating a new object that extends an existing one can be done at once by making use of the **extend:with:** function. Figure 2.3 shows an example of extending the `Point` object defined in Figure 2.2.

Because the **super** field is a regular field, invoking a method directly on the **super** field is not what is traditionally understood by a super send. When invoking methods on the super field the **self** variable is not bound to the sender of the message but as regularly to the receiver of the message. To explicitly delegate a message to another object AmbientTalk supports the delegate (^) operator. When sending a message using the delegate operator, the **self** variable is left bound to the sender of the message.

```
def Circle := extend: Point with: {
  def r := 0;
  def init(r,x,y) {
    super^init(x,y);
  }
};
```

Figure 2.3: Extending an object in AmbientTalk.

Scoping. Variables in AmbientTalk are looked up in the lexical scope. When they are not found in the lexical scope they are looked up in the delegation chain of the object in which the variable was refereed. The programmer can explicitly choose to ignore the lexical scope of a variable by qualifying the object in which the variable needs to be looked up, i.e. **self**.x.

Block Closures. Anonymous functions or blocks are extensively used in AmbientTalk i.e. to implement delayed computations in the branches of an if-then-else expression. Figure 2.4 shows an example of a basic block to sum two numbers. Blocks are created using curly braces and the expected arguments are listed between bars (|). In the example the block requires two numbers which are bound to the variables a and b when the closure is applied. If the block does not require any argument, the |<varlist>| can be omitted.

```
>{|a, b| a+b }(3,2)
>>5
```

Figure 2.4: Defining and applying a block.

Keyworded messages. AmbientTalk supports keyworded messages as in Smalltalk [GR89] and Self [US87]. In Figure 2.5, the definition of the keyworded function map:onto: is shown. This function takes a closure and a table as arguments and applies the closure to each element of the table. In the example, we explicitly create a block that increments its argument. This closure is passed to the map:onto: function together with the table [1,2,3]. As expected, the result of this function application is [2,3,4].

```
>def map: clo onto: tbl {
  ...
}
>> <closure:map:onto:>
>map: {|x|x+1} onto: [1,2,3]
>>[2,3,4]
```

Figure 2.5: Keyworded function definition in AmbientTalk.

Type Tags. All objects in AmbientTalk have a set of type tags. A type tag categorizes the object to which it is attached in a certain group. A type tag can be best compared with an *empty* Java interface. All primitive objects such as numbers, strings and tables have type tags to indicate to which category they belong. The programmer can create new type tags with the keyword **deftype**. For example, **deftype** Circle; creates a new Circle type tag that can be used in order to tag new objects. Tagging an object is done with the **object**:**taggedAs**: construct. Once an object is created it is impossible to change the tags of the object, i.e. the type tags of an object are immutable. The most important use of type tags is to define in which way objects are passed to remote objects as shown in section 2.3.3.

2.3.2 A Concurrent Language

The AmbientTalk's concurrency model is based on the model of the E language's communicating event loops [Mil06], which is itself an adaptation of the well-known actor model [Agh86]. In this model, an actor is a container of regular objects encapsulating a single thread of execution called an *event loop*. Throughout this text, the terms *event loop* and *actor* are used interchangeably and always refer to the "actors" as defined in the communicating event loop model. An event loop perpetually takes a message from its message queue and invokes the corresponding method of the object denoted as the receiver of the message. The method is then run to completion denoting a *turn*. A turn is executed atomically, i.e. an actor cannot be suspended or blocked while processing a message. Messages are processed serially and to completion to avoid low-level race conditions on the mutable state of the contained objects.

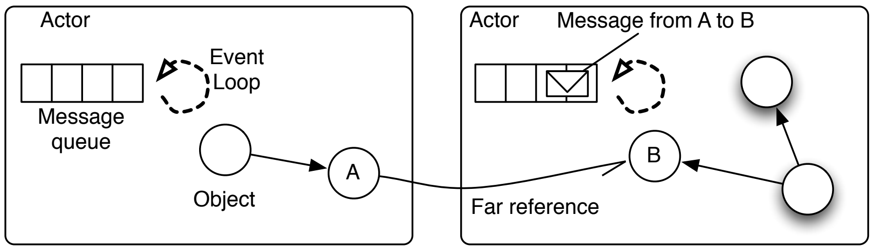


Figure 2.6: Communicating event loop model.

Figure 2.6 illustrates actors as communicating event loops. In the AmbientTalk actor model, there is no sharing of objects between actors and each object is owned by exactly one actor. The event loop (represented by dashed lines) processes incoming messages one by one and synchronously executes the corresponding methods on the actor's owned objects. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor may communicate using standard, sequential message passing or using asynchronous message passing. It is possible for objects owned by one actor to refer directly to individual objects owned by another actor. Such references that span different actors are named *far references*. The terminology stems from E [Mil06]. Far references only allow asynchronous access to the remote object. Any messages sent via a far reference to an object are enqueued in the message queue of the owner of the object and processed by the owner itself. For example, when A sends a message to B, the message is enqueued in B's message queue, which eventually processes it. A turn consists of the execution of a number of synchronous method invocations and asynchronous message sends. The method invocation stack is empty both at the start and at the end of a turn.

Message Passing. There are two types of messages that can be sent to an object, synchronous or asynchronous. Synchronous messages can only be sent to objects which reside on the same actor. Communication with an object in another actor happens asynchronously by means of far references: object references that span different actors. In AmbientTalk asynchronous method invocations are indicated by the arrow operator $\leftarrow$. An asynchronous message is enqueued in the receivers' message queue, which eventually processes it. Important to note is that AmbientTalk guarantees message ordering on the outgoing messages, i.e. messages sent from one actor to another actor are received in the same order as they are sent.

When a far reference to a remote object `A` is passed in a message a new far reference is created at the receiver side which points directly to the object `A`. When a far reference pointing to an object `B` is sent to the actor that owns the object `B` it is resolved back to a local reference.

By default, the objects in AmbientTalk are passed by far reference. In order to pass objects by copy, they have to be created with the **isolate:** construct instead of the **object:** construct. The name isolate comes from the fact that isolates do not have a surrounding lexical scope i.e. they are isolated pieces of code. Lexically visible variables have to be manually copied into the isolate's scope. When an objects tagged as an **Isolate** is sent in an asynchronous message a copy of the object is created and all the objects that are referenced by the isolate are also copied according to their own type tags. Primitive types such as strings and numbers are all tagged as **Isolate**.

Future-type Message Passing. Synchronous and asynchronous messages are not only different in how arguments are passed, they also differ in how the result of processing the message is obtained. Sending a synchronous message evaluates to the result of processing the message immediately. The result of sending an asynchronous message is a future [YBS86]. A future acts as the placeholder for the return value of processing an asynchronous message. Special about the futures in AmbientTalk is that retrieving the result where the future resolves to is also a non-blocking operation.

Figure 2.7 shows a concrete example of using future-type message passing. First, an asynchronous message `getName` is sent to a remote messenger application. The result of this message is a future which is bound to the variable `fut`. Retrieving the value where the futures resolves to is done by registering a callback function with the **when:becomes:** construct. This construct expects a future and a block that is registered to the future as an observer. When the future resolves to a value the registered callback is applied to the resolved value. In the example, the future is expected to resolve to a string representing the name of the remote messenger application, which is printed when the future resolves (line 5). The **when:becomes:** construct returns a future on its own which is resolved to the return value of evaluating the registered callback function.

```
1  ...
2  def fut := messenger<-getName();
3
4  when: fut becomes: { |name|
5    system.println("Buddy Name = " + name);
6  };
```

Figure 2.7: Future type message passing in AmbientTalk.

As will be shown in the operational semantics of the AmbientTalk (Chapter 4), futures in AmbientTalk allow asynchronous messages to be sent to the return value of a remote computation even when it is not computed yet. Messages sent to an "unresolved" future are stored in the queue of the future. When the future resolves, the queue of messages is forwarded to the resolved value.

2.3.3 A Distributed Language

Now that we have shown the object-oriented and concurrent language constructs of AmbientTalk, we can show the distributed language constructs. We start by showing how objects can be published in the network and how a remote client can discover objects without having to rely on a centralized server. We then show how the remote object references can be used to communicate values over the network.

Publishing Objects. AmbientTalk provides language support to make an object available to other objects that reside in remote actors by means of the **`export:as:`** construct. The **`export:as:`** construct takes two arguments: the object that is made remotely accessible and a service type under which the object can be discovered. For example, in a distributed chat application the messenger service of one node in the network is made accessible to the other nodes as shown in Figure 2.8.

```
deftype InstantMessenger;
def messenger := object: {
  getName() { ... };
  ...
};
export: messenger as: InstantMessenger;
```

Figure 2.8: Publishing objects in the Ambient for remote discovery.

When an object is exported by its actor, it becomes discoverable by other actors by means of its service type. In this example the `messenger` object is made accessible to other nodes in the proximity by means of the `InstantMessenger` type tag. Internally, the object is placed in the export table of the actor. As shown in the example, a service type is represented by a type tag. Services types denote an abstract publication topic. The **`export:as:`** construct returns a publication object that responds to a `cancel` method that can be used to cancel the publication, i.e. unexport the object.

Registering for Object Discovery. AmbientTalk has a decentralized object discovery mechanism. When exporting an object it is advertised in the network by means of its service type as specified in the **`export`** construct. AmbientTalk allows the programmer to register an observer that is triggered when a remote object of a certain service type becomes available in the network. For example, one can discover an object exported with the `InstantMessenger` service type by means of the **`when:discovered:`** construct, as shown in Figure 2.9. Similar to **`export:as:`** it returns a subscription object that responds to a cancel method that can be used to cancel the subscription so that the callback is no longer invoked.

```
when: InstantMessenger discovered: { |messenger|
  when: (messenger<-getName()) becomes: { |name|
    buddyList.put(name, messenger);
    system.println("Added buddy: " + name);
  };
};
```

Figure 2.9: Object discovery in AmbientTalk.

The callback to execute when the service type becomes available receives as parameter the actual remote reference to the discovered service object. In Figure 2.9, `messenger` is a remote reference to the remote object exporting the `InstantMessenger` service type (defined in figure 2.8).

Imagine the interaction between the instant messenger applications executing on the mobile devices of two persons, e.g. Chris and Lisa. When Chris' instant messenger and Lisa's instant messenger come into one another's communication range, Chris will discover Lisa and vice versa since both are exporting and discovering the `InstantMessenger` service type. Both obtain a far reference, which is bound to the variable `messenger`, over which only asynchronous messages can be sent. The **`when`** **`:discovered:`** callbacks are triggered only once when an `InstantMessenger` service type becomes available in the network. To be able to discover all other instant messenger buddies available in the network, the **`whenever:discovered:`** construct can be used. The block of code specified in **`whenever:discovered`** can be fired multiple times upon discovering several exported objects. Note that objects exported by an actor do not trigger the actor's own **`when:discovered:`** nor **`whenever:discovered:`** observers.

Network Failures. The communication model of AmbientTalk makes intermittent disconnections transparent for the programmer. Whenever an asynchronous message is sent to a disconnected far reference the message is buffered and sent when the connection is reestablished. However, in certain situations it is important to make the user aware of a disconnection. For example, in the messenger application the GUI could gray out disconnected communication partners. Figure 2.10 shows how this functionality can be encoded in AmbientTalk. In order to be notified of the state of a far reference the programmer can register observers to be notified of disconnections and reconnections. The two constructs **`whenever: disconnected:`** and **`whenever:`** `messenger` **`reconnected:`** both expect a far-reference and a block which is applied when the far reference disconnects or reconnects respectively.

```
whenever: messenger disconnected: {
  //update the gui
}

whenever: messenger reconnected: {
  //update the gui
}
```

Figure 2.10: Dealing with disconnections in AmbientTalk.

2.4 An Early Preview of AmbientTalk/C

In this section, we scratch the surface of our ambient contract system called AmbientTalk/C. We start by showing the general validation strategy used for the validation of ambient contracts. Subsequently, we show the expressiveness of our contract system by how ambient contracts can validate functions, methods and object in the presence of ambient-oriented language abstractions. Ambient contracts in this section are explained from an end programmer's perspective. The overview in this section is not detailed and only serve as a sneak preview of the kind of contracts explored in this dissertation.

2.4.1 Ambient Contracts Validation Strategy

As mentioned in the introduction chapter, ambient contracts are validated at runtime because static techniques to verify the properties of an ambient contract system would either require: extensive annotations, a lot of computation or both. Contracts in AmbientTalk/C validate the operation of runtime values such as objects, functions, far references, futures etc. In contrast to early contract systems such as adopted in Eiffel [Mey91] AmbientTalk/C has support for complex first-class values. The validation strategy to validate complex first-class values is adopted from the research on contracts for higher-order functions [FF02]. In the following sections we exemplify why the validation of higher-order functions is more complex and explain the concept of blame assignment.

Runtime Errors and Contract Systems. Many times when an error is detected at runtime, the point in the program where the error is detected (e.g. line number) is not related to the point in the program where the problem is caused. Contract systems help the programmer in relating the point where the error is detected to its cause. An example of an error that is difficult to relate to its cause is shown in figure 2.11. Each code snippet shows two pieces of code, the top part of the examples shows pseudo code for a supplier module, which exposes two functions: `set` and `increment`. The bottom part of the examples shows pseudo code for the clients of the supplier modules. When a client applies the `set` function the supplier stores this value in a variable `x`. When the client applies the `increment` method the value stored in the variable `x` is assigned to its value incremented by one.

```
 1  //suplier
 2  def x := 0;
 3
 4  def set(y) { x := y; };
 5  def increment()  {
 6    x := x + 1; ← ERROR
 7  };
 8
 9  //client
10  set("Wrong"); ← CAUSE
11  increment();
```

```
 1  //suplier
 2  def x := 0;
 3  //int -> int
 4  def set(y) { x := y; };
 5  def increment()  {
 6    x := x + 1;
 7  };
 8
 9  //client
10  set("Wrong"); ← ERROR,CAUSE
11  get();
```

Figure 2.11: Errors signaling without contracts (left) and with Contracts (right)

On the left side of figure 2.11, the client code first applies the `set` function provided by the server to the value `"wrong"`. The server stores the value `"wrong"` in the variable `x`. Subsequently, the client applies the function `increment`. The server then tries to increment the value stored in the variable `x` which at that moment in time is still the value `"wrong"` (line 6). In many languages this program crashes with an error message informing the programmer that strings can not be added to numbers. Note that the error is inside of the body of the function `increment` (line 6) while the responsibility actually lies at the caller of the function `set` (line 10). Finding such an error in a large program is not trivial because the programmer needs to verify all paths in the code through which the value `x` can be assigned.

Using a contract system significantly improves the programmer's understanding of where the error was caused. Figure 2.11 on the right shows exactly the same code as shown on the left but now a contract (int $\rightarrow$ int) over the function `set` is defined. This contract validates that the argument over the function `set` is an integer and that the result is an integer. The contract system uses this information and can detect the problem when the `set` function is applied. Therefore, the place in the code where the error is signaled and the cause of the problem are now synchronized. A contract system thus helps the programmer in relating the point where the error is detected and which part of the software is responsible for the error.

The kind of contracts shown in figure 2.11 define trivial properties, i.e. is the value an integer ?. Such contracts are called flat contracts [FF02]. Note that, Rice's theorem has shown that it is undecidable whether an algorithm implements a partial function with a non-trivial property [Ric53]. Therefore, contract systems can not deal with higher-order functions in the same way as with data. Contracts that can also deal with higher-order functions and objects are called *higher-order contracts*.

Higher-Order Contracts. Higher-order contracts deal with the validation of complex first-class values such as functions by postponing the validation of the contracted functions until they are `used`. When an error occurs during the execution of the program the cause of an error is no longer directly related to where the error is detected. At first sight this reduces the usability of a contract system significantly, as again the cause of the error and where the error manifests itself are drifting apart. However, by making use of a technique called *blame assignment* the contract system can still point out the responsible of the violation.

Consider the example shown in figure 2.12. The contract defined over the function `set` is now changed to indicate that `set` should be applied to a function. This function should be applied to a number and should return a number. The server simply assigns this function to the variable y. The second function of the server is the function `getResult`. This function applies the function stored in the variable `x` to the number 42.

```
//supplier
def x;
//(int -> int) -> int
def set(y) { x := y; };
def getResult()  {
  x(42); ← ERROR
};

//client
set({|x| "Wrong"}); ← BLAME, CAUSE
getResult();
```

Figure 2.12: Contracts in a system with first-class functions.

The client shown at the bottom of figure 2.12 first applies the function `set` with a function that always returns the string `"wrong"`. Subsequently, the client applies the function `getResult` and a violation is detected. What a contract system for higher-order functions does is to keep track of the responsibilities of the server and the client in order to point out the responsible when a contract violation is detected. In the example this means that the contract system can assign blame to the client module when the variable x is applied to the value 42.

Summary. Ambient contracts are validated at runtime and allow the specification of contracts over higher-order functions, objects, far references, and futures. For all these values the contract system points out the responsible party when an error is detected, i.e. the party to blame. As we show in the following sections the introduction of such values introduces a number of problems that were not studied before.

2.4.2 Behavioral Validation: Computational Contracts

Ambient contracts allow the programmer to express that certain events should or should not happen during the execution of the contracted value. We call such contracts *compu-*

tational contracts. In order to give an intuitive feeling of a computational contract, consider the widely known observer pattern [GHJV95]. In this pattern, observers can be registered to a subject which notifies the observers of interesting events. Imagine a subject called `eventSource` with an `addMeasurement` method. When the `addMeasurement` method is invoked the `eventSource` should notify all registered observers. With computational contracts the programmer can ensure that the `notifyAll` method is invoked during the execution of the `addMeasurement`. The way to specify such a contract is shown in figure 2.13. In this figure, there are two object contracts defined with the **`ObjectContract`** construct. An object contract is defined similarly to how an object is defined. The methods of an object contract define the interface that the contracted values should support. The bodies of these methods define the contracts that should be satisfied by the methods of the contracted object.

```
1  def measurment := ObjectContract: {
2    def getMeasurement() { void -> int  }
3  };
4
5  def eventSource := ObjectContract: {
6    ...
7    def addMeasurement(m) {
8      int -ensure_c(notifyAll)-> void;
9    };
10 };
11
12 def moduleInterface := object: {
13   def contractedSource := export: source withContract: Ref(eventSource);
14 };
```

Figure 2.13: A computational contract example.

The first object contract `measurement` validates whether an object has a `getMeasurement` method. The contract (`void -> int`) defined over this methods specifies that the method does not take any arguments and must return an integer value.

The second object contract is called `eventSource`. The contract over the method `addMeasurement`, (`int -ensure_c(notifyAll)-> nothing`), specifies that the method must be applied to a integer value and should not return any value. The computational contract is defined over the arrow (`ensure_c(notifyAll)`). It specifies that the `notifyAll` method should be applied during the execution of the method over which it is defined, i.e. `addMeasurement`. When the `addMeasurement` method does not apply the `notifyAll` method blame is assigned.

In AmbientTalk, the interface of a module is the latest object defined in the module. All the fields of that object are accessible to modules that import the module. In our example, this interface is shown from lines 12 till 14. Contracts are deployed over objects when they are exported by the module. For example, the code to export an object called `source` is shown at line 13. Note that the object contract is instantiated with the function `Ref` in order to indicate that this is a *local* object. Later another instantiation function `FarRef` is shown to indicate that the contract is defined over a far reference.

As we will show in chapter 5, computational contracts go beyond single invocations, they also allow the validation of a sequence of function applications and object protocols. For example, an object protocol over a file object can specify that the `write` method can only be applied after an invocation of the `open` method.

Summary Ambient contracts allow the validation of internal behavior of the contracted entities. This is important in the light of ambient-oriented applications where messages are being sent as the effect of receiving messages. Ambient contracts support contracts that can validate these messages by means a novel contract system called *computational contracts*. These contracts are the main subject of chapter 5.

2.4.3 Validation of Event-Driven Applications

To minimize the effects of volatile connections on the program code all communication with remote entities is asynchronous. The result of sending an asynchronous message is a *non-blocking* future. The AmbientTalk/C framework allows the programmer to define contracts over where such futures should resolve to. Such contracts are created with a contract constructor `Fut`. For example the contract `Fut[nat]` creates a new contract that can validate whether or not a future resolves to an natural number. We will show in chapter 6 that future values invert the responsibilities of the supplier and the client of the contract. This inverted responsibility also inverts the blame assignment strategy.

As an example consider again the observer pattern shown in the previous section but instead of receiving a measurement object directly, the `eventSource` receives a future that should resolve to a measurement object. The code for this example in AmbientTalk/C is shown in figure 2.14. This example combines object contracts, future contracts and computational contracts. The contract on line 4, specifies that the `addMeasurement` method should receive a future that resolves to a value that satisfies the `measurement` contract. The computational contract on line 4, again validates that the `notifyAll` method is applied during the execution of the `addMeasurement` and finally it specifies that the method should not return anything.

```
1  def eventSource := ObjectContract: {
2    ...
3    def addMeasurement(m) {
4      fut[measurment] -ensure_c(notifyAll)-> nothing;
5    };
6  };
```

Figure 2.14: Supplier Module

Summary The use of futures and their callbacks is omnipresent in the development of ambient applications. In the AmbientTalk/C framework we have provided abstractions specifically for defining contracts over futures and their callbacks. These contracts are explained in chapter 6.

2.4.4 Validation of Distributed Applications

As previously explained, AmbientTalk is a distributed language that employe a non-blocking communication model.

Figure 2.15 depicts two communicating actors. The server actor hosts an object `A`, the client actor has a far-reference to this object `A`. When we refer to the *server-side* of a distributed ambient contract, we refer to the module that hosts the object under contract. When referring to the *client-side* of a distributed ambient contract, we refer to that module which receives a far reference i.e. a reference to an object hosted by another actor. Note that the server-side and client-side view depends on the object over which the contract is defined. It is perfectly possible for an actor to be a server for one object and a client of another object.

In an ambient application programmers should be able to define contracts both at the server or at the client-side. From the server's perspective, the idea is to protect the incoming messages to the exposed objects. From the client's perspective, the idea it to protect the themselves from receiving misbehaving far references.

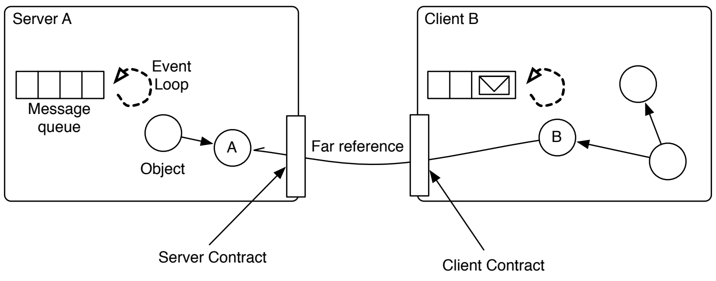


Figure 2.15: Server side and client side contracts over objects.

Object contracts can be defined over local objects (server side) with `Ref` as shown in section 2.4.2, but they can also be used to specify contracts over far references with `FarRef` (client side). As an example of a distributed ambient contract, consider the instant messenger application previously explained in section 2.3.3. Figure 2.16 shows two object contracts called `user` and `chat`. The `user` contract specifies that the user should should have a method `getName` that does not take any arguments and returns a `string`. The `chat` contract specifies that the instant messenger understands a `message` method that receives a *far reference* that satisfies the `user` contract and a value satisfying the `string` contract.

Assume that there is a `messenger` object that satisfies the `chat` contract. This object is exported a line 13 so that clients in the network can discover this object. The actor shown in figure 2.16 is the server of the `chatApplication` object. Therefore, exporting the object to the network is performed with a `Ref` contract constructor. When the `chatApplication` receives a message from a remote actor, the actor shown in figure 2.16 is the *client* of the `remoteUser`. Therefore, the incoming message is constructed with `farRef`.

```
 1  def user := ObjectContract: {
 2    def getName() {
 3      void -> string;
 4    };
 5  };
 6
 7  def chat := ObjectContract: {
 8    def message(remoteUser, message) {
 9      farRef(user) * string -> nothing;
10    };
11  };
12  ...
13  export: messenger as: InstantMessenger withContract: Ref(chat);
14  ....
```

Figure 2.16: Distributed ambient contracts

Summary The AmbientTalk/C framework supports both server-side and client-side (distributed) contracts. Server-side contracts are the subject of chapter 6 while client-side contracts are discussed in chapter 7.

2.4.5 Ambient Contracts Overview

In this section, we give an overview of the design space of the AmbientTalk/C contracts system. For a long time, contracts were limited to flat contracts which can be expressed as simple predicates. With the introduction of higher-order contracts [FF02] the design space became bigger. It was no longer necessary to limit contracts to be simple predicates. Therefore, contracts could also be defined over higher-order functions and objects. Ambient contracts apply higher-order contracts to distributed ambient-oriented programming and introduce two other dimensions. A first dimension allows contract systems to capture meaningful properties over the execution of a contracted entity. The second dimension is the exploration of contracts over remote object references.

The design space explored for the definition of the AmbientTalk/C framework is shown in figure 2.17. The figure summarizes the 3 dimensions of an ambient contract system: the distribution, the contract type and the value over which the contract is defined. For the values dimension we consider, data, functions, objects and future values. We consider non-blocking futures apart from objects because they require a different validation strategy as we show in chapter 6. For distribution we consider whether the contract is a server-side contract or a client-side contract. Finally, we consider flat, higher-order and computational contracts as the different kind of contracts.

The sum of all these variations shown in figure 2.17 is what we called `ambient contracts`. While the main contribution of this dissertation are those contracts which were not explored before, ambient contracts also allow the programmer to encode contracts that were already studied before. As such, ambient contracts subsume a number of previously studied contract systems.

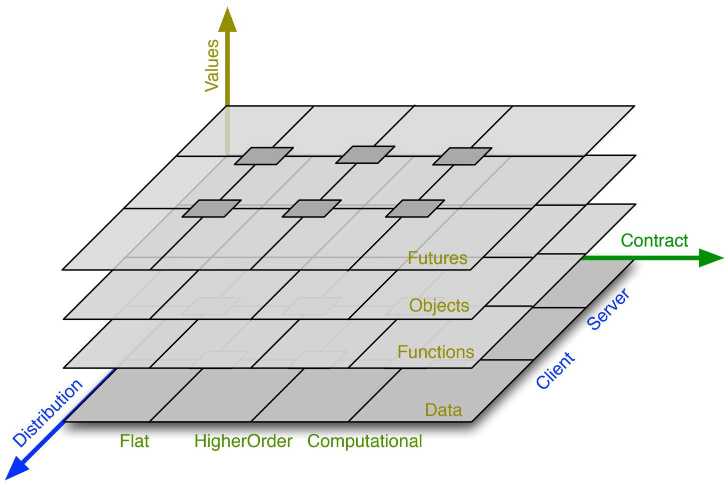


Figure 2.17: The ambient contracts design space.

In the rest of this dissertation, we explore each of the variations shown in figure 2.17. In chapter 5 we focus on computational contracts over functions. Chapter 6 shows object contracts at the server-side and the exploration of non-blocking future contracts. In chapter 7 we give an overview of client-side contracts on far references.

2.5 Conclusion

In this chapter, we have given an overview of the ambient-oriented programming paradigm. We have shown the basic requirements that a programming language should support to be applicable for the development of ambient applications. Subsequently, we have shown an overview of the AmbientTalk language, an ambient-oriented programming language that adheres to the requirements distilled from the hardware characteristics of mobile ad hoc networks.

The ambient-oriented programming paradigm and its concrete instantiation in the AmbientTalk language form the foundations on which we have developed our ambient contract framework called AmbientTalk/C. We have shown the design decisions taken for the development of the AmbientTalk/C framework and briefly introduced the design space explored. In the next chapter, we discuss related work and show that many of existing contract systems, have support for some of the variations in our design space. However, for other variations in this grid no previous work was conducted. The contribution of the AmbientTalk/C framework lies in the development of a contract system that supports all the variations necessary to operate in an ambient environment.

Chapter 3

Related Work: Software Verification Techniques

The goal of this dissertation is to define a contract system which is applicable for ambient-oriented programming languages. This chapter starts with an overview of existing systems for design-by-contract. We categorize contracts based on the kind of properties that are stated over the exchanged values. Once the different categories of contracts are shown, we zoom in on the challenges of ambient contracts and present the focus of our work with respect to the different categories of contracts system. With this focus in mind, an overview of related contract systems is given. The related work is presented by giving a description of the advantages and disadvantages of each existing system with respect to monitoring an ambient system. Finally, we conclude the related work with a short summary about which elements of the related work provide us with the best starting point for the definition of ambient contracts.

3.1 Categorization of Software Contracts

Software contracts can be applied for verifying different aspects of the execution of a program. In this dissertation, we follow the categorizations introduced by Beugnard et al [BJPW99]. An overview of the categorization is shown in figure 3.1. There are basically four levels at which contracts can be applied with increasing negotiable properties. At the lower level, contracts are not flexible and consist of the basic requirements that make a program work, i.e. the type signature of functions. At the highest level, non-functional requirements such as the quality of return values of functions are incorporated. The grey zone in between these extremes consists of behavioral and synchronization contracts which in certain situations maybe negotiable or not. A detailed description of these categorizations is given in the remainder of this section.

Syntactic Contracts (or basic contracts). The goal of a syntactic contract is simply to help ensuring that a program works. One possibility of specifying syntactic contracts is by means of an Interface Definition Language (IDLs). In typed programming languages syntactic contracts are defined by the specification of types. Syntactic contracts are not negotiable as failing to adhere to the contract results into a system that does not work.

Figure 3.2 shows an example of a syntactic contract in the form of an interface in a typed language. The example shows the interface for a specific stack where the

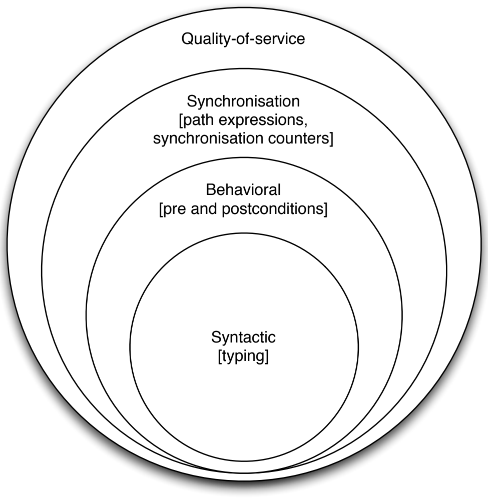


Figure 3.1: Overview of the categorizations of contracts by Beugnard et al [BJPW99]

elements that can be pushed must be integer values. This is indicated by the type declaration of the `push` method. Similarly, the type declaration of the `pop` method indicates that its return value must be an integer. The interface also specifies that any `IntStack` must implement a `clear` method. Depending on the expressiveness of the type system these contracts can be verified statically.

```
public interface IntStack {
 void push(int x);
 int pop();
 void clear();
}
```

Figure 3.2: Example: Interface Contract in a Java Like Language.

Behavioral Contracts. Unfortunately, syntactic contracts do not say anything about the effect of executing an operation. The `pop` operation of the stack example will probably return an integer value, but the syntactic contract does not specify what happens in case that the stack is empty. Influenced by abstract data type theory, Beugnard et al. defined behavioral contracts by using boolean predicates over the provided operations [BJPW99]. Such boolean predicates are executed before and after the operations are executed and, hence, are called pre and post-conditions. The goal of such pre and post-conditions is to capture the effect of executing each operation. While syntactic contracts ensure compatibility at a syntactic level, behavioral contracts also try to capture semantic compatibility.

An example of a behavioral contract over the `IntStack` is shown in figure 3.3. Pre-conditions are indicated by the keyword `Require` and specify requirements for the client. For example, the pre-condition of the pop operation specifies that the client can only call the pop operations in case the size of the stack is bigger than zero. Post-conditions are indicated by the keyword `Ensure` and specify requirements for the supplier. The post-condition of the pop operation specifies that the size of the stack after executing the pop operation should be the old value (indicated with `@pre`) decreased by one.

```
public interface IntStack {
 void push(int x){
  Ensure size() = size()@pre + 1;
 };
 int pop() {
  Require size() > 0;
  Ensure size() = size()@pre - 1;
 };
 void clear() {
  Ensure size() = 0;
 };
}
```

Figure 3.3: Example: Behavioral Contract in a Typed Language.

Synchronization Contracts. The goal of a synchronization contract is to specify the coordination of the exposed operations. Often such synchronization contracts can be expressed by a labeled transition system which ensures that the operations are used in the proper order. Some systems define a domain-specific language in order to specify such constraints in the form of path expressions. In systems that do not have direct support for synchronization contracts, the programmer often has to fall back to low-level synchronization techniques like locks and semaphores.

An example of a synchronization contract for the stack example is shown in figure 3.4. The goal of this contract is to make sure that threads access the stack are mutually exclusive. This is achieved by defining all the methods of the `IntStack` to be synchronized.

```
public interface IntStack {
 void synchronized push(int x);
 int synchronized pop();
 void synchronized clear();
}
```

Figure 3.4: Example: Synchronization Contract in a Java like Language.

Quality of Service Contracts. The goal of a quality of service contract is to validate certain qualitative aspects of the operations. Examples of such quality of service contracts may include the duration of the operation, leasing time, precision of the reply, the resolution of a video stream etc. Unlike the other levels the quality of the service is subject to negotiation in many situations.

3.1.1 Design considerations for AmbientTalk/C

In this dissertation, we focus on the formulation of a contract system to validate the behavioral and synchronization aspects of an ambient application.

Syntactic Contracts. The focus of this dissertation is to define a contract system for a dynamically typed language. In particular, in this dissertation ambient contracts are investigated in AmbientTalk. The AmbientTalk language does not have a static type-system for the verification of ambient-oriented programs. However, many advanced static verification techniques have been proposed in order to verify the behavioral or synchronization properties of distributed applications. In the following sections, we highlight static verification techniques, show their advantages and show why it would be difficult to incorporate them in an ambient-oriented programming language.

Behavioral Contracts. The core of the ambient contracts system presented in this dissertation is focused on the behavioral verification of ambient applications. Ambient contracts provides the programmer with pre and post-conditions that allow the programmer to capture the behavior of the ambient application in a contract. When programming ambient applications programmers make extensive use of language constructs such as non-blocking futures and callbacks. The combination of these properties into a single contract system introduces problems that have not been investigated in the related work.

Synchronization. As explained in section 2.3 the core of the ambient-oriented paradigm is based on the well known actor model. At the heart of this model lies a synchronization model that helps the programmer to deal with certain problems introduced by concurrency. One of the major design decisions for synchronization when ABCL [Yon90] proposed active objects as an extension to the actor model was:

> "*One at a time: An object always performs a single sequence of actions in response to a single acceptable message. It does not execute more than one sequence of actions at the same time.*"

Since then sequentiality of processing messages, has been one of the main rules in stateful actor languages such as Erlang [AVWW96], Akka and Scala [HO07], Kilim [SM08], ProActive [BBC$^+$06], E [Sti04], Salsa [VA01] and AmbientTalk [DVM$^+$05]. In all these languages execution of parallel messages within a single actor is disallowed by construction, e.g., every actor has only one thread of control and data cannot be shared between actors. Therefore, the synchronization over individual objects within an actor is dealt with by the underlying paradigm. Besides the concurrent access of one specific object there is also synchronization needed to manage the order of individual messages sent by one actor to another actor. Such synchronization properties are not easy to encode in an event-loop concurrency model as adopted in AmbientTalk. Therefore, the focus of ambient contract with respect to synchronization is on the order of the outgoing and incoming messages. For other synchronization issues such as deadlocks and low-level race conditions, we rely on the underlying concurrency model of AmbientTalk that precludes such issues by design.

Quality of Service The quality of service of the different components of an ambient applications has been studied in the context of ambient systems [ZBS97, FK99, LN99]. The focus of most of those systems is to guarantee a certain quality of service according to the fluctuations in the context of the application. As shown in the introduction the research on context-oriented programming for ambient systems is still under investigation and contract systems to validate the quality of service would certainly be a valuable asset. However, most of the quality of service contracts are defined for systems that assume that mobile devices are very slow and have very limited resources. In the design and implementation of the ambient-oriented paradigm, none of those characteristics are taken into account because they are likely to be solved by technological advances. The focus of ambient contracts is not on the negotiable properties nor on how to adapt the applications according to the context. Nevertheless, we also survey the more interesting frameworks for quality of service validation.

3.1.2 Conclusion

In this section, we have shown an overview of the categorizations of design by contract systems. We have shown that contracts can be categorized in four levels depending on their functionality and negotiability. Subsequently we evaluated these categories and showed on which of these categories ambient contracts have focused. Contracts have proven to be applicable in a wide range of software engineering concepts such as components, testing and debugging of distributed real-time systems and even high performance computing. In the following section an overview of the most related work is given.

3.2 Survey of Related Work

In this section, we give an in depth overview of systems (contracts, session types, interaction protocols, process calculi, etc.) with the common goal to verify or monitor the behavior of a program. The overview is not limited to systems that have been designed to operate in an ambient environment. We broaden our scope because many of the interesting design decisions that constitute ambient contracts are adopted from contract systems that were not specifically designed for an ambient environment. The discussion of the systems that are not designed for ambient applications puts the attention on the differences of both kind of systems. We evaluate these systems against the specific requirements for the development of ambient applications, i.e. we highlight the advantages of those systems and we explain their limitations for the deployment in an ambient environment.

3.2.1 Higher-Order Behavioral Verification Techniques

The use of higher-order programming languages makes the design and implementation of many systems easier, more reusable and more general. The development of ambient systems is no exception in that regard. Object references are sent from one node to another and sending messages to objects transitively results in receiving more object references. In this context we discuss the related work that deals with contracts for higher-order functions and objects.

3.2.1.1 Runtime Verification Techniques

We start by giving an outline of the work that has been conducted for runtime verification followed by the work that has been done using static verification.

Contracts for Higher-Order Functions A decade ago Findler and Felleisen [FF02] introduced higher-order contracts in a Scheme like language [KCR98]. It differentiates between contracts defined over simple values and contracts defined over functions. Contracts over simple values are predicates for example, a predicate that checks whether a value is bigger than ten. Such contracts are called *flat contracts* while contracts over functions are called *function contracts*.

Function contracts are of the form $C_d \to C_r$ where C_d is a contract over the domain of the function and C_r is a contract defined over the range of the function. C_d and C_r can be either a flat or a function contract as the contract system supports higher-order contracts. Higher-order contracts are specified over the set of functions provided (exported) by a module. At all other parts in the program's code the use of contracts is transparent. Prior research showed that this way of specifying contracts is the most effective [FF02].

At the core of their contract system lies a validation mechanism that allows the programmer to define contracts over higher-order functions. As shown in section 2.4, higher-order function contracts are validated by postponing the validation of the contract until the contracted function is used. Because the validation of contracts over functions is postponed, it is no longer clear (for the programmer) which party violated the contract when a violation is detected. Therefore, the contract system tracks the responsibilities of the supplier and the client in order to point out the responsible party upon detecting a contract violation. This mechanism, called blame assignment, is one of the main contributions of their work. An in depth explanation of this mechanism is shown in section 5.1.

Recall from section 2.2.2, that in ambient systems the problems of volatile connections are mitigated by using asynchronous message passing in combination with futures. When the result of an asynchronous message is available the future is resolved with that value. Findler and Felleisen do not consider applications dealing with (asynchronous) communication but in their original paper they do point out the importance of dealing with callbacks. Their solution is based on a concept called *dependent contracts*. Dependent contracts allow the programmer to specify contracts that depend on the value of the function's *argument*. The signature of a dependent contract is as follows: $C_d \overset{d}{\rightarrow} (\lambda(a)C_r)$. In this signature C_d is a contract over the domain of the function and C_r is the range contract. The lambda expression is a function that is applied to the original arguments of the function and is expected to return the contract over the range. Because the range contract is nested in the lambda expression, it can access the arguments.

Findler and Felleisen propose to validate callbacks with dependent contracts by capturing the state S_x of the system relevant to validate the contract before the function is applied and then validate it against the state S_y of the system after the function is applied. This is possible with dependent contracts because the lambda expression of the dependent contracts is applied *before* the function is applied. After the contracted function is finished the range contract C_r is applied. This contract can then inspect the state S_x saved and validate it against the current state S_y of the system.

In later publications it becomes clear that the use of dependent contracts is not entirely satisfactory [BM06]. The main problem is that the arguments accessed in the postconditions are not subject to contract validation. Therefore, violations against the contracts defined over the arguments are not captured by the contract system. Moreover, it is not immediately clear how to define a contract over the arguments used in the post-condition. The reason is that a dependent contract itself may introduce violations against the arguments defined in the contract that can not be attributed to the supplier nor to the client of the contracted entity. The solution proposed by Dimoulas et al. in later publications [DF11, DFFF11] is to look at the contract validation code as an *independent* party and assign blame to the contract itself when a violation is detected during contract validation.

Temporal Higher-Order Contracts. Synchronization contracts in the presence of higher-order values have also been investigated. Higher Order Temporal (HOT) contracts [DFM11] extend prior higher-order contract systems to also express and enforce temporal properties between modules. In their formalization, a module's behavior is modeled as a trace of events such as function calls and returns. A HOT contract monitors this trace and validates it against a protocol. An important observation is that the monitored trace does not include internal module calls. When dealing with temporal behavior, it is not only important to check that a client respects a given protocol, but also that a provider of the contracted value obeys the protocol. This makes it possible to define a module that internally violates its own HOT contract but will never be blamed for it. A second drawback is the way in which HOT contracts are monitored. In their monitoring mechanism important primitive operations (for example message sending) fall outside of the scope of the contract system.

Relationally Parametric Polymorphic Contracts. Other extensions based on the original higher-order contracts work includes the introduction of universal quantifiers [GMFK07]. For example, the definition of a `map` function can be expressed by two polymorphic contract variables α and β: $(\forall(\alpha,\beta)((\alpha \rightarrow \beta)(listof\ \alpha) \rightarrow (listof\ \beta)))$. With this contract definition, the contract system verifies three important things: First of all, that the values in the list of α's are indeed homogenous; Second that these α's are transformed to a homogenous list of β's; And finally, that the map function indeed returns a homogenous list of β values i.e. that the `map` function does not invent some arbitrary values as a return value. Note that while these polymorphic values, look like primitive values they can be also functions. This means that when the `map` function returns a list of functions β the use of these functions is also monitored for consistent usage. This work has not been applied for object-oriented languages nor does it explore ambient-oriented concepts. Nevertheless, the basic concepts introduced in this work are very valuable in a distributed system. For example, it can validate that the values returned by a server (i.e. a distributed supplier) are constructed by the values given by the client.

Object-Oriented Higher-Order Contracts. Other work has looked into inheritance in combination with higher-order contracts [FLF01]. The key point of this work is to enforce a correct inheritance relationship and assign blame to the *contract* in case that the inheritance relationship of the contract is violated. This leads to the introduction of hierarchical contracts which validate that the subtyping relationship follows the Liskov substitution principle [LW94]. In case that this principle is violated blame is assigned to the hierarchy.

Higher-Order Contract Satisfaction A large body of spin-off research has been conducted inspired by the initial work on higher-order contracts. In particular, a large group of research has dealt with the fundamental question of what it means to satisfy a higher-order contract. In the original Findler and Felleisen paper they do not provide an independent definition of what it means for a component to satisfy its contract. Blume and McAllester [BM06] are the first to provide a definition of contract satisfaction. In their work they point out that the original proposal by Findler and Felleisen misses a number of contract violations. Findler and Blume [FB06] redefine the original higher-order contract system as a pair of projections and explain the discrepancy. Hinze et al. [HJL06] and Chitil et al. [CMR03] add contracts to lazy languages and propose with two different systems in terms of contract satisfaction. Greenberg et al. [GPW10] notice that contract satisfaction with the addition of dependent higher-order contracts complicates the picture. Dimoulas and Felleisen [DF11, DFFF11] use the idea of observational equivalence and establish the notion of *independent contracts*.

With independent contracts the contracts themselves are seen as an independent party which can be blamed for violating other contracts.

This concludes our overview of contract systems which are dynamically validated in a higher-order setting. One of the challenges of designing an ambient contract system is to deal with values beyond data types. The work on contracts for higher-order functions is thus very valuable. Spin-off research spawned from the work on contracts for higher-order functions has not focussed on ambient-oriented systems. Language features such as *non-blocking* futures have not been considered. The existing work on contracts for higher-order values also does not explore the scope of contract validation in the presence of an event-driven system.

3.2.1.2 Static Verification Techniques

We now turn our attention to contract systems which statically verify contracts.

Higher-Order Static Verification. There has been a significant amount of research in the field of specification languages that deal with the (static) verification [1] of higher-order functions.

Ernst et al. [ENO82] have proposed a verification system in order to prove the correctness of programs with functions that take other functions as one of their arguments. In their framework, the pre and post-conditions of a function can refer to the pre and post-conditions of the function's arguments. While this allows a certain functionality to be verified, it requires a considerable effort from the programmer. First, the programmer must be able to (re)construct the arguments passed to the contracted function. Second, the programmer must be able to (re)construct all the invocations to the function argument in the body of the contracted function. This is particularly difficult in case of control structures such as if statements or loops. Finally, in order to define such specifications, the specification language needs to be a higher-order logic as the pre and post-conditions themselves refer to pre and post-conditions. Verification of such higher-order logic is a process that usually needs human intervention, for example, in interactive proof systems. However, in their work they show that the higher-order logic can be often be reduced to a classical logic. Another limitation of this work is that they only consider a semantics where parameters are passed by copy. Furthermore, they only support the verification of functions that receive other functions as an argument, i.e. they do not specify the semantics of returning functions.

A similar system based on Hoare-Logic has been proposed by Damm et al [DJ83]. In their system, they explicitly limit the language to one that does not have global variables in a procedure body. The reason for this limitation (already proven by Clarke [Cla77]) is that a language that has: procedures as arguments, recursion, static scope global variables in procedure bodies and nested procedure declarations can not have a sound and relative complete Hoare-Logic. More surprisingly, Clarke showed that whenever one of these language constructs is removed, a sound and relative complete Hoare-Logic can be derived [Cla77]. He showed this for most variations but did not show it for the elimination of global variables. In their paper, Damn et al., derive a system where it is possible to refer to the Hoare Triples of functional arguments, resulting in a system with similar limitations as the one of Ernst et al.

[1] We use verification instead of validation to make the distinction with the systems discussed before.

In short, although there is quite a lot of work on the static verification of higher-order functions, the work does not take into account many of the standard language features of modern programming languages. Finally in order to truly work, severe restrictions must be put on the underlying language.

Abstract Interpretation. Abstract interpretation is a program verification technique that allows the concrete values of a program to be mapped on abstract ones in such a way that the subsequent interpretation with abstract values gives some useful information about the concrete values [CC77]. The prototypical example is the rule of signs. For example the text -42 * 28 can be mapped on the abstract domain $\{(+), (-), (\pm)\}$. The abstract interpretation $-42 * 28 \Rightarrow (-) * (+) \Rightarrow (-)$ effectively proves that the result of the example will be a negative number. Abstract interpretation however, is not always as accurate as one would want it to be. For example, the abstract interpretation $-42 + 28 \Rightarrow (-) + (+) \Rightarrow (\pm)$ only concludes that the result can be positive or negative.

Extensions of abstract interpretation have been applied in the context of higher-order contracts by Tobin-Hochstadt et al. [THVH12]. The focus of this work is to use higher-order contracts as the specification of abstract values. This technique has the advantage that the programmer no longer needs to do a whole programing analysis as the modules on which the program depends can be abstracted by their contracts. In their work, they describe a tool that uses their abstract interpretation for the verification of Scheme programs. Similar work has been performed in the Haskell research community [XPJC09] with the use of compiler optimization techniques. Another similar approach can be found in Clousot for the verification of contracts written in the .NET languages [FL11]. Those approaches, however, still need a whole program analysis and they do not support higher-order contracts on argument values.

Object-Oriented Higher-Order Contracts. Many of the static verification systems discussed so far do not take into account object-orientation whatsoever. In this section, we give an overview of systems that do deal with contracts over values in an object-oriented setting.

In the context of behavioral specifications, Soundarajan et al. propose a contract system that allows the programmer to specify constraints over the execution traces of a method [SF04]. In their work, they show several examples of how to use their system in order to specify traces that enforce behavioral subtyping. Unfortunately, these execution traces are difficult to specify and reason about and they require a significant amount of effort in order to deduce proofs for them.

Extensions to the Java Modeling Language (JML) language have dealt with higher-order methods (HOM) as proposed by Shaner et al. [SLN07]. They define a higher-order method as any method whose behavior critically depends on one or more mandatory calls. Their approach to this problem is to adopt a gray box abstraction and make use of model programs in order to specify the mandatory method calls. While their approach is simple, it can encode a number of interesting patterns such as for example the observer design pattern. Unfortunately, their approach is not very flexible in the sense that the model program is entangled with the base-level functionality. The reason for this is that the matching of mandatory function calls from the model program happens in a purely syntactic way. This implies that even the smallest variation may lead to a violation. Moreover, it does not differentiates programs that make two or more mandatory method calls instead of one. A related approach by Tyler and Soundarajan [TS03] allows trace-based specifications, but suffers from similar limitations as Shaner's system.

While JML approaches allow contracts to be defined over classes they do not allow true higher-order contracts in the sense of Findler and Felleisen [FF02]. Contracts in JML and other contract languages depend on the *nominal type* when defining object contracts. Therefore, is possible to define contracts at the class and interface level but it is not possible to specify a contract over an object received as argument other than to define a new interface or class. Often, however, the programmer that wants to impose a stronger contract is not the programmer that originally created the class. In such cases, it is possible for the programmer to write an entirely new class that bridges the gap between the two components that are being composed. However, in such a case, the programer needs to manually maintain the adaptor, validate the interactions between the composed components and signal errors. The programmer thus has the task to manually identify violations and assign blame in case of a violation. Dealing with blame assignment in such an ad-hoc fashion is less trustworthy than when blame assignment is validated by a dedicated contract system. This problem was put forward by Robert Findler et al. [FFF04] and their solution is to allow the refinement of contracts on the method arguments with a construction called *semantic casts*.

3.2.1.3 Conclusion

An overview of the contract systems dealing with higher-order values is shown in figure 3.1. From this overview it can be seen that the runtime verification of object-oriented higher-order contracts exhibit many of the properties that are required for the specification of an ambient contract system. Unfortunately, these systems have not been applied in an event-driven distributed language. Furthermore, we observe that static systems both for the verification of higher order functions and objects require a substantial amount of programming effort from the programmer compared to runtime validation. Moreover, the static verification of object-oriented programs is mostly investigated in class-based languages. As shown by Dedecker et. al. [Ded06], class-based languages have proven to be difficult to reconcile in an ambient-oriented setting. Therefore, we conclude that these systems are less appropriate as a starting point for the definition of an ambient contract system.

	Higher-Order Verification					
	Higher-Order Values		Protocols	OO	Class	Event
	Dom	Range				
Runtime Verification						
Higher-Order Contracts	Y	Y	N	N	N	N
HOT Contracts	Y	Y	Y	N	N	N
Semantic Casts	Y	Y	N	Y	Y	N
Static Verification						
Ernst et al.	Y	N	N	N	N	N
Clarke	Y	Y	N	N	N	N
JML extensions	N	N	Y	N	Y	N
Abstract Interpretation						
Racket	Y	Y	N	N	N	N
Haskell	Y	Y	N	N	N	N

Table 3.1: Higher-Order Contract Validation.

3.2.2 Behavioral Contracts for Concurrent and Distributed Systems

In this section, we provide an overview of contract systems for the use in concurrent and distributed systems.

3.2.2.1 Concurrency

Pre and post-condtions as defined by Hoare [Hoa72] can be chained together using inference rules so that the programmer can compose contracted components in a correct way. In a sequential setting, testing a component in isolation is enough to be sure that this component will behave the same when used in a composition. Unfortunately, in a concurrent language this assumption no longer holds. For example, consider a stack which has a pre-condition defined over the `pop` operation which specifies the stack can not be empty. In a single threaded environment the client can easily ensure this property by verifying that the stack is not empty before executing a `pop` operation. In a concurrent setting, it is impossible for a client to ensure this property. Other clients of the stack might execute a `pop` operation concurrently while the client invokes the `isEmpty` operation. Therefore, the client can never be sure that the stack is not empty before executing the `pop` operation. Even more severe is the fact that the evaluation of the precondition can be executed concurrently with another `pop` operation leading to a system where a runtime exception within the `pop` operation is thrown while the intent of the programmer was to avoid this runtime exception by constraining the stack with the precondition. The phenomenon that during the execution of a concurrent system the precondition can be violated during the execution of an operation is known as the *concurrent precondition paradox* [NMO09].

As shown in the previous chapter, such concurrency issues are eliminated by design in the event-loop concurrency model since concurrency is limited to the concurrent execution actors. The pre and post-conditions of ambient contracts over objects are always executed within a single actor. As there can only be one thread of execution the programmer does not need to be concerned with the concurrent precondition paradox found in other concurrent languages.

3.2.2.2 Distribution

The bulk of advanced contract systems for distributed systems can be summarized by systems for wireless sensor networks and web services.

Sensor Networks. Sensor Networks are a popular kind of network with characteristics similar to mobile ad-hoc networks. Although the sensors in such a network are typically stationary, they have a high rate of failure e.g. because the sensor runs out of battery power. A difference with mobile ad-hoc networks is that the computational power available in a sensor network is much less than the one of a mobile phone. Passive distributed assertions (PDA) are introduced to cope with the distributed assertions in the context of wireless sensor networks [RM09]. The programmer can specify assumptions over the program variables of several distributed sensors. He has the ability to do this in a declarative manner as in PDA there are abstractions for location designation. While the underlying abstractions are interesting in the light of mobile ad hoc networks, unfortunately the design of this contract system assumes the availability of a centralized server.

Web services. Work on distributed contracts has focused on the development of web services. As summarized by Castagna et al most of this work finds it roots in the formal framework of contracts for process calculi [Mil82], and session types [CGP08]. Carbone et al. allow the programmer to capture web service choreographies by means of a global calculus [CHY07]. The individual processes are obtained as projections of the global description. Fournet et al. [FHRR04] make extensions to CSS in order to derive that certain substitutions of processes will lead to a deadlock free composition or not.

Many of these formalisms have found their entrance at the industrial level in languages such as the Web Service Description Language (WSDL) and the Web Service Conversation Language (WSCL). While WDSL only allows asynchronous and synchronous request reply style interactions, WSCL allows more extensive protocols which can even include cycles. For example, Hollunder [Hol] added code contracts to the web-service description language so that .Net programs can reap the benefits of design by contract while developing web-services. These code contracts are, however, limited to invariants, pre and post-conditions, they do not support higher-order contracts. Moreover, they do not specify how programmers can express synchronization contracts.

3.2.3 Systems for Synchronization Verification

In the previous sections we have mainly shown systems for behavioral validation. We now turn our attention to systems for dealing with synchronization. There is a large body of work in type systems that allows the programmer to state specific properties over the execution of his program. In general, a type system allows the programmer to annotate his program to express certain properties over the annotated entities. For example, the programmer can annotate a variable to express that it will always refer to an integer. A type system then constructs a proof from the program text in order to make sure that all these annotations are consistent. For example, when a program is correctly typed and a variable is annotated as an integer, the type system ensures that at runtime this variable never refers to any other value than an integer. Under certain conditions it is also possible to alleviate the task of placing annotations and let the type system infer the types for the programmer. In recent years, there has been a tremendous progress in the kind of properties for which a type system can provide guarantees. For example, type systems have been used for interaction protocols and to verify the state of objects. While these type systems are not specifically designed to act as contracts they have similar goals as we show in the remainder of this section.

3.2.3.1 Session Types.

A session is an ordered set of communication interactions between two parties [MY07]. Session types have been combined with the π-calculus [Mil99], and have even been added to a distributed Java [DCMYD06]. At the industry level, a well known adaptation of session types is the WC3 Choreography Web Description Language.

Basic session types describe the sequence of input and the output values that the client or the server has to send and receive over a shared communication channel. A session type is basically a sequence of primitive types such as `int`, `string`, `bool` etc. together with the sequencing operator indicated with a dot (`.`). In a session type S, ? implies receive, ! send, dot (.) sequencing and End indicates the closing of the channel. For example, a possible specification of a protocol for the server which receives two integers and returns their sum could be encoded as: $\langle ?Int.?Int.!Int.End\rangle$ For basic protocols, the client protocol and the server protocol are each others dual and can be obtained by changing all sends (!) into receives (?) and all receives (?) into sends (!). Consequently the session type of the client for the multiplication service is: $\langle !Int.!Int.?Int.End\rangle$

Next to sequences of input and output values, session types also support the traditional internal ($\oplus$) and external choice ($+$) operators. For example the session type shown below specifies that the server has the internal choice to transition to OK or to NOK. When the server selects OK it waits for two numbers and returns a number (their sum), when the server selects NOK the session is closed immediately.
$\oplus\langle OK \rightarrow ?Int.?Int!.End; NOK \rightarrow End\rangle$
Consequently the client has to be able to anticipate both state transitions, i.e. the dual session type of the client is as shown below.
$+\langle OK \rightarrow !Int.!Int?.End; NOK \rightarrow End\rangle$

Session types are typically integrated into a concrete language by introducing primitives to make and type explicit channels. Type checking of session types is done by going over the program text and verifying that the successive operations over the channel follow the session type specification. Both branches of an if statement have to leave the channel in the same state, so that the channel is in the same state independently of which branch is taken at runtime.

Object-oriented Session Types. Session types have also been explored in the context of object-oriented languages. Moose [DCMYD06, DCGDY07] is a multi-threaded object-oriented language with support for session types based on channels. In their description of future work, the authors of Moose mention that it would be interesting to merge remote method invocations (Java RMI) with session types instead of using explicit channels. As shown before programing languages based on RPC such as (Java RMI) introduce blocking communication primitives that can easily lead to deadlocks and thus are not considered to be applicable in an ambient environment. Their systems first requires further research to be truly object-oriented as well as extra research to be combined to a non-blocking communication model.

In STOOP [DDCC07] and SAM^g [CCDC$^+$09] the integration of object-orientation with session types has been taken further and channels are made implicit in the language. However, in their system values can be exchanged within the body of a method with primitives `send` and `receive` that operate on an implicit channel. This forces the programer to think in a channel-oriented style instead of an object-oriented style.

The work on session types that is the most interesting for an ambient contract system is "modular session types for distributed object-oriented programming" [GVR$^+$10] by Simon J. Gay et al. In their work they developed a distributed object-oriented language with type-state where the availability of methods must follow the session-type of a channel accessible from within the object. Even though this channel is hidden within the verified object, the programmer still needs to write his code using a channel-based communication model.

Session Type Limitations Session types suffer from a number of limitations in order to adopt them in an ambient-oriented programming model. The main issue is that the use of explicit communication channels in traditional session types is hard to combine with an object-oriented (ambient) programming language. First, distributed object-oriented languages already have a means to send and receive values by means of remote object references. The introduction of a communication channel requires the programmer to think in two different paradigms, channel-based and an object-oriented based one. Moreover, distributed objects and channel-based communication can interfere with each other. For example, what should happen with an active channel when an object that encapsulates it is sent over another communication channel ? Finally, the primitives defined over the communication channels usually block the thread in which they are executed. As shown in section 2.2.2, in ambient applications the use of such blocking operations significantly increases the impact of disconnections on the application. For example, a service waiting for a value from a disconnected client may be blocked forever.

3.2.3.2 Contracts for Process Calculi

Behavioral contracts for process calculi [CP09, ABZ10] are very similar to session types. The main difference is that instead of describing the type of a single communication link they describe the type of a whole process. While this difference may not seem to be paramount at first sight, it has important implications with respect of the kind of global properties that can be proven by the type system. The main advantage of process calculi over session types is that there are less restrictions over the communication channels in order to ensure global properties such as progress.

A process type T is either deadlocked (0) successfully terminated (1) or can make infinite internal progress (Ω). Processes can send (α!x) or receive (α?x) a value x over a communication channel alpha. As usual the dot operator is used as sequencing operator i.e. first sent x over α and then behave as T is indicated as $\alpha!x.T$ Processes can be combined by the parallel composition operator (|). Processes can make internal ($\oplus$) and external choices (+). External choices are based on the communicated value for example: $(\alpha?string + \alpha?int)$. Just like it is an open research question how session types can be integrated into an object-oriented system, it is not clear how process calculi with channels can be fitted into an object-oriented system.

3.2.3.3 Grey Box Verification Techniques.

There exists a group of research frameworks that focuses on grey box verification techniques. These verification mechanisms allow the programmer to define more expressive verification statements than simple pre/post conditions. Helm et al. [HHG90] and Holland [Hol92] were among the first ones to investigate such advanced mechanisms. Their approach uses *model programs* in order to describe contractual specifications, but they do not present a method for automatic conformance monitoring.

MaC [DJLS08] is a runtime verification system where program executions points, such as the application of a function, are reified as events. Over these events the programer can write rules in order to verify the program execution. While it is likely that the expressive power of the MaC system allows computational contracts to be defined it has not been designed for higher-order functions. Moreover, blame assignment is also not considered.

State invariants also known as integrity constraints in the world of databases are constraints over how certain operations are allowed to affect the state of the program. Early specification languages suffered from the inability to express that a certain operation only modifies what it should modify and leaves everything else untouched. This problem has been called the *frame problem* and finds its origins in the field of artificial intelligence. When a logical system is built for a certain world, there is an implicit specification that when a certain change in this world takes place, all the rest remains the same. However, in many proof systems the specification of such rules requires explicit statements about the fact that everything else is not affected. The specification of all the unaltered states is tedious and difficult to extend when, for example, new elements are added that also remain unaltered. Similarly, in the specification of integrity constraints the frame problem has manifested itself. As a solution to the frame problem languages as JML have introduced the notion of *frame axioms*. A frame axiom consists of a constraint over an operation which specifies which state might be altered by the operation. Such frame axioms can be used for the specification of disallowing certain state to be altered.

A remarkable contract system that goes beyond pre- and post-conditions was recently proposed by Heidegger et al. [HBT12]. They propose access permission contracts, which allow programmers to annotate methods with a set of read and write access paths. During the execution of a contracted function the dynamic extent of the contracted function can only read and write to those variables in their access paths. Access permission contracts are a particular instantiation of computational contracts as shown in chapter 5.

A very related approach by Fischer [FII00] introduces trace-based assertions. These contracts allow the verification of code to follow a certain protocol. However, trace-based assertions do not support functional contracts to be defined over the argument values of a function.

3.2.3.4 Typestate-Oriented Programming.

In typestate-oriented approaches object methods are restricted depending on which state the receiving object is currently in [SY86, SNS$^+$11]. The prototypical example of typestate-oriented programming is the verification of a file handle. When the file is closed the write operation should not be applicable and when it is opened the write operation becomes applicable. Typestate-oriented programming allows the programmer to express such concerns and – under certain restrictions – verify them statically. With typestate-oriented programming, it is thus possible to govern the interactions between objects in a similar way as session types. The main difference between typestate-oriented approaches and session types is that session types unify typing and interactions into one global type while this is not the case in typestates. Their limitations, however, are very similar: in order to enforce multi-party interactions they resort to restrictions such as linear typing. This means that program variables x can only be used once and simple programs such as $f(x,x)$ can not be verified. Despite these limitations, typestate-oriented programming is valuable technique that is able to statically verify highly complex programs [SNS$^+$11]. However, it has not been investigated how typestate-oriented programming can be applied in a system where the objects communicate over (volatile) connections using asynchronous message passing.

3.2.3.5 Type and Effect Systems.

There is a branch of type systems that is able to express which effects a certain computation exhibits and in which context it may have these effects. A simple example of a well-known type and effect system is the validation of exceptions. Functions have to declare that they might throw an exception and the functions that apply an exception throwing function have to catch these exception or declare that they might throw this exception themselves. However, the example of exception throwing is relatively simple, type and effect systems can be used for the static verification of a range of effects [MM09]. Type and effect systems thus allow programmers to verify certain protocol like constraints over their programs. However, type and effect systems are usually very specific about the effects that a function can have for example, to throw an exception. How these systems can account for arbitrary effects such as invoking a certain method is not clear.

3.2.3.6 Conclusion

There is a significant progress in the kind of synchronization properties that can be verified statically. However, many of the static verification mechanisms shown for dealing with complex first-class values are not clear in how they could be used in a practical prototype-based ambient programming language. First, many of the proposed solutions have opted for a (blocking) channel-based communication model which has proven to be incompatible with the volatile connections of mobile ad hoc networks. Typed asynchronous communication channels have been explored but it is unclear how such models can be used in a distributed object-based language. Finally, typestate-oriented programming and type and effect systems are two widely known techniques, which (to some extend) allow local synchronization contracts to verified statically. However, it is unclear how these techniques could be scaled to operate in a distributed environment.

3.2.4 QoS Contracts

In the component-based middleware community, contracts have been incorporated in order to compose and adapt applications in order to meet a certain quality of service. QoS frameworks such as QuO [ZBS97], QML [FK99] and 2KQ+ [LN99], provide abstractions in order to enforce a contract over the *bindings* between a client and a server component. Depending on the implementation, a QoS contract verifies quality constraints such as latency, duration of a computation, throughput etc. The contracts in such systems mainly describe how the component should adapt itself depending on these quality constraints. The typical example of such frameworks is to switch between compressing an image when the connection is slow or sending the uncompressed image when the connection is fast. Monitoring such QoS constraints can happen on the application of certain functionalities of the component or during their execution. Several QoS contracts have made use of aspect technology in order to adapt the application according to the context [FK99, HBG+01]. The focus of these contract systems is to adapt the application such that a certain QoS can be offered. Such QoS contracts can be very useful in the context of ambient-applications in order to make sure that the application adapts itself in a well-defined way. However, in this dissertation the focus lies on a contract system that monitors the flow of values between the different modules of the ambient-application. Validating how the application adapts itself to the current context in order to meet certain QoS is an orthogonal dimension that has not been explored in this dissertation.

3.2.5 Aspect-Oriented Programming and Contracts

Many contract systems use aspect-oriented technology in order to validate contracts. In our work we have also explored aspect-oriented programming in order to validate contracts. Therefore, we give a short overview of the related work on aspect-oriented programming in relation to contracts.

Aspect-oriented programming allows the specification of crosscutting concerns in a modular way so that they are no longer scattered through the code but localized in one place of the code. These modular crosscutting expressions are defined by the specification of additional behavior called *advice* on particular points in the programs execution called *join points*. The developer specifies on which set of join points the advice has to be executed by specifying a *pointcut*.

Dutchyn et al. have explored the semantics and scoping of aspects in a dynamically typed higher-order languages [DTK06]. Ambient contracts have built on top of their results in order to allow the interception of critical points in the program that will be subject to contract verification. While their scoping mechanism has proven extremely valuable during the implementation of our system, their aspect language does not allow the programmer to express contracts nor to assign blame in case of a violation.

Frameworks such as Barter [Sza02], Jose [FBT06] and Contract4J [Wam06] have also used aspect-oriented programming as an implementation technique in order to provide design-by-contract. However, these techniques do not support blame assignment in the context of higher-order programming languages. Several systems have added contracts for aspects [BRLM11, SL04] where the focus lies on the definition of contracts *over* an aspect. For example, Pipa [ZR03] extends JML [LCC$^+$05] to support DbC for programs written with AspectJ. However, they do not focus on using aspects in order to verify certain properties of the computation. Finally, Contract-Based Verification for Aspect-Oriented Refactoring [UPST08] uses aspects in order to specify contracts over refactoring. Scoping strategies for distributed aspects are investigated by Tanter et al. [TFD$^+$10].

Aspect-oriented programming has also proven to be useful for dealing with context-awareness [FG09, TGDB06]. This is *not* the focus of this thesis, where we have used aspects as an interception abstraction to intercept interesting points in the execution of a certain function application.

3.3 Conclusion

There is a large body of related work on contract frameworks for a wide variety of applications. In this chapter we have shown an overview of the systems which have similar goals as ambient contracts. We have first showed that related work on contract systems in general has been categorized in four levels: syntactic, behavioral, synchronization and quality of service. We evaluated these four categories and outlined the focus of our ambient contract system which mainly focuses on behavioral contracts. From our survey of related work we conclude that:

- Syntactic contracts would require a lot of annotations in order to be able to verify properties over objects, far-references and future values. Most of the static verification mechanism can not deal with the appearance of random communication partners at runtime. As the ad-hoc nature of ambient applications precisely require a contract system that maintains such ad hoc interactions static verification techniques have not been considered as a suitable starting point for the definition of ambient contracts. Moreover, as we showed in the related work static verification techniques in the presence of higher-order functions requires strong restrictions over the underlying programming language.

- Runtime behavioral contracts support some of the necessary properties for the development of ambient-oriented programs but not all of them. Nevertheless, the higher-order contracts systems developed from the work on higher-order function [FF02] contracts seem to be the best starting point for the development of an ambient contract system. This is also the approach followed in this dissertation.

- Synchronization contracts do not make a lot of sense for the definition of an ambient contract system because many of the concurrency issues such as low level race conditions and deadlocks are precluded by the ambient-oriented programming model. Therefore, most of the work on concurrent contract validation is not applicable for the definition of an ambient contract system. We therefore focused in the related work on synchronization contracts over the messages in a protocol like fashion. Within an actor synchronization is dealt with by the underlying programming paradigm. Synchronization over the messages has been dealt with by session types, CCS extensions. However, these systems introduce severe restrictions in order to deal with higher-order communication channels. Most systems require that communication channels are used exactly once (linearly typed). In an object-oriented language, the objects themselves are used as communication channels. Sending an object to a remote communication partner in a linearly typed language would thus leave a hole in the object graph of the sender. Typestate-oriented programming requires similar restrictions. It is thus unclear how higher-order communication channels can be transparently integrated in a distributed object-oriented language.

- Quality of service contracts are more related to the work in context aware programming as opposed to building bigger and more robust ambient applications which is the focus of this dissertation. Most of the work on quality of service actually does not monitor the flow of values between the different components. Instead they formulate a system that is dynamically adapted in order to conform to a certain specification. While such systems are very useful, they go far beyond the responsibility of what normally constitutes a contract system. A contract system as interpreted in this dissertation should not adapt the execution of the underlying program.

Figure 3.5 provides an overview of the related work compared to the design space that we outlined in section 2.4.5. In this overview we do not consider contract systems which do not have support for blame assignment because we investigate contract systems that are validated at runtime. At each intersection of the three axes there is a different kind of contract for example: a computational contract over an object at the server-side. We colored each of these intersection points in order to give an overview of the contract systems that are explored by the related work. Red squares indicate that related work does not have support for such contracts and green squares indicate that they have been explored. White squares indicate non applicable systems. For example, a contract to monitor the behavior of a number does not make a lot of sense. As can be seen in this overview, related work has left open many questions for the design of an ambient contract system. At the top we can also see that the use of non-blocking futures in combination with their callbacks is not explored. However, as we argue in the rest of this dissertation, the contract system proposed by Findler and Felleisen [FF02] provide a good basis on which we have built ambient contracts.

In the next chapter, we first give an overview of communicating event loops to clearly specify the underlying programming model in which ambient contracts are designed. In the chapters following our exposition of communicating even loops, we give

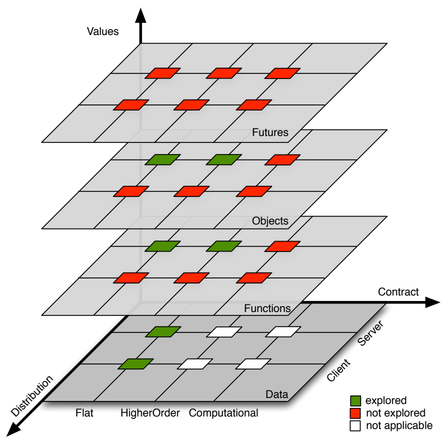


Figure 3.5: Overview of the design space of ambient contracts explored by the related work.

an overview of how ambient contracts are able to deal with each of the points in the ambient contracts design space.

Chapter 4

Communicating Event-Loops: Formal Specification

In this dissertation we advocate the use of ambient contracts in order to specify the behavior of a volatile group of distributed objects communicating over an uncertain network. This chapter gives a formal treatise of the ambient actor model which is the computational model for concurrency and distribution, that adheres to the AmOP criteria presented in chapter 2. The formal semantics of the programming kernel we present in this chapter is further used in the remainder of this dissertation to define the language constructs that constitute ambient contracts. The explanation given in this chapter closely follows the technical report as shown in [VCSHDM].

4.1 AmbientTalk Operational Semantics

Our exposition of the semantics of a subset of AmbientTalk, named AT-LITE, is based primarily on that of the Cobox model [SPH10]. Coboxes feature a similar runtime model, but differ on important points such as the ability to execute multiple coroutines inside a single actor, and the ability to block (suspend) on a future. Our notion of futures as presented here is significantly different from the notion of futures as presented in the Cobox model as our notion of futures does not involve any blocking operation.

Our operational semantics models objects, isolates (pass-by-copy objects), far references, actors as event loops, non-blocking futures, asynchronous message sending and inter-actor parameter-passing. In Section 4.1.3 we extend AT-LITE with the necessary primitives for service discovery and buffered communication.

4.1.1 Syntax

Figure 4.1 lists the different semantic entities of AT-LITE. Caligraphic letters like $\mathcal{F}$ and $\mathcal{M}$ are used as "constructors" to distinguish the different semantic entities syntactically. Actors, futures, resolvers and objects each have a distinct address or identity, denoted ι_a, ι_f, ι_r and ι_o respectively.

Configurations are sets of concurrently executing actors. Each actor is an event loop consisting of an identity ι_a, a heap O denoting the set of objects, futures and resolvers *owned* by the actor, a queue Q containing a sequence of messages to process in the future, and the expression e that the actor is currently executing.

$K \in$ **Configuration**	$::=$	A	Configurations
$a \in A \subseteq$ **Actor**	$::=$	$\mathcal{A}\langle \iota_a, O, Q, e \rangle$	Actors
Object	$::=$	$\mathcal{O}\langle \iota_o, t, F, M \rangle$	Objects
Future	$::=$	$\mathcal{F}\langle \iota_f, Q, v \rangle$	Futures
Resolver	$::=$	$\mathcal{R}\langle \iota_r, \iota_f \rangle$	Resolvers
$\mathrm{m} \in$ **Message**	$::=$	$\mathcal{M}\langle v, m, \overline{v} \rangle$	Messages
$Q \in$ **Queue**	$::=$	$\overline{\mathrm{m}}$	Queues
$M \subseteq$ **Method**	$::=$	$m(\overline{x})\{e\}$	Methods
$F \subseteq$ **Field**	$::=$	$f := v$	Fields
$v \in$ **Value**	$::=$	$r \mid \text{null} \mid \epsilon$	Values
$r \in$ **Reference**	$::=$	$\iota_a.\iota_o \mid \iota_a.\iota_f \mid \iota_a.\iota_r$	References
$t \in$ **Tag**	$::=$	$\textsc{o} \mid \textsc{i}$	Object tags

$$o \in O \subseteq \textbf{Object} \cup \textbf{Future} \cup \textbf{Resolver}$$
$$\iota_o \in \mathbf{ObjectId}, \iota_a \in \mathbf{ActorId}$$
$$\iota_f \in \mathbf{FutureId} \subset \mathbf{ObjectId}$$
$$\iota_r \in \mathbf{ResolverId} \subset \mathbf{ObjectId}$$

Figure 4.1: Semantic entities of AT-LITE.

Objects consist of an identity ι_o, a tag t and a set of fields F and methods M. The tag t is used to distinguish objects from so-called *isolate* objects, with $t = \textsc{o}$ denoting an object and $t = \textsc{I}$ denoting an isolate. Isolates differ from regular objects in that they are parameter-passed by-copy rather than by-reference in inter-actor message sends, but otherwise behave the same.

AT-LITE supports *futures*, which are first-class objects that are placeholders for a value that is asynchronously awaited. Futures consist of an identity ι_f, a queue of pending messages Q and a resolved value v. A future is initially *unresolved*, in which case its resolved value v is set to the unique empty value ϵ. As long as the future is unresolved, any messages sent to the future are queued up in Q. When the future becomes resolved, all messages in Q are forwarded to the resolved value v and the queue is emptied.

A *resolver* object provides the right to assign a value to its unique paired future. Resolvers consist of an identity ι_r and the identity of their paired future ι_f. The resolver is the only means through which a future can be resolved with a value. Our notion of future-resolver pairs descends directly from E's promise-resolver pairs [MTS05], which are themselves inspired by logic variables in concurrent constraint programming [Sar93].

Messages are triplets consisting of a receiver value v, a method name m and a sequence of argument values $\overline{v}$. They denote asynchronous messages that are enqueued in the message queue of actors or futures.

All object references consist of a global component ι_a that identifies the actor owning the referenced value, and a local component ι_o, ι_f or ι_r. The local component indicates that the reference refers to either an object, a resolver or a future. We define $\mathbf{FutureId}$ and $\mathbf{ResolverId}$ to be a subset of $\mathbf{ObjectId}$ such that a reference to a future or a resolver is also a valid object reference. As such, $\iota_a.\iota_o$ can refer to either an object, a resolver or a future, but $\iota_a.\iota_f$ can refer only to a future.

Syntax Figure 4.2 shows the syntax of the AT-LITE language. AT-LITE features both object-oriented and functional elements. The functional elements descend directly from the λ-calculus. Anonymous functions are denoted by $\lambda\overline{x}.e$. Variable lookup in AT-LITE is lexically scoped. Local variables can be introduced via $\text{let}\, x = e \,\text{in}\, e$.

AT-LITE is also an imperative classless object-oriented language. It features `object` and `isolate` literal expressions to define fresh, anonymous objects. Objects consist of a set of fields and methods. Fields may be accessed and updated. Methods can be invoked either synchronously via $e.m(\overline{e})$ or asynchronously via $e \leftarrow m(\overline{e})$.

In the context of a method, the pseudovariable `this` refers to the enclosing object or literal. `this` cannot be used as a parameter name in methods or can not be redefined using `let`.

New actors can be spawned using the `actor` literal expression. This creates a new object with the given fields and methods in a fresh actor that executes in parallel with the creating actor. Actor and isolate literals may not refer to lexically enclosing variables, apart from the `this`-pseudovariable. That is, they must satisfy $FV(e) \subseteq \{\text{this}\}$ for all field initialiser and method body expressions e. Isolates and actors are literally "isolated" from their surrounding lexical scope, making them self-contained.

New futures can be created explicitly using the expression $\text{let}\, x_f, x_r = \text{future}\,\text{in}\, e$. This binds a fresh future to the variable x_f and a fresh, paired resolver object to x_r. A resolver object denotes the right to assign a value to its paired future. The expression $\text{resolve}\; x_r\; e$ resolves the future x_f via its paired resolver x_r with the value of e. The value of a future x_f can be awaited using the expression $\text{when}(x_f \rightarrow x)\{e\}$. When the future becomes resolved with a value v, the expression e is evaluated with x bound to v.

AT-LITE supports two forms of asynchronous message passing. Expressions of the form $e \leftarrow m(\overline{e})$ denote one-way asynchronous message sends that do not return any value. If a return value is expected, the expression $e \leftarrow_f m(\overline{e})$ denotes a two-way asynchronous message send that immediately returns a future for the result of invoking the method m.

Syntactic Sugar Functions are defined as objects with a single method called `apply` as shown in Figure 4.2. Note that function definitions can appear at arbitrary positions within the code, also within methods. Therefore care has to be taken in order to make sure not to alter the semantics of `this` in such a context. In AT-LITE the substitution $[x_{this}/\text{this}]e$ is necessary to ensure that within function bodies nested inside object methods, the `this`-pseudovariable remains bound to the original enclosing object, and not to the object representing the function. In order to make sure that this substitution works for top-level defined functions the `this` variable is bound to `null` when the actor is initialized. Function application $e(\overline{e})$ is desugared into invoking an object's `apply` method.

A two-way message send $e \leftarrow_f m(\overline{e})$ is syntactic sugar for a one-way message send that carries a fresh resolver object x_r, added as a hidden last argument. The message m is marked m_f such that the recipient actor can decode the argument list, knowing that it has to pass the result of the method invocation to x_r. The two-way message send itself evaluates to the future x_f paired to the passed resolver x_r.

The desugaring of "when" and "resolve" make use of special messages named resolve_μ and register_μ. The μ (for "meta") suffix identifies these messages as special meta-level messages that should be interpreted differently by actors. A regular AT-LITE program cannot fabricate these messages other than via the "when" and "re-

Syntax

$$\begin{array}{lcl}
e \in E \subseteq \textbf{Expr} & ::= & \text{this} \mid x \mid \text{null} \mid e \; ; \; e \mid \lambda\overline{x}.e \mid e(\overline{e}) \mid \text{let } x = e \text{ in } e \mid e.f \mid e.f := e \\
& \mid & e.m(\overline{e}) \mid \text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \mid \text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
& \mid & \text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \mid \text{let } x_f, x_r = \text{future in } e \mid \text{resolve } e\; e \\
& \mid & e \leftarrow m(\overline{e}) \mid e \leftarrow_f m(\overline{e}) \mid \text{when}(e \rightarrow x)\{e\}
\end{array}$$

$$x, x_f, x_r \in \textbf{VarName}, f \in \textbf{FieldName}, m \in \textbf{MethodName}$$

Syntactic Sugar

$$\begin{array}{rcll}
e \; ; \; e' & \stackrel{\text{def}}{=} & \text{let } x = e \text{ in } e' & x \notin \text{FV}(e') \\
\lambda\overline{x}.e & \stackrel{\text{def}}{=} & \text{let } x_{this} = \text{this in object } \{ & x_{this} \notin \text{FV}(e) \\
& & \quad \text{apply}(\overline{x})\{[x_{this}/\text{this}]e\} & \\
& & \} & \\
e(\overline{e}) & \stackrel{\text{def}}{=} & e.\text{apply}(\overline{e}) & \\
e \leftarrow_f m(\overline{e}) & \stackrel{\text{def}}{=} & \text{let } x_f, x_r = \text{future in} & x_f, x_r \notin \text{FV}(e) \cup \text{FV}(\overline{e}) \\
& & \quad e \leftarrow m_f(\overline{e} \cdot x_r) \; ; \; x_f & \\
\text{when}(e \rightarrow x)\{e'\} & \stackrel{\text{def}}{=} & \text{let } x_f, x_r = \text{future in} & x_f, x_r \notin \text{FV}(e) \cup \text{FV}(e') \\
& & \quad \text{let } x_l = \lambda x.(x_r.\text{resolve}_\mu(e')) \text{ in} & x_l \notin \text{FV}(e) \\
& & \quad e \leftarrow \text{register}_\mu(x_l) \; ; \; x_f & \\
\text{resolve } e\; e' & \stackrel{\text{def}}{=} & \text{let } x_r = e \text{ in} & x_r \notin \text{FV}(e') \\
& & \quad \text{let } x_l = \lambda x.(x_r \leftarrow \text{resolve}_\mu(x)) \text{ in} & x_l \notin \text{FV}(e') \\
& & \quad e' \leftarrow \text{register}_\mu(x_l) &
\end{array}$$

Evaluation Contexts and Runtime Syntax

$$\begin{array}{lcl}
e_\Box & ::= & \Box \mid \text{let } x \; = \; e_\Box \text{ in } e \mid e_\Box.f \mid e_\Box.f := e \mid v.f := e_\Box \mid e_\Box.m(\overline{e}) \mid v.m(\overline{v}, e_\Box, \overline{e}) \\
& \mid & e_\Box \leftarrow m(\overline{e}) \mid v \leftarrow m(\overline{v}, e_\Box, \overline{e}) \\
\\
e & ::= & \ldots \mid r
\end{array}$$

Figure 4.2: AT-LITE Syntax

solve" expressions.

The expression $\text{when}(e \rightarrow x)\{e'\}$ is used to await the value of a future. It is syntactic sugar for registering a "listener" function with the future. The expression as a whole returns a *dependent* future x_f that will become resolved with the expression e' when the future denoted by e eventually resolves.

The expression $\text{resolve } e\; e'$ is used to resolve a future with a value, where e must

Substitution Rules

$$\begin{array}{rcl@{\qquad}rcl}
[v/x]x' & = & x' & [v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{e\} \text{ if } x \in \overline{x} \\
[v/x]x & = & v & [v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{[v/x]e\} \text{ if } x \notin \overline{x} \\
[v/x]e.f & = & ([v/x]e).f & [v/x]e.f := e & = & ([v/x]e).f := [v/x]e \\
[v/x]\text{null} & = & \text{null} & [v/x]e.m(\overline{e}) & = & ()[v/x]e).m([v/x]\overline{e}) \\
[v/x]r & = & r & [v/x]e \leftarrow m(\overline{e}) & = & ([v/x]e) \leftarrow m([v/x]\overline{e})
\end{array}$$

$$\begin{array}{rcl}
[v/x]\text{let } x' = e \text{ in } e & = & \text{let } x' = [v/x]e \text{ in } [v/x]e \\
[v/x]\text{let } x = e \text{ in } e & = & \text{let } x = [v/x]e \text{ in } e \\
[v/x]\text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
[v/x]\text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
[v/x]\text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := [v/x]e}, \overline{[v/x]m(\overline{x})\{e\}}\} \text{ if } x \neq \text{this} \\
[v/\text{this}]\text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\} \\
[v/x]\text{let } x_f, x_r = \text{future in } e & = & \text{let } x_f, x_r = \text{future in } [v/x]e \\
[v/x]\text{let } x, x_r = \text{future in } e & = & \text{let } x, x_r = \text{future in } e \\
[v/x]\text{let } x_f, x = \text{future in } e & = & \text{let } x_f, x = \text{future in } e
\end{array}$$

Figure 4.3: Substitution rules: x denotes a variable name or the pseudovariable this.

reduce to a resolver and e' to any value. If e' reduces to a non-future value, the listener function x_l will be called with x bound to the value of e'. If e' reduces to a future value, the listener function will be called later, with x bound to the resolved value of the future. Thus, this definition ensures that futures can only be truly resolved with non-future values.

Evaluation Contexts and Runtime Expressions We use evaluation contexts [FH92] to indicate the subexpressions of an expression that have to be fully reduced before the compound expression itself can be further reduced. $e_\Box$ denotes an expression with a "hole". Each appearance of $e_\Box$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression.

Our reduction rules operate on "runtime expressions", which are simply all expressions including references r, as a subexpression may reduce to a reference before being reduced further.

4.1.2 Reduction Rules

Notation Actor heaps O are sets of objects, resolvers and futures. To extract values from a set O, we use the notation $O = O' \cup \{o\}$. This splits the set O into a singleton set containing the desired object o and the disjoint set $O' = O \setminus \{o\}$. The notation $Q = Q' \cdot \text{m}$ deconstructs a sequence Q into a subsequence Q' and the last element m. In AT-LITE, queues are sequences of messages and are processed right-to-left, meaning that the last message in the sequence is the first to be processed. We denote both the empty set and the empty sequence using $\emptyset$. The notation $e_\square[e]$ indicates that the expression e is part of a compound expression $e_\square$, and should be reduced first before the compound expression can be reduced further. In our semantics we distinguish between local and global reduction rules. Local reduction rules are evaluated in the context of a single actor while global reduction rules affect more than one actor.

Actor-local reductions Actors operate by perpetually taking the next message from their message queue, transforming the message into an appropriate expression to evaluate, and then reducing this expression to a value. When the expression is fully reduced, the next message is processed. As discussed previously, the process of reducing such a single expression to a value is called a *turn*. It is not possible to suspend a turn and start processing a next message in the middle of a reduction.

If no reduction rule is applicable to further reduce a reducible expression, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its message queue is empty, and its current expression is fully reduced to a value. The actor then sits idle until it receives a new message.

In order to keep the semantics concise we make extensive use of auxiliary functions, as shown in Figure 4.6. We now summarize the actor-local reduction rules in Figure 4.4:

- LET: a "let"-expression simply substitutes the value v for x in e according to the substitution rules outlined in Figure 4.3.

- NEW-OBJECT, NEW-ISOLATE: these rules are identical except for the tag of the fresh object, which is set to O for objects and I for isolates. The effect of evaluating an object or literal expression is the addition of a new object to the actor's heap. The fields of the new object are initialised to `null`. The literal expression reduces to a sequence of field update expressions. The `this` pseudovariable within these field update expressions refers to the new object. The last expression in the reduced sequence is a reference r to the new object.

- INVOKE: a method invocation simply looks up the method m in the receiver object and reduces the method body expression e with appropriate values for the parameters $\overline{x}$ and the pseudovariable `this`. Method invocations are *only* possible on *local* objects (the receiver's global component ι_a must match that of the current actor).

- FIELD-ACCESS, FIELD-UPDATE: a field update modifies the actor's heap such that it contains an object with the same address but with an updated set of fields. Again, field access and field update apply only to *local* objects.

- MAKE-FUTURE: a new future-resolver pair is created such that the future has an empty queue and is unresolved (its value is ϵ), and the resolver contains the future's identity ι_f. The expression e is further reduced with x_f and x_r bound to references to the new future and resolver respectively.

- LOCAL-ASYNCHRONOUS-SEND: an asynchronous message sent to a *local* object (i.e. an object owned by the same actor as the sender) simply appends a new message to the end of the actor's own message queue. The message send itself immediately reduces to `null`.

- PROCESS-MESSAGE: this rule describes the processing of incoming asynchronous messages directed at local objects or resolvers (but not futures). A new message can be processed only if two conditions are satisfied: the actor's queue Q must not be empty, and its current expression cannot be reduced any further (the expression is a value v). The auxiliary function $process$ shown in Figure 4.6 distinguishes between:
  - a regular message m (or the meta-level message resolve_μ), which is processed by invoking the corresponding method on the receiver object.
  - a two-way message m_f, as generated by the desugaring of $e \leftarrow_f m(\overline{e})$. Such a message is processed by invoking the corresponding method on the receiver object, and by sending the result of the invocation to the hidden last parameter r, which denotes a resolver object.
  - a meta-level message register_μ, which indicates the registration of a listener function v, to be applied to the value of a resolved future. Since $process$ is only invoked on non-future values $\iota_a.\iota_o$, the listener function v is asynchronously applied to $\iota_a.\iota_o$ directly.
- PROCESS-MSG-TO-FUTURE: this rule describes the processing of incoming asynchronous messages directed at local futures. The processing of the message depends on the state of the recipient future, as determined by the auxiliary function $store$ shown in Figure 4.6. This function returns a tuple $(\mathtt{m}, e)$ where $\mathtt{m}$ denotes either a message or the empty sequence, and e denotes either an asynchronous message send expression or `null`. The message $\mathtt{m}$ is then appended to the future's queue, and the actor will continue reducing the expression e. $store$ determines whether to store or forward the message $\mathtt{m}$, depending on the state of the future and the type of message:
  - If the future is unresolved (its value is still ϵ), the message is enqueued and must not be forwarded yet (e is `null`).
  - If the future is resolved and the message name m is not register_μ, the message need not be enqueued ($\mathtt{m}$ is $\emptyset$), but is rather immediately forwarded to the resolved value v.
  - If the future is resolved and the message is register_μ, which indicates a request to register a listener function $\iota_a.\iota_o$ with the future, the function is asynchronously applied to the resolved value v. This request need not be enqueued ($\mathtt{m}$ is $\emptyset$).
- RESOLVE: this rule describes the reduction of the meta-level message resolve_μ, as used in the desugaring of the "when" and "resolve" expressions. This message can only be reduced when directed at a resolver object ι_r whose paired future ι_f is still unresolved (the resolved value of the future is still ϵ). The paired future is updated such that it is resolved with the value v, and its queue Q' is emptied. The messages previously stored in Q' are forwarded, as described by the auxiliary function fwd shown in Figure 4.6. This function generates a sequence of message sends as follows:
  - If the queue is empty, no more messages need to be forwarded and the expression reduces to `null`.

  - If the queue contains a normal message m (or a meta-level message resolve_μ), that message is forwarded to v.
  - If the queue contains a meta-level message register_μ, indicating the request to notify the listener function $\iota_a.\iota_o$ when the future becomes resolved, the function is asynchronously applied with the future's resolved value v.

Actor-global reductions We summarize the actor-global reduction rules in Figure 4.5:

- NEW-ACTOR: when an actor ι_a is reducing an actor literal expression, a new actor $\iota_{a'}$ is added to the configuration K. The new actor's heap consists of a single new object ι_o whose fields and methods are described by the literal expression. As in the rule for NEW-OBJECT, the object's fields are initialized to `null`. The new actor has an empty queue and will, as its first action, initialize the fields of its only object. The actor literal expression itself reduces to a far reference to the new object, allowing the creating actor to communicate further with the newly spawned actor.
- FAR-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a far reference, i.e. a reference whose global component $\iota_{a'}$ differs from that of the current actor ι_a. A new message is appended to the queue of the recipient actor $\iota_{a'}$. The arguments $\overline{v}$ of the message send expression are parameter-passed as described by the auxiliary function $pass$. This function, described in Figure 4.6, returns a set O'' of copied isolate objects that must be added to the recipient's heap and an updated sequence of values $\overline{v'}$ with updated addresses referring to the copied isolates, if any. The full details of the parameter passing semantics are given in the next section. As in the LOCAL-ASYNCHRONOUS-SEND rule, the message send expression itself evaluates to `null`.
- CONGRUENCE: this rule simply connects the local reduction rules to the global reduction rules.

An AT-LITE program e is reduced in an initial configuration containing a single "main" actor $K_{init} = \{\mathcal{A}\langle\iota_a, \emptyset, \emptyset, [\text{null}/\text{this}]e\rangle\}$. The `this`-pseudovariable is bound to `null` to ensure that there is always a `this` variable present in the environment when a top level function is desugared.

Parameter-passing rules The auxiliary function $pass(\iota_a, O, \overline{v}, \iota_{a'})$ describes the rules for parameter-passing the values $\overline{v}$ from actor ι_a to actor $\iota_{a'}$, where O is the heap of the originating actor ι_a.

The parameter-passing rules for AT-LITE values are simple: objects are passed by reference, isolates are passed by copy, and `null` is passed by value. When an isolate is passed by copy, all of its constituent field values are recursively parameter-passed as well.

The auxiliary function $reach(O, \overline{v})$ returns the set of all isolate objects reachable in O starting from the root values $\overline{v}$. The first two cases define base case for the inductive definition. In the third case, an isolate object o is encountered and added to the result. All of o's field values are added to the set of roots, and o itself is removed from the set of objects to consider, so that it is never visited twice. The fourth rule skips all other

(LET)

$$\begin{array}{l}\mathcal{A}\langle \iota_a, O, Q, e_\square[\text{let } x = v \text{ in } e]\rangle \\ \quad \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e_\square[[v/x]e]\rangle\end{array}$$

(NEW-OBJECT)

$$\frac{\begin{array}{c}\iota_o \text{ fresh} \\ o = \mathcal{O}\langle \iota_o, \text{O}, \overline{f := \text{null}}, \overline{m(\overline{x})\{e'\}}\rangle \quad r = \iota_a.\iota_o\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\text{object}\{\overline{f := e}, \overline{m(\overline{x})\{e'\}}\}]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O \cup \{o\}, Q, e_\square[\overline{r.f := [r/\text{this}]e}; r]\rangle\end{array}}$$

(NEW-ISOLATE)

$$\frac{\begin{array}{c}\iota_o \text{ fresh} \\ o = \mathcal{O}\langle \iota_o, \text{I}, \overline{f := \text{null}}, \overline{m(\overline{x})\{e'\}}\rangle \quad r = \iota_a.\iota_o\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\text{isolate}\{\overline{f := e}, \overline{m(\overline{x})\{e'\}}\}]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O \cup \{o\}, Q, e_\square[\overline{r.f := [r/\text{this}]e}; r]\rangle\end{array}}$$

(INVOKE)

$$\frac{\begin{array}{c}\mathcal{O}\langle \iota_o, t, F, M\rangle \in O \\ r = \iota_a.\iota_o \qquad m(\overline{x})\{e\} \in M\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[r.m(\overline{v})]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e_\square[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\end{array}}$$

(FIELD-ACCESS)

$$\frac{\mathcal{O}\langle \iota_o, t, F, M\rangle \in O \qquad f := v \in F}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\iota_a.\iota_o.f]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e_\square[v]\rangle\end{array}}$$

(FIELD-UPDATE)

$$\frac{\begin{array}{c}O = O' \uplus \{\mathcal{O}\langle \iota_o, t, F \uplus \{f := v'\}, M\rangle\} \\ O'' = O' \cup \{\mathcal{O}\langle \iota_o, t, F \cup \{f := v\}, M\rangle\}\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\iota_a.\iota_o.f := v]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O'', Q, e_\square[v]\rangle\end{array}}$$

(MAKE-FUTURE)

$$\frac{\begin{array}{c}\iota_f, \iota_r \text{ fresh} \\ O' = O \cup \{\mathcal{F}\langle \iota_f, \emptyset, \epsilon\rangle, \mathcal{R}\langle \iota_r, \iota_f\rangle\}\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\text{let } x_f, x_r = \text{future in } e]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O', Q, e_\square[[\iota_a.\iota_f/x_f][\iota_a.\iota_r/x_r]e]\rangle\end{array}}$$

(LOCAL-ASYNCHRONOUS-SEND)

$$\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\iota_a.\iota_o \leftarrow m(\overline{v})]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle \cdot Q, e_\square[\text{null}]\rangle\end{array}$$

(PROCESS-MESSAGE)

$$\frac{\begin{array}{c}\iota_o \notin \textbf{FutureId} \\ e = \mathit{process}(\iota_a.\iota_o, m, \overline{v})\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, e\rangle\end{array}}$$

(PROCESS-MSG-TO-FUTURE)

$$\frac{\begin{array}{c}O = O' \uplus \{\mathcal{F}\langle \iota_f, Q', v'\rangle\} \\ (\text{m}, e) = \mathit{store}(m, \overline{v}, v')\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q \cdot \mathcal{M}\langle \iota_a.\iota_f, m, \overline{v}\rangle, v\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O' \cup \{\mathcal{F}\langle \iota_f, \text{m} \cdot Q', v'\rangle\}, Q, e\rangle\end{array}}$$

(RESOLVE)

$$\frac{\mathcal{R}\langle \iota_r, \iota_f\rangle \in O \qquad O = O' \uplus \{\mathcal{F}\langle \iota_f, Q', \epsilon\rangle\} \qquad v \neq \iota_{a'}.\iota_{f'}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\square[\iota_a.\iota_r.\text{resolve}_\mu(v)]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O' \cup \{\mathcal{F}\langle \iota_f, \emptyset, v\rangle\}, Q, e_\square[\mathit{fwd}(v, Q')]\rangle\end{array}}$$

Figure 4.4: Actor-local reduction rules.

(NEW-ACTOR)

$$\frac{\begin{array}{c}\iota_{a'}, \iota_o \text{ fresh} \\ r = \iota_{a'}.\iota_o \qquad a' = \mathcal{A}\langle \iota_{a'}, \{\mathcal{O}\langle \iota_o, \text{o}, \overline{f := \text{null}}, \overline{m(\overline{x})\{e'\}}\rangle\}, \emptyset, \overline{r.f := [r/\text{this}]e}\rangle\end{array}}{K \dot\cup \mathcal{A}\langle \iota_a, O, Q, e_\Box[\text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e'\}}\}]\rangle \rightarrow_k K \cup \mathcal{A}\langle \iota_a, O, Q, e_\Box[r]\rangle \cup a'}$$

(FAR-ASYNCHRONOUS-SEND)

$$\frac{\begin{array}{c}K = K' \dot\cup \mathcal{A}\langle \iota_{a'}, O', Q', e'\rangle \\ (O'', \overline{v}') = pass(\iota_a, O, \overline{v}, \iota_{a'}) \qquad Q'' = \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}'\rangle \cdot Q'\end{array}}{K \dot\cup \mathcal{A}\langle \iota_a, O, Q, e_\Box[\iota_{a'}.\iota_o \leftarrow m(\overline{v})]\rangle \rightarrow_k K' \cup \mathcal{A}\langle \iota_a, O, Q, e_\Box[\text{null}]\rangle \cup \mathcal{A}\langle \iota_{a'}, O' \cup O'', Q'', e'\rangle}$$

(CONGRUENCE)

$$\frac{a \rightarrow_a a'}{K \dot\cup \{a\} \rightarrow_k K \cup \{a'\}}$$

Figure 4.5: Actor-global reduction rules.

values and applies when v is `null`, a far reference $\iota_{a'}.\iota_{o'}$, an object that was already visited ($v = \iota_a.\iota_o, \iota_o \notin O$) or a non-isolate object ($v = \iota_a.\iota_o, \mathcal{O}\langle \iota_o, \text{o}, F, M\rangle \in O$).

The mapping σ simply defines fresh identities for each isolate in O'. The function $pass$ then returns the set of isolates O'_σ which is simply the set O' with all isolates renamed according to σ. The function σ_v replaces references to parameter-passed isolates with references to the fresh copies, and is the identity function for all other values.

4.1.3 Service Discovery

We now extend AT-LITE with the primitives necessary to give the operational semantics of "service discovery", i.e. the ability for objects in different actors to discover one another by means of a publish/subscribe-style mechanism.

We extend AT-LITE actors with a set of exported objects E and a set of import listeners I. We extend values to include type tags θ. Objects can be exported, and callbacks can be registered by means of a type tag. When the type tag of a listener matches the type tag of an exported object in another actor, the callback is applied.

The AT-LITE syntax is extended with syntax to export objects ($\text{export } e\ e$), to register callbacks for discovery ($\text{discover } e\ e$) and syntactic sugar $\text{whenDiscovered}(e \rightarrow x)\{e'\}$.

Figure 4.7 lists the additional reduction rules for service discovery:

- PUBLISH: to reduce an `export` expression, the first argument must be reduced to a type tag θ and the second argument must be reduced to a reference (which may be a far reference). The effect of reducing an `export` expression is that the actor's set of exported objects E is extended to include the exported object and type tag. An exported object is serialized as if it were included in an inter-actor message. Hence, if the object is an isolate, a copy of the isolate is made at the time it is exported.

- SUBSCRIBE: to reduce a `discover` expression, the first argument must be reduced to a type tag θ and the second argument must be reduced to an object

Auxiliary functions and predicates

$$
\begin{array}{rcll}
process(\iota_a.\iota_o, m, \overline{v}) & \overset{def}{=} & \iota_a.\iota_o.m(\overline{v}) & m \neq m_f, m \neq \text{register}_\mu \\
process(\iota_a.\iota_o, m_f, \overline{v} \cdot r) & \overset{def}{=} & r \leftarrow \text{resolve}_\mu(\iota_a.\iota_o.m(\overline{v})) & \\
process(\iota_a.\iota_o, \text{register}_\mu, v) & \overset{def}{=} & v \leftarrow \text{apply}(\iota_a.\iota_o) & \\
 & & & \\
store(m, \overline{v}, \epsilon) & \overset{def}{=} & (\mathcal{M}\langle \epsilon, m, \overline{v}\rangle, \text{null}) & \\
store(m, \overline{v}, v) & \overset{def}{=} & (\emptyset, v \leftarrow m(\overline{v})) & m \neq \text{register}_\mu, v \neq \epsilon \\
store(m, \iota_a.\iota_o, v) & \overset{def}{=} & (\emptyset, \iota_a.\iota_o \leftarrow \text{apply}(v)) & m = \text{register}_\mu, v \neq \epsilon \\
 & & & \\
fwd(v, \emptyset) & \overset{def}{=} & \text{null} & \\
fwd(v, Q \cdot \mathcal{M}\langle \epsilon, m, \overline{v}\rangle) & \overset{def}{=} & v \leftarrow m(\overline{v})\ ;\ fwd(v, Q) & m \neq \text{register}_\mu \\
fwd(v, Q \cdot \mathcal{M}\langle \epsilon, m, \iota_a.\iota_o\rangle) & \overset{def}{=} & \iota_a.\iota_o \leftarrow \text{apply}(v)\ ;\ fwd(v, Q) & m = \text{register}_\mu \\
 & & & \\
reach(\emptyset, \overline{v}) & \overset{def}{=} & \emptyset & \\
reach(O, \emptyset) & \overset{def}{=} & \emptyset & \\
reach(O \uplus o, \overline{v} \cdot \iota_a.\iota_o) & \overset{def}{=} & reach(O, \overline{v} \cdot \overline{v}') \cup \{o\} & \text{if } o = \mathcal{O}\langle \iota_o, \text{I}, \overline{f := v'}, M\rangle \\
reach(O, \overline{v} \cdot v) & \overset{def}{=} & reach(O, \overline{v}) & \text{otherwise} \\
 & & & \\
pass(\iota_a, O, \overline{v}, \iota'_a) & \overset{def}{=} & (O'_\sigma, \sigma_v(\overline{v})) & \\
 & & \text{where } O' = reach(O, \overline{v}) & \\
 & & \sigma = \{\iota_o \mapsto \iota'_o \mid \mathcal{O}\langle \iota_o, t, F, M\rangle \in O', \iota'_o \text{ fresh }\} & \\
 & & O'_\sigma = \{\mathcal{O}\langle \sigma(\iota_o), \text{I}, \overline{f := \sigma_v(v)}, M\rangle \mid \mathcal{O}\langle \iota_o, \text{I}, \overline{f := v}, M\rangle \in O'\} & \\
 & & \sigma_v(v) = \left\{ \begin{array}{ll} \iota'_a.\iota'_o & \text{if } v = \iota_a.\iota_o,\ \iota_o \mapsto \iota'_o \in \sigma \\ v & \text{otherwise} \end{array} \right. &
\end{array}
$$

Figure 4.6: Auxiliary functions and predicates

reference. The effect of reducing a `discover` expression is that the actor's set of import listeners I is extended to include the local callback, and the type tag.

- MATCH: this rule is applicable when a configuration of actors contains both an actor $\iota_{a'}$ that exports an object under a type tag θ, and a different actor ι_a that has registered a listener under the same type tag θ. The effect of service discovery is that an asynchronous `apply` message will be sent to the registered listener object in ι_a. The listener is simultaneously removed from the import set of its actor so that it can be notified at most once. The exported object v is parameter-passed again, this time to copy it from the publication actor $\iota_{a'}$ to the subscription actor ι_a.

Extensions for Service Discovery

Semantic Entities

$$\begin{array}{rcl} a \in A \subseteq \mathbf{Actor} & ::= & \mathcal{A}\langle \iota_a, O, Q, E, I, e\rangle \\ v \in \mathbf{Value} & ::= & \ldots \mid \theta \\ \theta \in \text{TypeTags} & & \end{array}$$

Syntax

$$e \quad ::= \quad \ldots \mid \text{export } e\ e \mid \text{discover } e\ e \mid \text{whenDiscovered}(e \rightarrow x)\{e\}$$

Evaluation Contexts

$$e_\square \quad ::= \quad \ldots \mid \text{export } e_\square\ e \mid \text{export } v\ e_\square \mid \text{discover } e_\square\ e \mid \text{discover } v\ e_\square$$

Syntactic Sugar

$$\text{whenDiscovered}(e \rightarrow x)\{e'\} \quad \overset{\text{def}}{=} \quad \text{discover } e\ (\lambda x.e')$$

(PUBLISH)

$$\frac{(O', v') = pass(\iota_a, O, \iota_{a'}.\iota_o, \iota_a)}{\begin{array}{l}\mathcal{A}\langle \iota_a, O, Q, E, I, e_\square[\text{export } \theta\ \iota_{a'}.\iota_o]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, E \cup (O', v', \theta), I, e_\square[\text{null}]\rangle\end{array}}$$

(SUBSCRIBE)

$$\begin{array}{l}\mathcal{A}\langle \iota_a, O, Q, E, I, e_\square[\text{discover } \theta\ \iota_a.\iota_o)]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, E, I \cup (\iota_a.\iota_o, \theta), e_\square[\text{null}]\rangle\end{array}$$

(MATCH)

$$\frac{\begin{array}{c}\mathcal{A}\langle \iota_{a'}, O', Q', E' \uplus (O'', v, \theta), I', e'\rangle \in K \\ (O''', v') = pass(\iota_{a'}, O'', v, \iota_a) \qquad Q'' = \mathcal{M}\langle \iota_a.\iota_o, \text{apply}, v'\rangle \cdot Q\end{array}}{K \uplus \mathcal{A}\langle \iota_a, O, Q, E, I \uplus (\iota_a.\iota_o, \theta), e\rangle \rightarrow_k K \cup \mathcal{A}\langle \iota_a, O \cup O''', Q'', E, I, e\rangle}$$

Figure 4.7: Reduction rules for service discovery

4.1.4 Robust time-decoupled message transmission

In the calculus presented so far, actors are assumed to be permanently connected to all other actors. The real world, however, shows that devices almost always reside in separate networks and only occasionally meet to exchange messages. In this extension to the calculus, we introduce *networks* that completely isolate their actors from other networks but still allow full communication between actors in the network. Each network has an identifier, which we shall model as a natural number.

Isolating actors from other actors introduces a problem: how can they communicate? Over time actors will move about and join other networks, opening up new message transmission opportunities. We formalize this by splitting the message-sending process into two parts: message creation and message transmission. Whenever an actor executes the $\leftarrow$ operator, the message is created and stored in a message outbox (called Q_{out}), to be transmitted at a later stage. We call this *time-decoupled message transmission*, as actors do not have to be connected to each other to create asynchronous messages.

Extensions for time-decoupled message transmission

Semantic Entities

$$\begin{array}{rcl} a \in A \subseteq \mathbf{Actor} & ::= & \mathcal{A}\langle \iota_a, O, Q, Q_{out}, n, e\rangle \\ n \in \mathbf{Network} & ::= & \mathbb{N} \\ \mathrm{m} \in \mathbf{Message} & ::= & \mathcal{M}\langle v, m, \overline{v}, O\rangle \end{array}$$

In the reduction rules, we replace asynchonous message sends (FAR-ASYNCHRONOUS-SEND) by rules for message creation (CREATE-MESSAGE) and message transmission (TRANSMIT-MESSAGE).

(FAR-ASYNCHRONOUS-SEND)
This rule is removed.

(PROCESS-MESSAGE)

$$\frac{\iota_o \notin \mathbf{FutureId} \qquad e = process(\iota_a.\iota_o, m, \overline{v})}{\mathcal{A}\langle \iota_a, O, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}, O'\rangle, Q_{out}, n, v\rangle \rightarrow_a \mathcal{A}\langle \iota_a, O \cup O', Q, Q_{out}, n, e\rangle}$$

(CREATE-MESSAGE)

$$\frac{(O', \overline{v}') = pass(\iota_a, O, \overline{v}, \iota_{a'}) \qquad \mathrm{m} = \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}', O'\rangle}{K \uplus \mathcal{A}\langle \iota_a, O, Q, Q_{out}, n, e_\square[\iota_{a'}.\iota_o \leftarrow m(\overline{v})]\rangle \rightarrow_a K \cup \mathcal{A}\langle \iota_a, O, Q, \mathrm{m} \cdot Q_{out}, n, e_\square[\mathrm{null}]\rangle}$$

(TRANSMIT-MESSAGE)

$$\frac{\exists \text{ first } \mathrm{m} \in Q_{out} : \mathrm{m} = \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}, O_m\rangle \qquad K = K' \uplus \mathcal{A}\langle \iota_{a'}, O', Q', Q'_{out}, n, e'\rangle}{K \uplus \mathcal{A}\langle \iota_a, O, Q, Q_{out}, n, e\rangle \rightarrow_k K' \cup \mathcal{A}\langle \iota_a, O, Q, Q_{out} \setminus \mathrm{m}, n, e\rangle \cup \mathcal{A}\langle \iota_{a'}, O', \mathrm{m} \cdot Q', Q'_{out}, n, e'\rangle}$$

Figure 4.8: Reduction rules for time-decoupled message transmission

Figure 4.8 lists the additional reduction rules for time-decoupled message transmission:

- CREATE-MESSAGE: This rule creates a message object and appends it to the outbox Q_{out}. It also stores a copy of the object graph returned by $pass$ into the message object. This must be done at message creation time, as the objects in this object graph might change between message creation and message transmission time. This rule is actor-local, so it can be invoked regardless of whether the actors are in the same network.

- PROCESS-MESSAGE: This rule is adapted to import the objects contained in the message into the actor's object heap.
- TRANSMIT-MESSAGE: This rule can fire whenever an actor is in the same network for which the actor has undelivered messages. If this is the case, this rule will extract the first of these undelivered messages, append it to the destination actor's message inbox and remove it from the outbox.

This extension to the rules diminishes the guarantees AT-LITE gives about message ordering. Assume the following scenario: actor A sends a message to actor C at time t_A and an actor B also sends a message to actor C at time T_B (assume $t_A < t_B$). Without the extension, actor C will process the message from A first, then the message from B. With the extension, the ordering depends not on the time of message *creation*, but message *reception*. The ordering of sequences of messages between two given actors is still maintained, as messages are transmitted in a FIFO manner.

4.2 Conclusion

We have presented an operational semantics for a key subset of the AmbientTalk programming language under the form of a calculus for communicating event loops. The operational semantics provides a formal account of AmbientTalk actors as communicating event loops, objects, isolates, futures, asynchronous message sends and service discovery. With the communicating event loop calculus we have given a formal overview of the event loop concurrency model in which we have formulated ambient contracts. This event-loop concurrency model forms the basis of the concurrency and distribution model that adheres to the AmOP criteria presented in chapter 2.

Novel about our semantics is the operational description of *non-blocking futures*. A future is a first-class value that acts as the placeholder for a value that is asynchronously awaited. While the future is unresolved, any messages sent to the future are queued. When the future becomes resolved, all messages in the queue are forwarded to the resolved value and the queue is emptied. In the semantics we give a formal account of future pipelining and show how futures can be used in a distributed environment. Our semantics also includes the primitives necessary for service discovery, i.e. the ability for objects in different actors to discover one another by means of a publish/subscribe-style mechanism. Finally, our semantics shows how to encode robust time-decoupled message transmission. The semantics of the communication event loop calculus has been validated by using the PLT-Redex [FFF09] tool.

Chapter 5

Computational Contracts for Functions

In this dissertation, we propose the use of ambient contracts in order to specify the behavior of a group of distributed objects communicating over a volatile network. In this chapter, we present *computational contracts*; a first step towards the development of ambient contracts. Computational contracts allow the specification of behavioral constraints such as: does a module write certain values to a file? does a module send out appropriate messages? etc.

The aim of contracts is to specify and validate well-defined properties over the values that are exchanged between the different modules of a system. As explained in chapter 3, Beugnard et al. [BJPW99] categorize contract systems in four levels: syntactic (type systems), behavioral contracts (pre/post conditions), temporal contracts (temporal ordering, time based synchronization) and quality of service contracts (e.g. time and space guarantees). In this chapter we focus on *runtime-validation of temporal contracts* for dynamically-typed languages. How computational contracts can be applied to objects and how they deal with distribution, is the subject of chapter 6 and chapter 7.

We have found that most contract systems do not provide mechanisms to check temporal aspects in a *higher-order setting*. A very simple example of the lack of expressiveness of current higher-order contract systems can be observed when implementing a temporal contract that disallows a function to write to a file. For example, with the use of dependent contracts as shown in section 3.2.1.1, it would require the programmer to manually save the state of the file before executing the contracted function and later validate that the state of the file has not changed. While this functionality on its own requires a substantial amount of work, other functions might open, write, and close files in the system concurrently. These writes are possibly allowed and thus should be ignored when validating the postcondition. Even more importantly, checking this particular contract in the (post-condition) is too late; i.e. the damage has already been done. Finally, most contract systems do not provide abstractions for the programmer to easily detect that a file was read.

In this chapter we propose the use of computational contracts in order to overcome the limitations of current contract systems. The explanation given in this chapter is a heavily revised version of our computational contracts for Scheme as explained in [STM11]. Computational contracts are an extension to the higher-order contract systems defined by Findler and Felleisen (2002) [FF02] with the ability to specify temporal constraints. Computational contracts allow the programmer to define and check contracts over *a function* or *an object* and its (possibly) *higher-order arguments*.

A computational contract C_c is denoted over a function as follows, $C_d \xrightarrow{C_c} C_r$, where C_d and C_r are the contracts specified over the the domain and range of the function. The computational contract is active during the execution of the function over which it is defined. During the execution of the contracted function the contract system validates that certain events should or should not happen. In order to intercept these events, computational contracts make use of scoped aspects [DTK06].

The focus of this chapter is to define the concept of computational contracts for higher-order functions. However, as our contract system also supports flat and higher-order contracts we also explain these concepts in detail. The main contract systems explained in this chapter are outlined in the design space of ambient contracts as shown in figure 5.1. Green squares indicate contracts that have been adopted by related work but will be explained in detail. The yellow square shows the novel contract system presented in this chapter and red squares indicate contracts that are not discussed here.

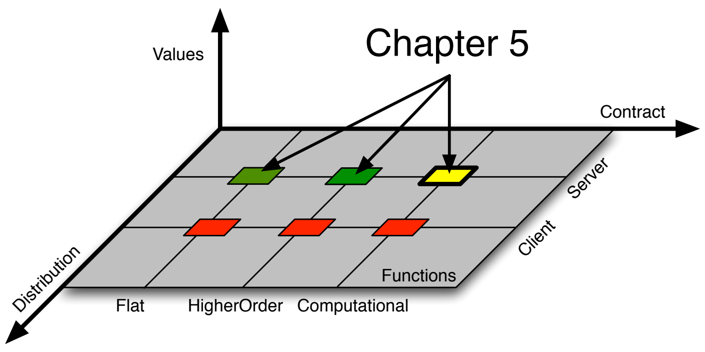


Figure 5.1: Situation of Chapter 5 in the ambient contract design space for functions.

Concretely, in this chapter we present:

- An expressive mechanism to specify and validate computational contracts based on scoped aspects.
- Various examples of computational contracts including permission contracts and protocol contracts.
- A concrete implementation with a mechanism to determine to whom to assign blame in case of a violation.
- The impact of computational contracts on other language constructs such as function identity.
- An executable operational semantics of computational contracts.

We have implemented the existing higher-order contracts (Section 5.1) from Findler and Felleisen in AmbientTalk and we extended them so that they support the specification of computational contracts (Section 5.2). We present various examples of computational contracts including mandatory function calls and protocol contracts (Section 5.2). We describe an expressive model to specify and validate higher-order computational contracts, including proper blame assignment (Section 5.3). We discuss interactions between computational contracts and existing contracts in Section 5.5.

5.1 Higher-Order Contracts in a Nutshell

Computational contracts are based upon Findler and Felleisen seminal work on higher-order pre/post contracts [FF02]. Recall from from section 3.2.1.1 that their system differentiates between contracts defined over simple values, called *flat contracts*, and contracts defined over functions dubbed *function contracts*. Function contracts are of the form $C_d \rightarrow C_r$ where C_d is a contract over the domain of the function and C_r is a contract defined over the range of the function. C_d and C_r can be either flat or function contracts as the contract system supports higher-order pre/post contracts. In the following sections we show examples of function contracts ($C_d \rightarrow C_r$) where C_d and C_r are flat contracts followed by an example where C_d is a function contract.

5.1.1 First-Order Function Contracts

The prototypical example of contract frameworks is to define a contract over the `sqrt` function. The purpose of the contract is to ensure that the argument passed to the `sqrt` function is a positive number (pre-condition) and that the result of the `sqrt` function is also a positive number (post-condition). A possible specification of this contract in AmbientTalk is shown in Figure 5.2. The AmbientTalk module system supports exporting functions and at the same time defining a contract over them by using **`provide :withContract:`**. In the example, the `MathModule` provides (i.e. exports) the function `sqrt` with a function contract that is composed out of two flat contracts. The `positive` contract is constructed by making use of the **`flat:`** contract constructor which, given a predicate function, returns a flat contract. The flat contract over the domain of the `sqrt` function `positive` verifies that the arguments passed to the `sqrt` function is positive (left from the arrow). Similarly the result of the function (after the arrow) must also pass the same flat contract. These flat contracts are composed into a function contract using the arrow operator (->).

Modern contract systems have a mechanism that allows the responsible party to be blamed in case a contract violation is detected. The process of pointing out the responsible party that violated the contract is called *blame assignment* [FF02]. For flat contracts, if the pre-conditions are violated the caller is blamed, if the post-condition is violated the callee is blamed (the `MathModule` in our example). An example of using the contracted `sqrt` function provided by the `MathModule` is shown in Figure 5.3. This figure shows the AmbientTalk interaction prompt. Lines starting with a single larger-than symbol (>) are typed in by the programmer. Lines starting with two larger-than

```
def MathModule := {
  def positive := flat: {|x| x > 0};
  def sqrt(x) { ... };
  def moduleInterface := object: {
    def sqrt := provide: sqrt withContract:  positive -> positive;
  };
};
```

Figure 5.2: Example math module providing a contracted `sqrt` function.

```
1  > import MathModule;
2  >>nil
3  > sqrt("wrong");
4  >>(1:1:REPL)  sqrt("wrong") violated the contract, expected positive given "wrong"
```

Figure 5.3: The AmbientTalk prompt: signaling a contract violation.

```
def MathModule := {
  def map_pos(f, a) {
    a.map: f;
  };
  def moduleInterface := object: {
    def map_pos := provide: map_pos withContract:
    (positive -> positive) * arrayOf(positive) -> arrayOf(positive);
  };
};
```

Figure 5.4: Higher-order pre/post contract over the function map_pos.

symbols (>>) are the return values of evaluating the previous line. In our example, the math module is imported (line 1), then the sqrt function is applied to the string "wrong" (line 3). The contract system verifies the precondition and assigns blame to the read-eval-print loop (1:1:REPL) as expected (line 4).

For a long time researchers agreed that assigning blame was simply a matter of determining where the violation took place. If pre-conditions are violated it is the callers fault, if post-conditions are violated it is the callee's fault (the sqrt function in our example). However, in the context of dynamic languages (such as Ruby, Python or Scheme) the use of higher-order functions makes blame assignment challenging. In the next sections, we show why higher-order values complicate blame assign.

5.1.2 Higher-Order Pre/Post Contracts

Higher-order contracts define contracts over higher-order functions, i.e. functions that receive other functions as an argument or return functions. A prototypical example of a higher-order function is the map function. The map function expects a function f:a->b and an array of elements a. The map function when applied creates a new array of elements b by applying the function f on each element of the array given as an argument. Figure 5.4 shows a higher-order contract that can be used to define a contract over a variation of the map function. This code excerpt specifies that the first argument of the map_pos function is a function that should be applied to a positive number and should return a positive number. The second argument of the map_pos function should be an array of positive numbers. Finally, the return value of the map_pos function should also be an array of positive numbers. Given this contract the contract system tracks the use of the contracted function map_pos and assigns blame in case a violation is detected.

As an example of the use of the map_pos function consider Figure 5.5. In this example the map_pos function is first applied correctly. Then the map function is applied again, however this time the function passed as an argument does not produce integer values. This is clearly a violation of the contract as the contract stipulates that the function map_pos should only be applied to functions that return integer values. While the map function is being evaluated the contract system detects this violation and correctly

```
>map_pos( {|x| x+1 }, [1,2,3])
>>[2, 3, 4]
>map_pos( {|x| "wrong" }, [1,2,3])
```

```
>> (1:1:REPL)  map_pos({ |x| "wrong"}, [1, 2, 3])  violated the contract,
expected positive given "wrong"
```

Figure 5.5: The AmbientTalk prompt: signaling a higher-order contract violation.

assigns blame to the caller of the `map` function, in this case the `REPL`.

Assigning blame in case of higher-order contracts is not as straightforward as with flat-contracts. Predicates defined over functions are in general undecidable, and validating them before starting the execution of the function under contract is in general impossible [FF02]. In the example, at the moment that the `map_pos` function is applied to another function it is in general undecidable to know whether this function behaves according to the specified contract. *Instead the verification of a higher-order contract is postponed by the contract system until the functional arguments under contract are used during the execution of the contracted function.* This is in contrast to flat contract systems, which are only active before and after the function is applied. Therefore, simple flat contract systems can not check higher-order contracts. Similar to how simple flat contract systems can not express contracts over functional argument or return values, higher-order contracts can not express behavioral constraints over values. How to specify behavioral constraints over values is the first contribution of this chapter and is explained in the following section.

5.2 Computational Contracts

A computational contract is a higher-order contract over the *execution* of a contracted entity. Computational contracts are applicable over functions, objects and their higher-order arguments. In contrast to traditional higher-order contracts, they are not restricted to only validating the *interface* of an applicable value, but they can also express assertions over the *computation* that is associated with that value. Programmers can specify what has to happen or what should not happen during the execution of the computation.

As an example of a high-level computational contract consider again the `sqrt` function shown in Section 5.1. This time, we want to express that the `sqrt` function should not display anything to the user. With computational contracts this behavior can be enforced by specifying a contract over the `sqrt` function, for example $\texttt{positive} \xrightarrow{!call(system.println)} \texttt{positive}$. The contract defined over the `sqrt` function again specifies that the argument has to be a positive number and that the return value has to be positive. In addition, the computational contract denoted by $!call(system.println)$ disallows all invocations of `system.println` during the execution of the `sqrt` function. The computational contract is active during the dynamic extent of the function application. Internally, the computational contract consist of two parts. First, the *interception component* is a description of *when* validation of the function under contract is needed, in our example when the method `system.println` is invoked. Second, the *blame assignment component* is applied when the interception component intercepts a potentially disallowed event. In general when a blame assignment component is applied it can decide to assign blame, update internal state and/or proceed with the computation, etc. In later sections, we show that the computational contract system

	A single function application	Sequence of function applications
Mandatory	`ensure_c(function)`	`ensure_c(protocol)`
Disallow	`prohibit_c(function)`	`prohibit_c(protocol)`

Table 5.1: Overview of the high-level functional computational contracts provided in AmbientTalk.

provides high-level abstractions, which make it easy for the end programmer to specify temporal constraints.

Prohibition of certain function applications within the dynamic extent of a function application is only one type of computational contract. Specifying that a certain function application is mandatory or that a sequence of function calls should satisfy a certain protocol are other examples that are directly supported by our computational contract system. To make it easy for the end programmer, all these variations can be expressed by two functions `ensure_c` and `prohibit_c`. These functions create basic computational contracts. Table 5.1 gives an overview of the functionality of these contract constructor functions. Depending on their arguments they either create a computational contract that prohibits or ensures that a certain function is applied or that a protocol is followed during the execution of the contracted function. How to make such protocols is shown in section 5.2.3. In the next sections we show how to define and use high-level computational contracts. In the rest of this section we do not make the distinction between high-level and basic computational contracts and just use the term "computational contracts".

Module Examples Recall that contracts are defined on the module boundaries, i.e. when a function is provided by a module for use by another module. The definition of a module always follows the same pattern. Consider again the listing shown in figure 5.2. In most cases to explain our contracts, the only interesting line of the module is line 5. Consequently, in the rest of our examples, instead of showing the entire listing 5.2 we only show line 5. As it would become confusing to know which module we are referring to, all definitions are assumed to be defined in a module called `"defs.at"`. These functions then are imported and applied from a different module called `"uses.at"`.

5.2.1 Prohibit Contracts

Prohibiting a function call with computational contracts is done by generating a contract with the function `prohibit_c`. In order to create a computational contract that prohibits a function to be applied during the execution of a contracted function, the developer only has to pass the function that is disallowed to the `prohibit_c` function. I.e. `prohibit_c(f)` prohibits the function `f` to be applied within the dynamic extent of the contracted function. There are two main mechanisms active during the enforcement of a prohibit contract: the interception mechanism and the blame assignment mechanism. The interception mechanism intercepts all applications of the prohibited function *before* they are actually applied. At that moment, control is transferred to the blame assignment mechanism which stops the entire computation and presents the programmer with an error message explaining who violated the prohibit contract.

Note that computational contracts are defined over values, not over variables. In our conceptual example `f` denotes applications of the function that the variable `f` refers to when the contract is created. When the variable `f` is destructively changed to point to another function g after defining the contract, applications of the function g will *not* be intercepted.

Figure 5.6 shows a concrete example of how the prohibit_c function can be used to contract exported functions. In this example, the function sqrt is exported with the prohibit contract that ensures that the function system.println is not applied. The prohibit contract assigns blame to the function sqrt whenever the function system.println is applied in the dynamic extent of the sqrt function.

Note that figure 5.6 is not pseudo code. The arrow notation of the computational contracts is implemented by overriding the minus - operator and the minus greater than operator (->) for contracts. The - operator binds a domain contract to a computational contract and returns a new contract. This new contract can then be bound to the range contract by using the -> operator.

```
def sqrt_c := provide: sqrt withContract: positive -prohibit_c(system.println)-> positive;
```

Figure 5.6: Using a prohibit contract to prevent the sqrt to display text.

The same prohibit_c contract constructor can also be used to contract functional arguments. Let us revisit the example of the map_pos function from Section 5.1.2 and add a prohibit contract over the provided function as shown in Figure 5.7. In this contract the ∗ is used to indicate that the map function receives two arguments. When

```
def map_pos := provide: map_pos withContract:
(positive -prohibit_c(system.println)-> positive) * arrayOf(positive) -> arrayOf(positive);
```

Figure 5.7: Defining a prohibit contract over the argument of the map_pos function.

using the contracted map_pos function correctly it behaves like any other function. Figure 5.8 shows a transcript where the function map_pos is applied to the increment function and the array [1,2,3]. As expected, the result of this function is the array [2,3,4].

```
1  Interactive AmbientTalk Shell, version 2.19 Contracts
2  >import ~/.defs;
3  >>nil
4  >map_pos({|x| x+1}, [1,2,3]);
5  >>[2, 3, 4]
6  > map_pos( { |x| system.println(x); x+1; }, [1,2,3] )
```

```
7  1:8:uses.at violated the contract prohibit_c(system.println)
8  computational contract violation
9  origin:
10 at system.println(x)  (1:16:REPL)
11 at a.map:(f)  (44:17:defs.at)
12 at map_pos({ |x| system.println(x); x.+(1)}, [1, 2, 3])  (1:1:REPL)
```

Figure 5.8: Using the contracted map_pos function (from uses.at).

Subsequently (line 6) the map_pos function is applied to a function that violates the computational contract, i.e. applies system.println. Therefore, blame is assigned to the caller of the map_pos function. The resulting error message (line 7 – 12) shows that the violation was caused by the file uses.at. As the transcript was taken from the interaction window of the uses.at module, it can be easily deduced

```
def display_average(a) {
  ((a.inject: 0 into: {|x,y| x+y})/a.length);
};

def moduleInterface := object: {
  def display_average := provide: display_average withContract:
                         arrayOf(pos) -ensure_c(system.println)-> any;
}
```

Figure 5.9: Exporting the function `display-average` with a promise contract.

that the blame is assigned to the call made from the prompt. It is important to note that blame is assigned when the argument function is applied within the body of the `map_pos` function. When `map_pos` is applied it is not possible to determine that the function passed as an argument behaves according to the contract. This is also the main reason why blame assignment is needed in the context of higher-order functions. For a programmer it would not always be clear which module to blame even when he knows that a certain contract is violated during the execution of a function under contract. The contract system keeps track of the responsibilities of the provider and user of the contracted function and assigns blame in case a violation is detected.

The stack trace can be used to pinpoint which line caused the violation of the computational contract (shown after the word `origin:`). It reveals that the origin of the violation was in the read-eval-print-loop on line 1, character 16. Note that the function `system.println` was never applied. The blame assignment of the computational contract stops the current evaluation and prevents the function `system.println` to be applied completely.

5.2.2 Ensure Contracts

The dual of prohibiting an action is to *ensure* that an action is performed. An ensure contract verifies that a certain promise is kept during the dynamic extent of the contracted function. For example, a function `g` can promise to apply another function `f`. An ensure contract does not specify the exact moment when the function `f` needs to be applied; it only specifies that at some moment during the execution of the contracted function `f` needs to be applied. Verifying an ensure contract is thus more subtle than verifying a prohibit contract as blame can only be assigned *after* the contracted function has been entirely executed. After all, the application of the function `f` could be the last statement of the contracted function. From a programmers perspective, defining an ensure contract with computational contracts is as simple as defining a prohibit contract. Ensure contracts are created with the function `ensure_c`.

To illustrate the use of an ensure contract, consider the function `display_average` shown in Figure 5.9. This function takes an array of numbers and displays the average of the numbers in the array. It is exported with an ensure contract that assigns blame when the promise of applying the function `system.println` is not held.

This function `display_average` correctly computes the average of the argument list. Unfortunately, the function under contract does not apply the function `system.println`. Therefore applying the list `[10, 20]` to the function `display_average` leads to a violation of the ensure contract. A transcript of this example is shown below. As highlighted in the transcript the function `display_average` violates the `ensure_c(system.println)` contract.

```
def OpenCloseProtocol() {
  UsageProtocol:  {
    def start()   { closed(); };
    def closed()  { (on: openFile)   => { opened(); }};
    def opened()  { (on: closeFile)  => { end();    }};
    def end()     { (on: any)        => { false;    }};
  };
};
def readCharFromFile(s) {
  openFile(s).readChar();
};

def moduleInterface := object: {
  def readCharFromFile  := provide: readCharFromFile withContract:
  string -ensure_c(OpenCloseProtocol)-> char;
}
```

Figure 5.10: Ensure protocol contract over the `readCharFromFile` function.

```
> display_average( [10,20] );
47:13:defs.at violated the contract ensure_c(system.println)
origin:
at display_average([10, 20])  (1:1:REPL)
```

5.2.3 Usage Protocols

Until now we have only considered computational contracts that are suppose to validate whether or not a certain function is applied once. In this section we show computational contracts that enable the programmer to describe a specification of the order in which certain functions have to be applied. In our implementation these orderings (protocols) are expressed by means of a finite state machine.

An example of a finite state machine is shown at the top of Figure 5.10. This finite state machine describes a protocol that specifies that the contracted function should perform exactly one application of the function `openFile` followed by one application of the function `closeFile`. The mandatory `start` state of the finite state machine indicates that the finite state machine has to be initialized to the state `closed`. Besides the `start` state the finite state machine has three states: `closed`, `opened`, `end`. In each of these states there is one possible transition, for example, when the finite state is in the `closed` state, function invocations to the function `openFile` will transition the finite state machine to the `opened` state. Each state can have multiple transitions and are denoted with the following syntax `(on: f)=> B;` where `f` refers to a function and `B` is a code-block which can contain arbitrary AmbientTalk code. The result of evaluating this code block is used as the next state of the finite state machine. In case the returned value is `false`, a contract violation is detected. States can also receive arguments which can be used in the transition code in order to decide to which state to transition next.

Functions that are not specified in the protocol can always be applied. For example the application sequence `openFile`, `system.println`, `closeFile` leads to the `end` state. In the `end` state the special wildcard qualifier `any` is used. This wildcard matches any function mentioned in the usage protocol. Therefore, calling a function mentioned in the protocol while in the `end` state results in a violation of the protocol. When an application sequence does not follow the protocol, blame is assigned. For example, the application sequence `openFile`, `closeFile`, `openFile` is not allowed as the `end` state does not allow any applications to `openFile`.

In the example protocol, sequentiality of opening and closing files is ensured. Protocols are not limited to deterministic sequential sequences. They can also express non-deterministic choice, in our implementation this is expressed by adding multiple transitions in a single state.

Ensure Protocols Once a protocol is defined it can be used to create a computational contract with the `ensure_c` function. The resulting computational contract assigns blame to the contracted function when the internal function applications in the dynamic extent of the contracted function *do not satisfy* the usage protocol. This happens when functions are applied in the wrong order or when the finite state machine is not in the end state when the contracted function returns.

To show the use of the `OpenCloseProtocol` consider the `readCharFromFile` function shown in Figure 5.10. This function opens a file and reads one character from this file (line 10). A version of the `readCharFromFile` function is exported with a contract that ensures that the `OpenCloseProtocol` is followed. Additionally, it is specified that the argument of the function should be a `string` and the return value a `char`.

Applying the exported `readCharFromFile` function results in an error message as shown below. Blame is assigned to the module `defs.at` because the `readCharFromFile` function did not close the file. The error message also shows that the violation was an `ensure_protocol_c` contract violation.

```
> readCharFromFile("testFile.at");
```

```
52:13:defs.at violated the contract ensure_protocol_c
origin:
at readCharFromFile("testFile.at")  (1:1:REPL)
```

Prohibit Protocols A computational contract that prohibits a given protocol can be created with the function `prohibit_c`. Such a computational contract assigns blame to the function over which it is defined whenever the function applications in the dynamic extent of the contracted function satisfy the usage protocol. This happens when all function applications are applied in an order such that the finite state machine does reach the end state when the contracted function returns. To show a use of a prohibit-protocol contract consider the protocol shown in Figure 5.11. This protocol reaches the end-state after exactly two applications of the `createWindow` function. When applying the `prohibit_c` function to this protocol it returns a contract that prohibits a function to create more than one window. As soon as the contracted function creates a second window it violates the prohibit contract because this application sequence leads to the end state of the finite state machine.

The function `readAndShow`, shown in Figure 5.11, expects a filename and shows the content to the user. In order to make sure that this function does not create more than one window it is exported with a prohibit contract: `prohibit_c(window-Protocol)`. Using the function `readAndShow` leads to a violation of the prohibit contract.

In summery, protocols provide the developer with the necessary machinery in order to express certain quality of service contracts. For example, in order to avoid service abuse the programmer can define a protocol that states that a function must be called at least twice and at most five times.

```
def windowProtocol :=
  UsageProtocol: {
    def start() {  checkWindow(1) };
    def checkWindow(x) {
      (on: createWindow) =>  {
        if: ( x == 1) then: { checkWindow(0); } else: { end(); }
      };
   };
  };

def readAndShow(filename) {
   ...
   createWindow( ... );
   createWindow( ... );
   ...
}

def moduleInterface := object: {
  def readAndShow := provide: readAndShow withContract:
                  string -prohibit_c(windowProtocol)-> any);
}
```

Figure 5.11: Defining a prohibit protocol contract over the `readAndShow` function.

5.3 Contract Verification and Blame Assignment

In this section we describe the inner workings of the computational contract system by presenting a didactical implementation [1]. In particular we discuss the algorithm to validate computational contracts and show how to assign blame when a violation is detected. As computational contracts are an extension to higher-order pre/post contracts, we first start with a detailed explanation of the higher-order pre/post contract system as presented in [FB06]. Readers who are already familiar with higher-order pre/post contract systems are still encouraged to read this section in order to get familiar with our notation. After explaining higher-order pre/post contracts, we show how we extend them in order to support computational contracts.

5.3.1 Flat and Higher-Order Pre/Post Contracts

Our higher-order pre/post contract system consists of three AmbientTalk functions, `flat`, `ho` (higer-order) and `guard` (reminiscent to the contract system as presented in [FB06]). These functions are shown in Figure 5.12. The function `flat` consumes a predicate and creates a contract that verifies this predicate. As mentioned before, flat contracts are used to define contracts over simple values. The function `ho` creates a function contract given a contract for the domain and a contract for the range (i.e. the `->` operator). Finally the function `guard` applies contracts over values. The first argument is a contract, created with either `flat` or `ho`. The second argument is the value over which the contract is defined. Finally, the last two arguments, `pos` and `neg` , are *blame labels*, i.e., textual representations of the supplier and consumer of the contracted value. These labels are passed to the contract in order to assign blame in case of a violation. Note that in the actual implementation of a higher-order contract system these blame labels are filled in by the contract system.

In order to get a better understanding of the contract verification mechanism in combination with blame assignment consider the example shown in Figure 5.13. In

[1] The complete code of this didactical implementation is available at `http://soft.vub.ac.be/~cfscholl/index.php?page=at_cc`

```
def flat( pred ) {
  { |pos, neg|
    { |val|
      if: ( pred(val) ) then: {
        val
      } else: { blame(pos); }
    };
  };
};

def ho(dom, rng) {
  { |pos,neg|
    { |val|
      if: ( is: val taggedAs: Closure) then: {
        { |x| rng(pos, neg)( dom(neg,pos)(x) ) }
      }
      else: { blame(pos); }
    }
  }
};

def guard(ctc, val, pos, neg) {
  ctc(pos,neg)(val);
};
```

Figure 5.12: Higher-Order pre/post contract constructors.

```
def fixTen(f) { f(10) == 10 };
def neg(x) { system.println(x);  -x; };

def nat  := flat( {|x| x > 0} );
def bool := flat( {|x| is: x taggedAs: Boolean; });

def cf :=  guard( ho(ho(nat, nat), bool),
                  fixTen,
                  "fixTen",
                  "prompt");
```

Figure 5.13: A higher-order contract definition over the function fixTen.

this example two functions `fixTen` and `neg` are defined. The function `fixTen` takes a function and verifies whether or not `10` is a fix point of that function. The function `neg` returns the negative value of an integer argument. The example also shows the creation of two flat contracts, one with the predicate `{|x| x > 0}` and one with the predicate `{|x| is: x taggedAs: Boolean;}`. Finally a contracted version of `fixTen`, `cf`, is defined by making use of the function `guard`, `ho`, and the newly created `flat` contracts. This function contract states that the argument of `fixTen` is a function that takes a natural number and returns a natural number. The contract also states that the function `fixTen` returns a boolean value.

The function `guard` applies the higher-order pre/post contract over the function `fixTen` given the blame labels. The result is a contract verification function that behaves almost exactly the same as the function `fixTen` with the difference that it verifies the domain ($\texttt{nat} \to \texttt{nat}$) and range (`bool`) contract of its argument.

In our notation, wrapping a function into a contracted function is represented by a box drawn around the function. Wrapping of the function `fixTen` into a contracted function is represented as follows:

$$\boxed{l_1, l_2|\ \texttt{(nat} \to \texttt{nat)} \to \texttt{bool}\ |\ \texttt{fixTen}}$$

where l_1 and l_2 are the blame labels textually representing the supplier and the consumer of the contracted function respectively.

In order to show how blame assignment of contracted functions works consider the following example of applying the contracted version of `fixTen` to the function `neg`:

$$\boxed{l_1, l_2|\ \texttt{(nat} \to \texttt{nat)} \to \texttt{bool?}\ \ |\ \texttt{fixTen}\ }\texttt{(neg)}$$

Here l_1 is the blame label of the supplier of the function `fixTen`, and l_2 is the blame label of the consumer of the function `fixTen`, i.e. the prompt.

When the contracted function `fixTen` is applied it verifies the domain contract, namely that the supplied function `neg` transforms natural numbers into natural numbers. Because this cannot be checked immediately when `fixTen` is applied, the `neg` function is wrapped into a new contracted function. This corresponds to line 15 of the `ho` function shown in Figure 5.12 where the argument `dom` is bound to the higher-order contract ($nat \to nat$). Note that for this new contracted function, the blame labels are swapped. An intuitive explanation for this blame label swapping is that within the function body of the function `fixTen` the prompt is the supplier of the function `neg`, and the contracted function is the client. After wrapping the function `neg` into a contracted function, the original function `fixTen` is applied as shown below:

$$\texttt{fixTen(}\boxed{l_2, l_1|\ \texttt{nat} \to \texttt{nat}\ |\ \texttt{neg}}\texttt{)}$$

As shown in Figure 5.13, within the body of `fixTen` the function `neg` is applied to the value 10. Since the argument is passed to the wrapped function, the domain contract is verified first:

$$\boxed{l_2, l_1|\ \texttt{nat} \to \texttt{nat}\ |\ \texttt{neg}}\texttt{(10)}$$

In this case the argument is a simple value (10) and the domain contract `nat` verifies that this value is indeed a positive integer value. As the supplied value, 10 in the example, passes the domain contract `nat`, the function `neg` is applied and the return value (-10) is verified by the range contract, `nat` again. This value does not pass the flat contract `nat`. Therefore blame is assigned to the prompt, as the prompt was the supplier of the function `neg`, which violated the contract.

In summary, when calling a contracted higher-order function h with a function f, the original higher-order function h calls the contracted version of the function f with blame labels swapped.

5.3.2 Aspect Extensions for Computational Contracts

This explained functional contracts as introduced by Findler and Felleisen [FF02]. We now turn our attention towards extending this system to support computational contracts. An important aspect of computational contracts is how to define and intercept erroneous behaviour during the execution of a computation. Therefore, we have focused our effort on making this interception mechanism as independent from the base level code as possible. To this end we have used aspect-oriented programming [KLM$^+$97]. In this section we describe the aspect-oriented language constructs relevant for our definition of computational contracts. The implementation of the computational contracts itself is shown in section 5.3.3.

In computational contracts, the concepts of pointcut and advice are used to specify at which point the contract has to be verified. By using the techniques established in aspect-oriented programming this can be done without having to make any *intrusive* changes to the contracted entity. In order to achieve the particular behavior of only enforcing the computational contract in the dynamic extent of the function over which it is defined, dynamically-scoped aspects [DTK06] are used. Instead of being statically and globally defined at the beginning of the program, aspects can be deployed over certain parts of the execution of the program.

The join points of an aspect language are highly determined by the underlying programming language and the particular aspect model at hand. In the semantics described here, only function applications are reified as join points. As there are only function applications, join points are represented by the intercepted functions. Sometimes it is important to know which functions were already applied in the past, for example when validating protocol contracts. Therefore, when the programmer defines a join point descriptor, the programmer has access to a stack of join points instead of only the current joint point.

An aspect consists of a pointcut descriptor and a corresponding advice. The advice is executed whenever the pointcut descriptor matches the current join point stack. An aspect descriptor is represented by a predicate function which is applied to a stack of join points. When the aspect descriptor returns `true` the corresponding advice is executed. The programmer can alter the execution of the program by executing other functionality before, after or instead of the intercepted function application. Advice is implemented as a closure that is applied by the underlying system with a function p. This function p allows the programmer to continue with the originally intercepted function application. The advice is expected to return a function that is applied by the underlying system to the arguments of the intercepted function application.

Now that it is clear how an aspect can be defined, there is still the question of how to deploy an aspect and how to limit its scope. Moreover, there also is the question whether the aspect should be active in the static or dynamic scope of evaluation. Full fledged aspect languages have constructs for both dynamic and static aspect deployment [DTK06, Tan08]. As the semantics of computational contracts depends only on dynamically deployed aspects we limit the rest of the explanation to the specification of dynamic aspects. In AmbientTalk/C, `fluid: A deploy: B`, deploys an aspect `A` in the dynamic extent of executing the body `B`.

To exemplify how a programmer can define a pointcut and an advice, consider the code listed in Figure 5.14. In this example, the idea is again to intercept all applications of the `sqrt` function and to validate that the argument is a positive number. The descriptor matches all join points when there is a call to the `sqrt` function (`call(&sqrt)`). The advice receives a function `proceed` that will proceed with the normal execution of the program when applied. The advice must return a function that is applied to the arguments of the intercepted function. When the advice is applied, it verifies that the argument is a positive number. In case that the argument is not a positive number an error is thrown. Otherwise, the proceed function is applied to the original arguments. In line 11 of the code, the aspect is fluidly deployed. In the body of the aspect deployment, an application of the `sqrt` function to the number `-2` will cause the aspect to be triggered. As the argument to the `sqrt` function is smaller than 0 an error is thrown. Note that the body of the `sqrt` function is never applied in this example. The aspect intercepts the call to the `sqrt` function and verifies the arguments *before* the body is executed. As shown in the example, aspects already help in capturing violations. However, writing contracts like this would be cumbersome and error

```
def a := aspect:  call(&sqrt)  advice: { |proceed, arguments|
    if:  ( 0 > arguments[1] ) then: {
      error: "Calling sqrt with an argument less than zero";
    } else: {
      proceed(@arguments);
    };
};

fluid: a deploy: {
  sqrt(-2);
};
```

Figure 5.14: Aspect to validate the argument of the `sqrt` function.

```
def cc(pc, adv, dom, rng) {
  {|pos, neg|
    { |val|
      if: (is: val taggedAs: Closure) then:  {
        { |x|
          def verified :=  dom(neg, pos)(x);
          def adv := adv(pos,neg);
          def asp := aspect: pc advice: { |proceed, args|
            adv(proceed,args);
          };
          def result := nil;
          fluid: asp deploy: {
            result :=  val(verified);
          };
          rng(pos,neg)( result);
        }
      } else: {
        blame(pos);
      };
    };
  };
};
```

Figure 5.15: Computational contract constructor.

prone.

5.3.3 Verification and Blame Assignment of Computational Contracts

Now that we have explained traditional higher-order contracts in section 5.3.1 and the aspect-oriented language features necessary for computational contracts in section 5.3.2 we can show the internal workings of our computational contract system. We first show the didactical implementation of computational contracts followed by a step-by-step example to explain how blame assignment works.

In Figure 5.15, the computational contract constructor function `cc` is shown. This function is used to create a computational contract (i.e. it implements the (-) and -> operators). The first argument of this function is the interception component of the computational contract. The interception component specifies the exact points where the contract needs to be validated. The second argument is the blame assignment component and is almost the same as an advice. The difference is that a blame assignment component also receives blame assignment labels. The two last arguments are the domain contract and the range contract. A computational contract verifies its contracted

```
def prohibit_c(pc) {
  cc(pc,
    {|pos,neg| {|p,a| blame(pos)}},
    flat( {|x| true;}),
    flat( {|x| true;}))
};

def ensure_c(pc) {
  def called := false;
  cc(pc,
    {|pos,neg| {|p,a| called := true; p(a);  }},
    flat( {|x| called := false; true;}),
    flat( {|x| called;} ) );
}
```

Figure 5.16: High-Level Abstraction for Computational Contracts.

```
def cfc := guard(ho( prohibit_c(call(system.println)), nat, nat), bool),
                 fixTen,
                 "fixTen",
                 "Prompt");
```

Figure 5.17: Computational contract implementation in use.

function very similarly to the way higher-order contracts are verified conform section 5.3. First, the domain contract is verified (line 6). Second, the blame assignment component is initialized and an aspect is created (line 7-10). This aspect is used to intercept interesting events in the dynamic extent of the applied contracted function[2]. Subsequently, this aspect is deployed and the function over which the contract is defined is applied (12-14) The blame assignment component is applied when a matching join point is encountered in the dynamic extent of applying the contracted function, as specified in the advice (line 9). *When there is no violation of the computational contract during the execution of the contracted function, the computational contract behaves exactly like a higher-order contract.*

The definition of a prohibit and ensure contracts can now be defined in terms of our general computational contract definition, as is shown in Figure 5.16. A prohibit contract simply assigns blame in case the blame assignment component is applied. An ensure contract initializes a variable $called$ to be $false$ in the precondition and sets this variable to $true$ in the blame assignment component. The postcondition returns this variable and blame is assigned when the variable was not set to $true$.

5.3.4 Computational Contracts: Step by Step Example

In order to show the impact of computational contracts on the execution of a contracted function let us revisit the `fixTen` example shown in Figure 5.13. However, now the contract defined over the *argument* of the `fixTen` function prohibits calls to `system.println` during its execution as shown in Figure 5.17.

In order to show how blame assignment of computational contracts works, consider the following example of applying the contracted function `fixTen` to the function `neg`:

[2]In practice, we have implemented the basic concepts of LAScheme as a mini aspect language for AmbientTalk [Tan10b]. LAScheme is directly based on AspectScheme [DTK06], but integrates a number of improvements, such as execution levels [Tan10a].

$\boxed{l_1, l_2| \ \texttt{(nat} \xrightarrow{!call(system.println)} \texttt{nat)} \rightarrow \texttt{bool} \mid \texttt{fixTen}}\texttt{(neg)}$

Recall that l_1 and l_2 are the blame labels textually representing the supplier (l_1) and the consumer (l_2) — in our example the prompt — of the contracted value respectively. When the contracted function `fixTen` is applied it verifies the contract, namely that the supplied function `neg` does not apply the function `println`. As explained before, this can not be verified when the contracted function is applied (Rice theorem). Therefore, the argument (`neg`) is wrapped into a new contracted function. The function `fixTen` is thus applied to the contracted `neg` as shown below.

$\texttt{fixTen(}\boxed{l_2, l_1| \ \texttt{nat} \xrightarrow{!call(system.println)} \texttt{nat} \mid \texttt{neg}}\texttt{)}$

Within the body of the `fixTen` function this contract-verification function is applied to the value 10. This results in validating the domain contract `nat` and as in the previous example the value 10 passes this contract. Finally the computational contract is applied.

$\boxed{l_2, l_1| \ \texttt{nat} \xrightarrow{!call(system.println)} \texttt{nat} \mid \texttt{neg}}\texttt{(10)}$

Applying the computational contract corresponds to deploying a dynamic aspect. The active pointcut is shown in superscript ($!call(system.println)$) and the responsible to blame in subscript (l_2):

$\texttt{neg(10)}^{!call(system.println)}_{l_2}$

In this example the function `neg` violates the computational contract by applying the function `system.println`. This matches the pointcut, and the blame advice is executed. The blame in this case is assigned to the prompt. This is correct because the prompt supplied the function (`neg`) which does not satisfy the contract.

5.4 Operational Semantics of Computational Contracts

In this section we precisely describe the inner workings of the computational contract system by presenting an operational semantics. In order to concentrate on the specific mechanism for checking computational contracts we do not consider objects, references or distribution at this point. Instead we describe the operational semantics of computational contracts in a simple higher-order functional language à la Scheme. An executable specification of the semantics is also available for PLT Redex [FFF09] [3]. We start our explanation by describing the CEK model of a Scheme-like language, and then explain how to extend it to support computational contracts.

5.4.1 CEK Model and Syntax Definition

The definition of our semantics follows the definition of the AspectScheme language as given by Dutchyn et al. [DTK06]. The semantics is defined as a variation on a CEK machine. The CEK machine defines program behavior by defining transition relations from one program state to the next. The state of a computation consists of a tuple:

[3] `http://soft.vub.ac.be/~cfscholl/index.php?page=at_contracts`

1. The control string (C) and its environment (E).
2. The continuation code (K).

Reduction rules of the machine are written in the form $\langle\langle \mathrm{C}\ \mathrm{E}\rangle\ \mathrm{K}\rangle \Rightarrow \langle\langle \mathrm{C}'\ \mathrm{E}'\rangle\ \mathrm{K}'\rangle$. Environments are finite maps from the set of variables x to values v. If E is an environment, then $\mathrm{E}[x := v]$ is like E except the point x where it is v.

$$\begin{array}{rcl}
v & ::= & (\lambda\,(x)\,e) \mid \texttt{true} \mid \texttt{false} \mid \texttt{empty} \mid (\texttt{cons}\ \mathrm{VC}\ \mathrm{VC}) \\
c & ::= & (\texttt{flat}\ e) \mid (\rightarrow c\ c) \mid (\rightarrow c\ c\ e\ e) \\
e & ::= & v\ x \mid (e\ e) \mid (\mathrm{prim}\ e \cdots) \mid (\texttt{if}\ e\ e\ e) \mid (\texttt{set!}\ x\ e) \mid (\texttt{mon}^{x}_{x}\ c\ e) \mid (\texttt{blame}\ x\ x) \\
\mathrm{prim} & ::= & \texttt{eq?} \mid \texttt{cons} \mid \texttt{first} \mid \texttt{rest} \mid \texttt{empty?} \mid > \mid < \\
\mathrm{MC} & ::= & \langle e\ \mathrm{E}\rangle \\
\mathrm{VC} & ::= & \langle v\ \mathrm{E}\rangle \\
\mathrm{E} & ::= & ((x\ \mathrm{MC}) \cdots) \\
\mathrm{K} & ::= & \texttt{stop} \mid \langle \texttt{ifk}\ \mathrm{MC}\ \mathrm{MC}\ \mathrm{K}\rangle \mid \langle \texttt{appprim1-k}\ \mathrm{MC}\ \ \mathrm{K}\rangle \mid \langle \texttt{appprim2-k}\ \mathrm{VC}\ \ \mathrm{K}\rangle \mid \\
 & & \langle \texttt{op-k}\ \mathrm{prim}\ \mathrm{E}\ \ e\ \mathrm{K}\rangle \mid \langle \texttt{op-k2}\ \mathrm{prim}\ \mathrm{VC}\ \mathrm{K}\rangle \\
x & ::= & \mathit{Symbol}*
\end{array}$$

Figure 5.18: Core syntax of the λ_c language

Higher-Order Contracts Figure 5.18 shows the core syntax of the λ_c language (c for contract). λ_c is a simple Scheme-like language with booleans, numbers and lists, together with the primitive operations applicable to them. The basic expressions consists of values, variables, function applications, if expressions, and assignments. There are three syntactical expressions for the definition of contracts. Flat contracts are represented by $(\texttt{flat}\ e)$ where e evaluates to a predicate function. For example, the definition of a flat contract that validates whether a value is greater than `10` can be represented as: $(\texttt{flat}\ (\lambda\ (x)\ (>\ x\ 10)))$. Composing flat contracts in order to define a functional contract is done by using the arrow operator: $(\rightarrow\ c_a\ c_r)$. In this representation c_a is the contract defined over the arguments of the function under contract, and c_r is the contract over the return value of the function under contract. Note that the definition of contracts is recursive and therefore, both c_a and c_r can be functional contracts.

Computational Contracts Computational contracts represented as: $(\rightarrow\ c\ c\ e_i\ e_b)$, extend functional contracts with: an interception component e_i and a blame assignment component e_b. Similar to the approach by Dimoulas et al. [DF11] guarding an expression with a contract is expressed by a monitor construct $(\texttt{mon}^{j}_{i}\ c\ e)$. The two labels i, j respectively indicate the supplier of the expression and the client of the expression. Finally, blame is expressed by $(\texttt{blame}\ i\ j)$.

CEK-Machine Reductions Rules The rules that govern the CEK machine are shown in Figure 5.19. There are four groups of rules: a first group governs initialization and termination, a second governs `if` statements, the third group deals with function applications and the last group deals with higher-order contracts and computational contracts. In this section we discuss how these rules evaluate simple expressions. In the next sections we extend this basic CEK machine with the reduction rules to evaluate computational contracts.

(INIT)
$$\frac{e}{\langle\langle e\ \mathrm{E}_0\rangle\ \texttt{stop}\rangle}$$

(TERM)
$$\frac{\langle\langle \mathrm{V}\ \mathrm{E}\rangle\ \texttt{stop}\rangle}{\texttt{Term}\ \mathrm{V}}$$

(IF)
$$\frac{\langle\langle(\texttt{if}\ e\ e_1\ e_2)\ \mathrm{E}\ \rangle\ \mathrm{K}\ \rangle}{\langle\langle\ e\ \mathrm{E}\ \rangle\langle\ \texttt{ifk}\ \langle\ e_1\ \mathrm{E}\rangle\ \langle\ e_2\ \mathrm{E}\ \rangle\ \mathrm{K}\rangle\rangle}$$

(IF_TRUE)
$$\frac{\langle\langle\texttt{true}\ \mathrm{E}\rangle\langle\texttt{ifk}\ \mathrm{MC}_1\ \mathrm{MC}_2\ \mathrm{K}\rangle\rangle}{\langle\mathrm{MC}_1\ \mathrm{K}\rangle}$$

(IF_FALSE)
$$\frac{\langle\langle\texttt{false}\ \mathrm{E}\ \rangle\langle\texttt{ifk}\ \mathrm{MC}_1\ \mathrm{MC}_2\ \mathrm{K}\rangle\rangle}{\langle\mathrm{MC}_2\ \mathrm{K}\rangle}$$

(APP/PRIM)
$$\frac{\langle\langle\langle\texttt{app/prim}\ e_{fun}\ e_{arg}\rangle\ \mathrm{E}\rangle\ \mathrm{K}\rangle}{\langle\langle e_{fun}\mathrm{E}\rangle\langle\texttt{appprim1-k}\langle e_{arg}\ \mathrm{E}\ \rangle\ \mathrm{K}\rangle\rangle}$$

(APP/PRIM-ARG)
$$\frac{\langle\mathrm{VC}_{\mathrm{fun}}\langle\texttt{appprim1-k}\ \mathrm{MC}_{\mathrm{arg}}\ \mathrm{K}\rangle\rangle}{\langle\mathrm{MC}_{\mathrm{arg}}\langle\texttt{appprim2-k}\ \mathrm{VC}_{\mathrm{fun}}\ \mathrm{K}\rangle\rangle}$$

(APP/PRIM-FUN)
$$\frac{\langle\mathrm{VC}_{\mathrm{arg}}\langle\texttt{appprim2-k}\ \langle(\lambda(x))\ e_{body})\ \mathrm{E}_{fun}\ \rangle\ \mathrm{K}\rangle\rangle}{\langle\langle e_{body}\ \mathrm{E}_{fun}[x := \mathrm{VC}_{\mathrm{arg}}]\rangle\ \mathrm{K}\rangle}$$

(MON)
$$\frac{\langle\langle\langle\texttt{mon}\ x_h\ x_f\ c\ e\rangle\ \mathrm{E}\ \rangle\ \mathrm{K}\rangle}{\langle\langle e\ \mathrm{E}\rangle\langle\texttt{chk}\ \mathrm{E}\ \ x_h\ x_f\ c\ \mathrm{K}\rangle\rangle}$$

(FLAT)
$$\frac{\langle\mathrm{VC}\ \langle\texttt{chk}\ \mathrm{E}\ \ x_h\ x_f\ (\texttt{flat}\ e)\ \mathrm{K}\rangle\rangle}{\langle\langle(\texttt{if}\ (e\ cv)\ cv\ (\texttt{blame}\ x_h\ x_f))\rangle\ \mathrm{E}[cv := \mathrm{VC}]\ \rangle\ \mathrm{K}\rangle}$$

(HO)
$$\frac{\langle\mathrm{VC}\ \langle\texttt{chk}\ \mathrm{E}\ \ x_h\ x_f\ (\rightarrow c_1\ c_2)\ \mathrm{K}\rangle\rangle}{\langle\langle(\lambda\ (p)\ (\texttt{mon}\ x_h\ x_f\ c_2\ (f\ (\texttt{mon}\ x_f\ x_h\ c_1\ p))))\ \mathrm{E}[f := \mathrm{VC}]\rangle\ \mathrm{K}\rangle}$$

Figure 5.19: Basic reduction rules of the CEK machine for higher-order contracts.

Initialization and Termination

- INIT: In order to evaluate a program e, the machine is started with the initial environment E_0 and the stop continuation `stop`. In the initial environment E_0 there are no bindings, and the set of variables are all mapped to error, i.e. $\mathrm{E}_0 \equiv x \mapsto$ `error` for all variables x. After initialization, the machine steps through a number of reductions until it reaches a terminal state.

- TERM: When a terminal state is reached the machine stops and the final value $TermV$ is returned as the answer of the evaluation.

If expressions

- IF: The first rule moves the predicate of the if expressions to the control string and builds a new continuation (`ifk`). This continuation saves the old continuation together with the two expression for the `true` and the `false` branch. Depending on whether the predicate evaluates to `true` or `false` the reduction rule IF_TRUE or IF_FALSE is applicable.

- IF_TRUE: In case that the IF_TRUE rule is applicable the evaluation moves the expression closure of the true branch to the control string and restores the old continuation.

- IF_FALSE: The IF_FALSE rule is analogous but moves the focus of evaluation to the false branch MC_2.

Function Applications

- APP/PRIM, APP/PRIM-ARG: These rules first evaluate the function and subsequently evaluate the argument.
- APP/PRIM-FUN: After the argument is evaluated, the body of the function is evaluated in the environment of definition extended with the binding of the formal parameter to the actual parameter.

Contracts

- MON: The monitor reduction rule moves evaluation to the body of the expression under contract and remembers the contract that is defined over the value in the continuation `chk`. This continuation `chk` can be both matched by the reduction rules for flat and higher order contracts. When the contracted value is evaluated the evaluation continues with either the rule to evaluate flat contracts or higher order contracts.
- FLAT: When the expression is evaluated to a value and there was a flat contract defined over this value the `flat` contract is transformed into an if test. This if test validates whether the predicate of the flat contract is satisfied. When the contracted value does not satisfies the contract blame is assigned. Note that contracted value is put in the environment under the variable name cv which is expected to be fresh.
- HO: Higher-order contracts are recursively translated into a wrapper function that expects one argument p. When applied the wrapper function creates two new monitors with the contracts c_1 and c_2. One for the received argument with c_1. The other monitor is created with c_2 and monitors the return value of applying the contracted function to the monitored argument. Note that the blame labels are switched for the monitor applied over the argument.

5.4.2 Higher-Order Aspect Language

In the reduction semantics for computational contracts there is a mechanism in between the evaluation of the arguments and the actual function application that allows the programmer to monitor function applications and to intercept them whenever necessary. In this section we give an overview of the language constructs adopted from aspect-oriented programming that are used in order to define computational contracts. The definition of the aspect language follows Dutchyn et al. [DTK06].

5.4.2.1 Syntax Extensions

The syntactic extensions for aspect-oriented programming required to support computational contracts are specified in figure 5.20. This syntax is defined as an extension of the syntax shown in Figure 5.18. Non-terminals appearing in the body of extended syntax replace those with the same name as in figure 5.18, unless an ellipsis (...) appears. In that case the syntax rule is extended. For example, expressions e are extended

$$
\begin{array}{rcl}
\mathsf{e} & ::= & ... \mid (\texttt{fluid-around}\ e\ e\ e) \mid (\texttt{app/prim}\ e\ e) \\
\mathrm{K} & ::= & ... \mid \langle \texttt{chk}\ \mathrm{E}\ \mathrm{A}\ x\ x\ x\ c\ \mathrm{K} \rangle \mid \langle \texttt{app1-k}\ \mathrm{MC}\ \mathrm{E}\ \mathrm{A}\ \mathrm{K} \rangle \mid \langle \texttt{app2-k}\ \mathrm{VC}\ \mathrm{E}\ \mathrm{A}\ \mathrm{K} \rangle \mid \\
 & & \langle \texttt{around1-k}\ \mathrm{MC}\ \mathrm{MC}\ \mathrm{K} \rangle \mid \langle \texttt{around2-k}\ \mathrm{VC}\ \mathrm{MC}\ \mathrm{K} \rangle \mid \langle \texttt{markapp-k}\ \mathrm{VC}\ \mathrm{K} \rangle \mid \\
\mathrm{MC} & ::= & \langle \mathrm{e}\ \mathrm{E}\ \ \mathrm{A} \rangle \\
\mathrm{VC} & ::= & \langle \mathrm{v}\ \mathrm{E}\ \ \mathrm{A} \rangle \\
\mathrm{A} & ::= & ((\mathrm{VC}\ \mathrm{VC}) \cdots)
\end{array}
$$

Figure 5.20: Syntax of the λ_c language

with the syntax to define `fluid-around` advice. Continuations are also extended in order to support the intermediate evaluation steps during aspect deployment.

When the programmer deploys an advice, the CEK machine needs to keep track of all the aspects that are currently active. Therefore value and expression closures are extended with a special aspect environment A. Such an aspect environment consist out of tuples $(\mathrm{VC}_{pcd}\ \mathrm{VC}_{adv})$, where VC_{pcd} and VC_{adv} represents the pointcut descriptor and the corresponding advice respectively.

5.4.2.2 Deployment of Aspects

In AspectScheme [DTK06], two different scoping mechanisms for aspects are considered. For the definition of computational contracts we only need the dynamically scoped semantics, which is defined with `fluid-around`. There are three reduction rules involved in the evaluation of a `fluid-around` aspect.

- FLUID-AROUND: Evaluates the pointcut descriptor to a value closure and saves the advice e_{adv} and the body of the fluid-around expression e_{body} into the continuation to be evaluated next.
- AROUND1-K: Evaluates the advice to a value closure and saves the body of the fluid-around expression to be evaluated next.
- AROUND2-K: Applies when both the pointcut and advice have been evaluated to values. It moves the evaluation to the body of the `fluid-around` expression with the currently active aspect environment `A` together with the newly deployed aspect.

5.4.2.3 Primitive Function Application

In the definition of the semantics there are two types of function applications. Primitive function applications denote the classical function applications as shown before. Regular function applications are different and inject the aspects into the computation. It is this mechanism that allows computational contracts to intercept and validate the function under contract. The three reduction rules that specify the semantics of primitive function application, shown in Figure 5.21 are almost the same as shown before. The only difference with the previous rules is the presence of the aspect environment.

5.4.2.4 Regular Function Application

At the heart of the semantics of the aspect language lies the mechanism that allows to invoke aspects during function application. During the reductions of a function

(FLUID-AROUND)

$$\xrightarrow[\langle\langle e_{pc}\ \mathrm{E}\ \mathrm{A}\rangle\ \langle\texttt{around1-k}\ \langle e_{adv}\ \mathrm{E}\ \mathrm{A}\rangle\ \langle e_{body}\ \mathrm{E}\ \mathrm{A}\rangle\ \mathrm{K}\rangle\rangle]{\langle\langle(\texttt{fluid-around}\ e_{pc}\ e_{adv}\ e_{body})\ \mathrm{E}\ \mathrm{A}\rangle\ \mathrm{K}\rangle}$$

(MARK)

$$\xrightarrow[\langle\mathrm{VC}\ \mathrm{K}\rangle]{\langle\mathrm{VC}\ \langle\texttt{markapp-k}\ \mathrm{VC_m}\ \mathrm{K}\rangle\rangle}$$

(AROUND1-K)

$$\xrightarrow[\langle\mathrm{MC_{adv}}\langle\texttt{around2-k}\ \mathrm{VC_{pc}}\ \mathrm{MC_{body}}\ \mathrm{K}\rangle\rangle]{\langle\mathrm{VC_{pc}}\ \langle\texttt{around1-k}\ \mathrm{MC_{adv}}\ \mathrm{MC_{body}}\ \mathrm{K}\rangle\rangle}$$

(AROUND2-K)

$$\xrightarrow[\langle\langle e_{body}\ \mathrm{E}\ (\mathrm{A}\cup\{\ \langle\ \mathrm{VC_{pc}}\ \mathrm{VC_{adv}}\rangle\rangle\})\ \mathrm{K}\rangle\rangle]{\langle\mathrm{VC_{adv}}\ \langle\texttt{around2-k}\ \mathrm{VC_{pc}}\ \langle e_{body}\ \mathrm{E}\ \mathrm{A}\ \mathrm{K}\rangle\rangle}$$

(APP/PRIM)

$$\xrightarrow[\langle\langle e_{fun}\ \mathrm{E}\ \mathrm{A}\rangle\langle\texttt{appprim1-k}\ \langle e_{arg}\ \mathrm{E}\ \mathrm{A}\rangle\ \mathrm{A}\ \mathrm{K}\rangle\rangle]{\langle\langle(\texttt{app/prim}\ e_{fun}\ e_{arg})\ \mathrm{E}\ \mathrm{A}\rangle\ \mathrm{K}\rangle}$$

(APP/PRIM-ARG)

$$\xrightarrow[\langle\mathrm{MC_{arg}}\langle\texttt{appprim2-k}\ \mathrm{VC_{fun}}\ \mathrm{A_{app}}\ \mathrm{K}\rangle\rangle]{\langle\mathrm{VC_{fun}}\langle\texttt{appprim1-k}\ \mathrm{MC_{arg}}\ \mathrm{A_{app}}\ \mathrm{K}\rangle\rangle}$$

(APP/PRIM-FUN)

$$\xrightarrow[\langle\langle e_{body}\ E_{fun}[x := \mathrm{VC_{arg}}]\ \mathrm{A_{app}}\rangle\ \mathrm{K}\rangle]{\langle\mathrm{VC_{arg}}\ \langle\texttt{appprim2-k}\ \langle(\lambda(x)\ e_{body})\ \mathrm{E_{fun}}\ \mathrm{A_{fun}}\rangle\ \mathrm{A_{app}}\ \mathrm{K}\rangle\rangle}$$

(APP)

$$\xrightarrow[\langle\langle e_{fun}\ \mathrm{E}\ \mathrm{A}\rangle\langle\texttt{app1-k}\ \langle e_{arg}\ \mathrm{E}\ \mathrm{A}\rangle\ \mathrm{E}\ \mathrm{A}\ \mathrm{K}\rangle\rangle]{\langle\langle\langle e_{fun}\ e_{arg}\rangle\ \mathrm{E}\ \mathrm{A}\rangle\ \mathrm{K}\rangle}$$

(APP_ARGS)

$$\xrightarrow[\langle\mathrm{MC_{arg}}\langle\texttt{app2-k}\ \mathrm{VC_{fun}}\ \mathrm{E_{app}}\ \mathrm{A_{app}}\ \mathrm{K}\rangle\rangle]{\langle\mathrm{VC_{fun}}\langle\texttt{app1-k}\ \mathrm{MC_{arg}}\ \mathrm{E_{app}}\ \mathrm{A_{app}}\ \mathrm{K}\rangle\rangle}$$

(APP_WEAVE)

$$\xrightarrow[\langle\langle(\texttt{app/prim}\ \langle\mathrm{W}_{|\mathrm{A_{app}}|}\rangle\ arg)\ \mathrm{E'}\ \mathrm{A_{app}}\rangle\ \langle\texttt{markapp-k}\ \langle(\lambda\ (x)\ e_{body})\ \mathrm{E_{fun}}\ \mathrm{A_{fun}}\rangle\ \mathrm{K}\rangle\rangle]{\langle\mathrm{VC_{arg}}\ \langle\texttt{app2-k}\ ((\lambda(x)\ e_{body})\ \mathrm{E_{fun}}\ \mathrm{A_{fun}})\ \mathrm{E_{app}}\ \mathrm{A_{app}}\ \mathrm{K}\rangle\rangle}$$

$$\begin{array}{l}
\mathrm{E'} = \mathrm{E_{app}}[fun := \langle((\lambda\ (x)\ e_{body})\ \mathrm{E_{fun}}\ \mathrm{A_{fun}}\rangle] \\
[arg := \mathrm{VC_{arg}}] \\
[jps := J(\mathrm{K})] \\
[pc_N := \mathrm{VC}_{pc_N}, adv_N := \mathrm{VC}_{adv_N} | \langle\mathrm{VC}_{pc_N}\ \mathrm{VC}_{adv_N}\rangle \in \mathrm{A_{app}}]
\end{array}$$

Figure 5.21: Reduction rules for aspects.

application there are *three main things happening at the same time*. First, a join point has to be created that represents the actual function application. Second, the join point stack has to be constructed. Finally, all deployed aspects have to be woven into the computation. All these things together are needed to execute the corresponding advice when an active aspect matches the join point stack.

There are three rules and two meta functions for the evaluation of regular function applications:

- APP,APP_ARGS: These rules are similar to the rules for primitive function applications, i.e. evaluate the function and the arguments of the function.

- APP_WEAVE: This rule moves the evaluation to the woven function application in an extended environment E′. Weaving is done by applying the function W as we explain later. Important to note here is that the function W has a subscript indicating how many aspects are in the environment i.e. $|\mathrm{A_{app}}|$. The environ-

$J(\texttt{stop}) = \langle \text{empty E}_0\ \text{A}_0 \rangle$
$J(\langle \texttt{markapp-k}\ \text{VC K} \rangle) = \langle (\texttt{stop}\ \text{VC}\ J(\text{K}))\ \text{E}_0\ \text{A}_0 \rangle$
$J(\langle ...\ \text{K} \rangle) = J(\text{K})$

$W_0 \overset{\text{def}}{=} \texttt{fun}$
$W_N \overset{\text{def}}{=} (\texttt{app/prim}\ (\lambda\ (f)\ (\texttt{if}\ (\texttt{app/prim}\ pc_N\ \ jps)\ (\ \texttt{app/prim}\ adv_N\ f\ )\ f\ ))$
$W_{N-1})$

Figure 5.22: Meta functions to weave aspects.

ment E′ is extended with with a function fun that is used in order to weave the aspects. This function when applied evaluates e_{body}. The environment is further extended with the evaluted arguments bound to arg. E′ also contains the joinpoint stack bound to the varibale jps. This joinpoint stack is created with the meta function J explained later. The environment is further extended with bindings for every pointcut pc_N and advice adv_N in the aspect environment. Every time a regular application is evaluated, a special continuation `markapp-k` is constructed. This continuation mark has no other use than marking the join point. The evaluation of a continuation mark is to simply remove it from the continuation stack, as shown in rule MARK.

Figure 5.22 shows the auxiliary functions that are needed in order to create the join point stack and weave the aspects. The function J creates the join point stack from a continuation stack K. It simply goes over the whole continuation stack and collects all marked continuations.

Once this join point stack is built, every aspect in the aspect environment must be applied. When weaving functions there are a number of variables that are assumed to be in the environment: pc_N, adv_N,fun and jps, these variables are created by the APP_WEAVE rule. The weaving function W_N creates a new primitive application for each number N bigger than zero. Each of the primitive applications validates that pc_N matches the join point stack jps. If this is the case the corresponding advice adv_N is executed, with the function f (proceed). If this is not the case the next primitive application is returned. W_0 returns the variable fun created by the rule APP_WEAVE. This implies that when no aspect is applicable the function fun is applied.

5.4.3 Computational Contracts

Now that the necessary infrastructure for building computational contracts has been presented we can define the reduction rule for computational contracts as shown in Figure 5.23.

- CC: The definition of a computational contract is similar to higher-order contracts but requires two extra parameters: the interception component and the blame assignment component. Like a higher-order contract it returns a wrapper function that — when applied — validates the arguments. The blame assignment component is first initialized with the correct blame labels. The computational contract then deploys a fluid aspect with the interception component, the initialized blame assignment component, the original function and the validated

(CC)

$$\frac{\langle \mathrm{VC}\ \langle \texttt{chk}\ \mathrm{E}\ \mathrm{A}\ x_h\ x_f\ (\rightarrow\ c_1\ c_2\ e_{pc}\ e_{adv})\ \mathrm{K}\rangle\rangle}{\langle\langle(\lambda\ (p)\ (\texttt{mon}\ x_f\ x_h\ c_2\ (mcc\ e_{pc}\ (e_{adv}\ x_h\ x_f)\ f\ (\texttt{mon}\ x_f\ x_h\ c_1\ p))))\ \mathrm{E}[f := \mathrm{VC}]\ \mathrm{A}\rangle\ \mathrm{K}\rangle}$$

$$(mcc\ e_{pc}\ e_{adv}\ f\ e_{arg}) \overset{\mathrm{def}}{=}$$
$$((\lambda\ (pc\ adv\ arg)\ (\texttt{fluid-around}\ pc\ adv\ (f\ arg)))\ e_{pc}\ e_{adv}\ e_{arg})$$

Figure 5.23: Reduction rule for computational contracts.

```
(call f) ≝
  (λ (f) (λ (jp) (eq? f (first jp))))

(prohibit_c f) ≝
  (→ (flat (λ (x) true)) (flat (λ (x) true)) (call f)
     (λ (p) (λ (n) (λ (jp) (λ (a) (blame p n))))))

(ensure_c f) ≝
  (let ((c false))
    (→ (flat (λ (x) (set! c false))) (flat (λ (x) c)) (call f)
       (λ (p) (λ (n) (λ (jp) (λ (a) (set! c true) (jp a)))))))
```

Figure 5.24: Syntactic sugar for the definition of Computational Contracts.

arguments. As before, the blame labels of the postcondition of a computational contract are switched around as explained in section 5.3.1.

The definition of a prohibit and ensure contract can now be defined in terms of the computational contract definition, as shown in Figure 5.24. A prohibit contract simply assigns positive blame in case the blame assignment component is applied. An ensure contract initializes a variable c to be $false$ in the precondition and sets this variable to $true$ in the blame assignment component. The postcondition returns this variable and blame is assigned when the variable was not set to $true$.

5.5 Discussion

Before concluding this Chapter we highlight some subtleties that arise in the interaction between computational contracts and higher-order contracts. We also highlight a subtlety in the blame assignment process that might otherwise be overlooked when using computational contracts. For both issues we present our solution and the design decisions taken to overcome these problems.

5.5.1 Identity, Sameness, and Difference

It is possible that a function is subject to verification by both a computational contract and a function contract at the same time. In that case, the question of which has precedence appears.

To exemplify this issue, consider the code example show in Figure 5.25. There are two functions defined, namely `process` and `remove`. For the discussion it suffices to know that `process` applies the function `f` given as argument. The function `remove` destructively deletes a file from the harddisk. Both functions are exported with a contract. `process` has a computational contract that prohibits the function `remove` to be applied. The function `remove` has a higher-order contract that states that the arguments of the function `remove` should be a `string` and that the return value should be a `boolean`. Let us assume that the exported function `process` is applied to the exported function `remove` from the prompt. Remember that exporting a function with a contract creates a new function that acts and behaves almost exactly like the original function, with that difference that the contract is verified. Following the notation introduced in Section 5.3, the function `process` in that application is represented as follows (l_1 is the blame label for `process`, and l_2 for the prompt):

$$\boxed{l_1, l_2|\ \texttt{(any/c} \xrightarrow{!call(remove)} \texttt{any/c)}\ |\ \texttt{process}}$$

Similarly the function `remove` is represented as follows (l_3 is the blame label for `remove`):

$$\boxed{l_3, l_2|\ \texttt{(string?}\ \to\ \texttt{boolean?)}\ \ |\ \texttt{remove}}$$

Applying the exported function `process` to the exported function `remove` results in the computational contract to be verified. After deploying the computational contract the function body of `process` is executed (line 1). In the body, the function argument `f` is applied to the number 4. Graphically, we have:

$$\boxed{l_3, l_2|\ \texttt{(string?}\ \to\ \texttt{boolean?)}\ \ |\ \texttt{remove}}\texttt{(4)}^{!call(remove)}_{l_1}$$

Note that the function `remove` is still contracted by the function contract `(string?` $\to$ `boolean?)`. At the same time a computational contract that prohibits applications of the function `remove` is also active (indicated in superscript). Evaluating either contract leads to a violation. The question is: which contract has precedence?

When precedence is given to the computational contract, blame is assigned to the function `process` and a computational contract violation is presented to the developer. For the developer it will be clear that some piece of code attempted to remove parts of his hard-disk while the contract clearly prohibits this.

When precedence is given to the function contract, blame is assigned to the module where the function `remove` was applied from. In this case the developer is presented with a precondition violation as the function `remove` is applied to a number instead of a string. A developer presented with this error message could be tempted to correct this error. Of course such an attempt would be futile as applications of the function

```
def  process( f) {  ... (f 4) ... }
def  remove(path) { ... }

def moduleInterface := object: {
 def process := provide: process withContract: any -prohibit_c(remove)-> any;
 def remove := provide: remove withContract: string -> bool;
};
```

Figure 5.25: Example where a computational contract and a function contract can be active over the *same* function.

`remove` are prohibited by the computational contract anyway.

In our implementation, by default, pointcuts like `(call f)` select applications of a function `f`, whether or not it is contracted; it relies on a `equals` function, which makes equality oblivious to contracts. This means that in the previous example, the programmer would get a computational contract violation. We also provide a `call-eq` pointcut designator, which relies on the low-level pointer equality function `eq`. In that case, the computational contract is not applied before the function contract. Hence the programmer gets a precondition violation.

Finally, note that a function can also be subject to verification by multiple computational contracts at the same time. In our implementation, we always give precedence to the computational contract that has been deployed last. The rationale behind this decision is that in case of multiple applicable violations the computational contract which is the "closest" to the violation will be presented to the programmer. We have not yet encountered any scenario where changing the precedence of the computational contracts makes sense. However, it would not be hard to support custom precedence declarations.

5.5.2 Who will Guard the Guards?

An important aspect of contract systems is whether they assume that the contracts themselves are trustworthy or not. Dependent contracts as described in Findler and Felleisen's original paper [FF02] do not enforce the domain contract defined over the arguments during the evaluation of the postcondition. Dependent contracts thus fall into the category of contracts where a contract is assumed to be always correct. This was criticised by Blume and McAllester [BM06] who extended the work on dependent contracts so that the domain contract is enforced both in the precondition *and* in the postcondition of the dependent contract. Blume and McAllester's contract system is dubbed *picky* while Findler and Felleisen's original dependent contracts are called *lax*. While *picky* contract systems capture more violations they do not assign blame to the contract. Recently, Dimoulas et al. [DFFF11] proposed *picky* blame assignment. This system dubbed *indy*, treats the contract as an *independent* party and in case that the postcondition violates the domain contract, blame is assigned to the contract.

Similar to lax contracts, during the verification of a computational contract the pre/-post contract might violate the computational contract. An example of this is shown below.

$$\boxed{l_1, l_2 |\ \texttt{(printArgument} \to \texttt{int)}\ |\ \texttt{foo}}\,\texttt{(1)}^{!call(system.println)}$$

The precondition, `printArgument` simply allows any argument to pass but also `prints` the argument. The computational contract however disallows any application of the `system.println` function. Our implementation provides lax computational contracts, as they would allow the above behavior. Adapting the notion of indy contracts to computational contracts is an important avenue of future work.

5.6 Conclusion

In this chapter we have shown that traditional contracts fall short in defining and validating concerns about the computation over which they are defined. As these concerns could not be expressed even on a single node, a first step towards ambient contracts was the exploration of how to define and validate concerns over the internal working of the computation. To this need we have presented a model called computational contracts. In this chapter we have limited the discussion of computational contracts to those defined over functions. The three main contributions presented in this chapter are:

- The definition and the concept of a novel contract system called computational contracts.
- A mechanism to specify and validate computational contracts based on scoped aspects.
- A concrete implementation of computational contracts in AmbientTalk with blame assignment.
- An executable operational semantics of computational contracts.

Computational contracts form the fundamental abstraction on top of which we will build ambient contracts. How computational contracts can be applied to prototypes and how they are sculpted towards distribution is the topic of the next chapter.

Chapter 6

Server Side Ambient Contracts

In the previous chapter we have shown how computational contracts allows us to express temporal constraints over *functions*. In this chapter we show how computational contracts can be integrated with language constructs for ambient-oriented programming. First, we give an overview of how computational contracts can be integrated with an object-oriented model. Subsequently, we give an overview of how computational contracts are integrated with the event-loop concurrency model. More concretely in this chapter we show how computational contracts can be unified with the following characteristics of ambient-oriented programming:

A Prototype-Based Object Model As argued by Dedecker et al, classes are difficult to reconcile with the distributed nature of mobile applications [DVM$^+$05]. The main issue is to keep the classes consistent amongst the distributed nodes so that objects are *self contained*. We therefore focus on how computational contracts can be reconciled with a (classless) prototype-based object model (section 6.1). We also give an overview of how the concepts of parametric polymorphism and dynamic contract inference can be used in combination with prototype-based objects (section 6.3).

The focus of this part of the chapter is to bring the concept of computational contracts to the object level. As our contract system also supports flat and higher-order contracts over objects we first give an overview of our implementation of these contracts in AmbientTalk/C. The contracts outlined at the start of this chapter are shown in figure 6.1. Green squares indicate contracts that have been adopted by related work but are explained here in the context of AmbientTalk/C. The yellow square shows the novel contract system presented and red squares indicate contracts that are not discussed here.

Event Loop Concurrency In order to mitigate the problems of volatile connections the ambient-oriented programming paradigm prescribes a non blocking communication event-loop concurrency model. A prominent characteristic in this model, is the extensive use of event handlers. Most of the time, these event handlers are triggered by an external source e.g. as reaction to distributed communication. The registration of these callbacks inverts the control flow of the application [HO06]. In such a model the control flow of the application is driven by the arrival of external events instead of being driven by the control flow of the program text. At the moment that an event handler is registered a temporal contract might be active. When at a later moment in time this event handler is triggered the contracts that were active during the registration of the

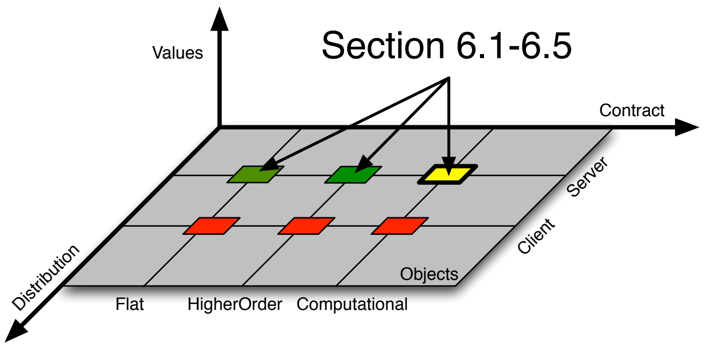


Figure 6.1: Situation in the ambient contracts design space for objects.

event handler should be restored. Current (temporal) higher-order contracts do not take such callbacks into account [FF02, SF10, DFM11, SHT+11, DF11]. In section 6.6.2 we show how computational contracts can be combined with event-handlers.

Future Contracts In order to synchronize concurrent processes the programmer can make use of future values. We will show that future values invert the responsibility of the supplier and the client of the contract. This inverted responsibility makes that the current strategies for assigning blame for higher-order values [DF11] are not suitable for the definition of contracts over future values. In section 6.6.1 we show that futures and their callbacks not only invert the control flow of the application, they also invert the blame assignment strategy leading to the notion of *blame inversion*.

The introduction of futures together with the use of callbacks is the main topic of the second part of this chapter as shown in figure 6.2. Futures as defined in AmbientTalk have not been investigated before; as such the design space of (non-blocking) futures was not explored. In this chapter we show how to define and validate flat, higher-order and computational contracts over non-blocking futures. While the focus in this chapter is on the server side of the ambient contract, futures in AmbientTalk are transparent. Therefore the future contracts shown in this section both account for the server and client side.

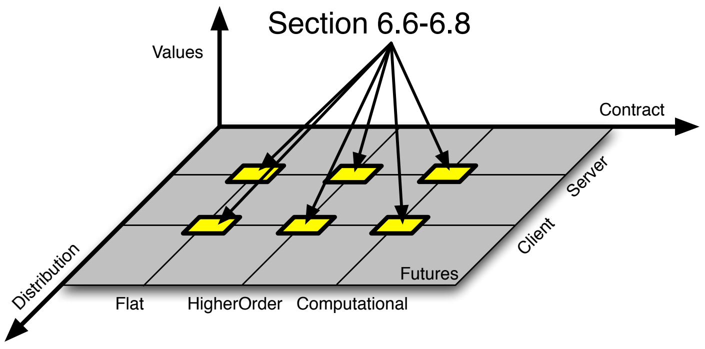


Figure 6.2: Situation in the ambient contracts design space for futures.

6.1 Object-Level Contracts

In the previous chapter, we described contracts defined over functions. In this section we show how (higher-order) contracts can be defined over objects. There are basically two ways to look at an object contract. First, an object contract can be a flat contract created with a simple predicate. For example, a predicate over the field of an object. Such a predicate can be immediately validated. Second, an object contract can be a specification for each of the methods of the contracted object. Each of these method contracts is similar to a higher-order function contract. The difference with a higher-order function contract is that a (higher-order) method contract allows the programmer to refer to the object over which the contract is defined. For example, the pop method of a stack can specify a pre-condition to ensure that the stack is not empty.

Ambient contracts are defined at the object-level. This means that a different object contract can be applied over each instance. Our ambient contract system has support for both flat, higher-order and computational object contracts as we show in the following sections.

6.1.1 Flat Object Contracts

Many contracts defined over objects can be expressed as simple predicates. For example, a contract could require that certain fields of an object are initialized. The programmer defines such a contract by constructing a predicate. This predicate takes as argument an object and checks whether all the required fields are not `null`. Flat object contracts can be validated before (as a precondition) or after (as a postcondition) executing the contracted entity and are very similar to the first-order function contracts. In fact, defining such a contract does not require any extra functionality than the one presented in the chapter 5. The only difference is that up till now the values passed to flat contracts were expected to be primitive values. However, our contract system also allows the creation of flat contracts which receive an object as argument instead of a simple value.

Figure 6.3 shows a flat object contract `sizeField` that specifies that the `size` field of an object has to be a number greater than zero. Recall that in the example snippets, exported functions are defined in a module called "defs.at" and imported by a module called "uses.at" as shown in section 5.2. This contract can then be used in order to define a contract over the argument of the `checkout` function as shown in Figure 6.3. In this example, the function `checkout` is exported with a contract that specifies that the `size` field of the `shoppingList` object should be greater than zero.

```
def sizeField := flat: {|object| object.size > 0};
def checkout( shoppingList ) { ... };
...
def checkout  := provide: checkout withContract: sizeField -> bool;
...
```

Figure 6.3: Flat Object Contract.

Importing a contracted function is transparent for the client module as shown in Figure 6.4. In this example, the `"uses.at"` module applies the function `checkout` to an object with a size field equal to zero. This application violates the contract because the argument passed to the `checkout` function should be an object with a field size bigger than zero. The "uses.at" module is blamed because it supplied a wrong argument to the contracted checkout function.

```
>checkout(object:{ def size := 0; });
```

```
1:8:uses.at violated the contract  type not respected, expected
sizeField given <obj:2037510537{size,size:=}>
at raise:(ContractException)  (249:32:ComputationalContracts.at)
at checkout(object:({ def size := 0 }))  (1:1:REPL)
```

Figure 6.4: Flat Object Contract Usage.

In general, the programmer would like to specify more detailed properties about objects. For example, a flat object contract cannot express anything about the behavior of the methods of an object. In the next section we present higher-order object contracts which allow the programmer to specify contracts over the individual methods of an object.

6.1.2 Higher-Order Object Contracts

Our approach for defining object contracts is similar to how object-level contracts were defined in Racket [SF10]. Higher-order object contracts can not be defined by using flat contracts for exactly the same reason that flat contracts are not sufficient for function contracts as explained in section 5.1. Similar to how function contracts can only be verified when the function is used in the body of the contracted function, object contracts can only be verified when the contracted object is used.

An object contract is defined by a protocol that defines a number of methods denoting the minimal interface that the object over which the contract is defined has to implement. As an example, consider Figure 6.5, which shows a contract for a stack of integers. The object contract specifies that there should be three methods implemented by the object, namely push, pop and isEmpty. The body of these methods denote the contract that is applied over the individual methods of the contracted object. The contract defined over the push method states that the argument should be an integer. Similarly, the values that are popped of the stack should be integers.

It is often important to access the fields of the object in order to specify object contracts. Therefore, flat contracts over methods receive next to the value over which they are defined a reference to the contracted object as an *optional* second argument. Such an optional argument allows flat contracts for functions to be reused in method contracts. Pre- and post-conditions that do not take any arguments are created with the keyword function contract:. The precondition of the pop operation states that the stack should not be empty. The last line in Figure 6.5 shows how the stack is exported so that it can be imported from other modules. Note that the stack contract needs to be initialized by giving it to the function Ref in order to indicate that the object is local to the actor. In the chapter 7, another initialization function FarRef is shown in order to define object contracts over far references. AmbientTalk/C also has support for isolates, with the contract constructor IsolateRef.

It is important to remark that object contracts are defined independently from the objects over which they are applied. Decoupling contract definition from application has the advantage that the same contract definition can be reused for multiple objects. In addition, it allows for better modularity since the concern of specifying the contract

```
def notEmpty := contract: { |s| !s.isEmpty() };

def StackC := ObjectContract: {
  def push(o)   { int      -> void; };
  def pop()     { notEmpty -> int;  };
  def isEmpty() { void     -> int;  };
};

...
def intStack := provide: stack withContract: Ref(StackC);
...
```

Figure 6.5: Object contract definition and deployment.

```
>import /.experimental.lang.defs;
>> null
>intStack.push("wrong");
```

```
intStack.push("wrong");
1:8:uses.at violated the contract expected Integer given "wrong"
origin:
at raise:(ContractException)  (249:32:ComputationalContracts.at)
at intStack.push("wrong")  (1:1:REPL)
```

Figure 6.6: Violation of the stack contract by the module uses.at.

is clearly separated from the functional requirements which are defined in the object itself.

Using the object contract from a different module works analogously to using a contracted function. Figure 6.6 shows an example of importing a contracted object and using it. First, the `defs` module is imported which makes the `intStack` visible in the module `uses`. Then the programmer invokes the method `push` with the string `"wrong"` on the `intStack`. As expected this results in blame being assigned by the contract system. As shown in the trace the contract system informs the programmer that an integer was expected instead of the string `"wrong"`. The error trace also shows that the blame of violating the contract needs to be assigned to the `uses` module. Note that the use of contracted values is again completely transparent for the client. It is up to the programmer of the client to read the contracts of the imported modules and abide by them.

The definition of our stack contract is sufficient for the definition of a stack of positive numbers but it is not very general. For example, in order to define a stack where the elements of the stack are integers greater than ten the programmer has to rewrite the whole contract. This is clearly not ideal and it would be better if the contract would be parameterized with another contract that defines the elements that can be pushed on and popped from the stack. In the section 6.3 we describe how we cater for these cases by making contracts first class values. Further, we give an overview of how in certain situations contracts can be inferred at runtime. In section 6.5, we show the semantics of object-level contracts. In the next section we first focus on the use of computational contracts in combination with object-level contracts.

```
1  def stack := object: {
2    def storage := [];
3    def push(object) { ... };
4    def pop() { ... };
5  };
6
7  def StackContract(pos) {
8    ObjectContract: {
9      def push(arg) { pos -prohibit_c(system.&println)-> any };
10     def pop()     { any -> pos };
11   };
12 };
13
14 def stack := object: {
15   def storage := [];
16   def push(arg) {
17     system.println("pushed " + arg);
18     storage := [arg] + storage;
19   };
20   def pop() {
21     //...
22   };
23 };
```

Figure 6.7: Computational contract over methods

```
73:21:defs.at violated the contract computational contract.
```

Figure 6.8: Methods with a computational contract blame assignment.

6.2 Computational Contracts over Objects

The object contracts shown up till now are an extension of the higher-order contracts for functions. They do not capture behavioral properties over the execution of the methods of the contracted object or the object itself. In this section we give an overview of how *computational contracts* can be defined over methods and over objects.

6.2.1 Method-level Computational Contracts

In this section, we show how computational contracts can be used in the definition of object contracts to define behavioral properties over the methods. The syntax and semantics of these contracts is identical to the syntax and semantics of computational contracts for functions, except that these contracts are now defined over methods instead of over functions.

All the high-level computational contract constructor functions defined for functions as shown in table 5.1 are also applicable over the methods of an object. This allows the programmer to prohibit or ensure that certain functions and methods are applied in a specific order when a method is executed. Figure 6.7 shows an example where the `push` method of a stack is prohibited to apply the `system.println` method. This contract is very similar to the one shown in Figure 6.5. However, now the `push` contract has a computational contract defined over it. This contract is created with the `prohibit_c` constructor function and prohibits `push` to invoke `system.println` (line 9). In order to show an example of a violation of this contract, we added an extra line in the `push` method that prints the objects that are being pushed on the stack (line 17). Figure 6.8 shows that when pushing elements on this stack, blame is correctly assigned to the `"def.at"` module.

In certain situations, the order in which the methods of an object are applied can also be important. In the next section we give an overview how the programmer can specify the order of method invocations over an object by making use of object protocol contracts.

6.2.2 Computational Contracts for Object Protocols

In this section we show the use of computational contracts in order to restrict temporal orderings of method calls on an object. Such temporal orderings over method calls are known as *object protocols* and is the subject of many research in the field of validation. In [BKA11], Beckman et al. analyze the use of object protocols in a large number of open-source projects, comprising almost two million lines of code. A remarkable result from this study is that about 90% of the protocols found, fit into the following five categories: Initialization (28.1%), Deactivation (25.8%), Type Qualifier (16.4 %), Dynamic Preparation (8.0%) and Redundant Operation (7.3%) [BKA11]. Each of these object protocols is a specification of the order of the methods that have to be invoked on an object. They are similar in nature to a computational contract but are active during the entire lifetime of the object. Moreover, they refer to a single object while computational contracts can express protocols spanning multiple objects.

We now show an extension to computational contracts in order to express object protocols. We show the applicability of our approach by showing the implementation of five computational contract constructor functions corresponding to the five categories given by Beckman et al. For each category, we first explain which protocol the category aims to validate. Then we show how we implemented the constructor function for the object contracts using computational contracts. We start our exposition of object protocols with then initialization protocol.

Object Protocol: Initialization The first object protocol category concerns the initialization of objects. In certain situations an objects must be initialized *after* construction time but before the object is meant to be used. An example of this category can be found in the `AlgorithmParameters` class of Java. Only after one of its three init methods has been invoked, calls to its `getEncoded` method are allowed. In the initialization category, calls to an instance method *m* after construction-time result in an error unless some particular initializing method *i* has been invoked at least once [BKA11].

Figure 6.9 shows the implementation of this object protocol in AmbientTalk/C. Object protocols are defined with the keyword function **`ObjectProtocol`** and are very similar to usage protocols as shown in section 5.2.3. The methods defined in an object protocol represent the states of the object protocol and their bodies define the possible transitions. In an object protocol the transitions are only triggered by method invocations on the object over which the object protocol is defined. In the initialization protocol there are three states defined: `initState`, `endState` and the obligatory `start` state. When the object protocol is applied over an object the `start` state is applied. An object protocol transitions from one state of the protocol to the next depending on which methods are applied on its object. In the `initState` there are two transitions defined. The first transition indicates that when the initMethod is invoked the state machine transitions to the `endState`. The second transition moves the state machine to the `false` state when any other method is applied by means of the predefined `anyMethod` selector. The value `false` is used by the programmer to encode invalid transitions and

```
def InitializationProtocol(initMethod) {
  ObjectProtocol: {
    def initState()     { (on: initMethod) =>  {  endState() };
                          (on: anyMethod ) =>  {  false      }; };
    def endState()      { (on: anyMethod ) =>  {  endState() }; };
    def start()         {  initState(); };
  };
};
...
provide: filter withContract:
ensure_c( InitializationProtocol("initWithPredicate") );
```

Figure 6.9: Initialization Protocol.

```
def DeactivationProtocol(deactivation) {
  ObjectProtocol: {
    def initState() {
      (on: deactivation) =>  { endState() };
      (on: anyMethod)  =>  {  initState(); };
    };
    def endState() {
      (on: anyMethod)  => {  false; };
    };
    def start() {
      initState();
    };
  };
};
```

Figure 6.10: Deactivation Protocol.

transitioning to the `false` state results in a contract violation. When the state machine is transitioned to the `endState` no violations can occur anymore because any method invocation simply transitions the state machine back to the `endState`.

Method names in the object protocol are matched using regular expressions, i.e. to select all methods that start with `set`, the following expression can be used `(on: "set.*")`. Transitions are encoded by a block (after the => operator) that invokes the next state of the protocol. Note that this transition code block can contain AmbientTalk code in order to determine the next state.

In order to use the `InitializationProtocol`, the programmer needs to provide the regular expression that selects the initialization methods. An example where the `InitializationProtocol` only allows method invocations to the contracted object after the method `initWithPredicate` is invoked is shown at the bottom of figure 6.9. A simple variation on the `InitializationProtocol` where calls to the `init` method are only allowed once, only requires adding another case in the `endState`, i.e. `(on: initMethod)=> false;`

Object Protocol: Deactivation The deactivation object protocol category validates that certain method invocation on a deactivated object instance results into an exception. A simple example of such an object is a file handler, after its `close` method is invoked method calls to its `write` method result into an exception. Like the `InitializationProtocol`, object protocols may or may not permit the deactivation method to be called more than once. The implementation of the deactivation protocol is very similar to the initialization protocol as shown in Figure 6.10. The `initState` has two transitions. The first transition moves the state machine to the `endState` when the deactivation method is invoked. The second moves the state machine back to the `initState` when any other method is invoked. In the `endState` invoking any method results into a contract violation.

Object Protocol: Type Qualifier Some object protocols disable certain methods for the lifetime of the object. An example of where such an object protocol is applicable is an unmodifiable list that inherits from a modifiable list. During the lifetime of an unmodifiable list any calls to the methods that modify the list should be disallowed. In the type qualifier category, an object instance thus enters an abstract state at construction-time which it never leaves [BKA11]. Calls to methods which are disabled after invoking the initialization method always fail. Many times the disallowed methods depend on the parameters that are passed to the initialization method. This case is shown in Figure 6.11. This object protocol is instantiated with an initializer method, a predicate and a regular expression describing the disallowed methods. When the `initMethod` is invoked in the `initState` the predicate is applied to the argument of the initialization method. This argument is accessible from within the protocol and bound to the `arg` variable. Depending on the predicate, the state machine transitions to the `allAllowed` or to the `disallowedState`. In the `allAllowed` state any method may be invoked. In the `disallowedState` any invocation of a method matching the `disallowed` regular expression result in a contract violation.

```
def TypeQualifierProtocol(initMethod,predicate, disallowed) {
  ObjectProtocol: {
    def initState() {
      (on: initMethod) =>  { |arg|
        if: !predicate(arg) then: {
          allAllowed();
        } else: {
          disallowedState();
        };
      };
      (on: anyMethod)  => {  initState(); };
    };

    def disallowedState() {
      (on: disallowed) => {  false; };
      (on: anyMethod)  => {  disallowedState(); };
    };

    def allAllowed() {
      (on: anyMethod)  => { allAllowed(); };
    };

    def start() {
      initState();
    };
  };
};
```

Figure 6.11: TypeQualifierProtocol.

Object Protocol: Dynamic Preparation In the dynamic preparation category, an instance method *m* fails unless another instance method *p* was invoked earlier [BKA11]. This protocol can again be applied over the file handler example. Method invocations to the `write` method must be preceded by an invocation to the `open` method. Objects in this category have two states, ready and not ready and the object may dynamically change from ready to not ready at numerous points in its lifetime (i.e. it is not monotonic). Implementing this category boils down to adding transition relations to the protocol shown in Figure 6.9 that move the object protocol from the not-ready state to the ready state when certain methods are applied. Figure 6.12 shows one possible implementation. In the `notReady` state invocation to the disallowed method violate the object protocol. In the `ready` state the disallowed method can transition the finite state back to the `notReady` state.

```
def DynamicPreparationProtocol(prepare, disallowed) {
  ObjectProtocol: {

    def notReady() {
      (on: disallowed ) => { |@args| false; };
      (on: prepare    ) => { |@args| ready() };
      (on: anyMethod)        => { |@args| notReady(); };
    };

    def ready() {
      (on: disallowed) => {|@args| notReady(); };
      (on: anyMethod)       => {|@args| ready();};
    };

    def start() {
      notReady();
    };
  };
};
```

Figure 6.12: Dynamic Preparation Protocol.

Object Protocol: Redundant Operation In the redundant operation category, a method invocation fails if it is invoked more than once on a given instance [BKA11]. For example, the programmer could specify that a file object can be only opened once. We found that this abstraction can be expressed as a type qualifier protocol where the `initMethod` and the `disallowed` method are the same method and the predicate always returns `true`.

Summary In this section we showed the applicability of computational contracts for checking object protocols by implementing the 5 object protocol categories as presented by Beckman et al [BKA11]. These categories account for 90 % of the object protocols currently found in the wild. Current practices to implement object protocols are low level and interwoven with the base functionality of the object, whereas we show that the implementation of these object protocols with computational contracts is concise and straightforward. Moreover, the use of computational contracts for the implementation of object protocols has the advantage that the validation code can be separated from implementation. Direct support for object protocols also has the advantage that the programmer is able to define more complex object protocols. Finally, our contract system assigns blame and gives information about where the violation against the object protocol took place and which module is the responsible for a violation.

6.3 Parametrized and Parametric Polymorphic Contracts for Objects

In this section, we show how the programmer can encode and use parametrized and parametric polymorphic contracts. We also show how a limited subset of parametric polymorphic contracts can be dynamically inferred by the contract system. The design and the implementation of the parametric polymorphic contracts in AmbientTalk/C mostly follows the work of Guha et. al [GMFK07].

6.3.1 Parameterized Contracts

Many statically typed languages provide language constructs which allow functions or class methods to work for many argument types without having to be rewritten for each type. Such types that can be specified or parameterized are called *parametrized polymorphic types*. These types were mainly introduced to improve code reuse. Many modern programming language have adopted these ideas. For example, in Java parametrized types are called generics. Parametrized types in static languages are often used for containers such as a list, a vector or a stack. In such generic containers the type of the elements in the container does not have to be defined in the implementation itself. The advantage of these generic containers is that the same implementation can be used for different types.

Similar to how generic containers are defined in a statically typed language, contracts can also be defined in a generic way. For example, consider the generic contract of a stack where the contract abstracts over the elements that can be pushed on and popped off the stack. The implementation of this generic contract is shown in Figure 6.13. The function `makeStackContract` expects a contract which verifies what is pushed on the stack and what is popped of the stack. Its return value is an instantiated object contract which can be applied on an exported object.

```
def makeStackContract(contract) {
  ObjectContract: {
    def push() { contract -> void };
    def pop()   { void -> contract };
  };
};
```

Figure 6.13: Parametrized contract constructor function for stack contracts.

As shown above the definition of a generic contract is nothing more than a constructor function which takes as argument the generic part of the contract. Once the parametrized contract constructor is defined it can be instantiated and used similarly to how templates are used in C++. The difference is that instead of parameterizing types, contracts are parameterized.

Figure 6.14 shows a snippet of a concrete use-case for the generic stack contract. The stack module interface has a function `makeStack` which expects a contract that is used as a parameter for the stack contract constructor function. This contract is applied over the newly created stack which is subsequently provided to the calling module. Once the stack is exported we can use this generic stack contract as shown in Figure 6.15. We can now define an arbitrary contract which should be satisfied by the elements that are pushed on and off the stack. In the example, we specify that the

```
def ModuleInterface: {
  def makeStack(contract)  {
    provide: stack.new() withContract:  Ref(makeStackContract(contract))
  };
};
```

Figure 6.14: Stack module making use of a parametric contract.

```
import /.at.collections.stack;
def stack := makeStack(arrayOf(maxStringSize(10)));
stack.push(["1","2",3]);
```

```
uses.at violated the contract  type not respected, expected maxStringSize 10 given number 3
origin:
at raise:(ContractException)  (249:32:ComputationalContracts.at)
at string_stack.push(["1", "2", 3])  (5:1:uses.at)
```

Figure 6.15: Parametric object contract violation.

elements of the stack should be a homogenous array where all the elements of the array satisfy the `maxStringSize(10)` contract. This contract validates that the strings in the array all have a size equal or larger than `10`. Subsequently the programmer invokes the `push` method of the newly created stack. The supplied argument is an array of which the first two elements are strings but whose last element is the number three. This is a violation of the contract and as expected the contract system assigns blame to the the `uses` module.

To conclude, contract constructor functions are a very useful abstraction in order to reuse contracts. The main reason of the simplicity of parametrized contracts is that contracts are first-class values that can be passed to and returned from functions. In the next section we elaborate on how to automate the process of instantiating contracts automatically.

6.3.2 Relationally-Parametric Polymorphic Contract Inference

In the parametrized contracts shown in the previous section the client module manually specifies which contract the elements on the stack have to satisfy. The client however, does not have any guarantee that the values that are popped of the stack are actually values that were previously pushed on the stack.

In order to facilitate for contracts that encode a certain relation between the para--metrized contracts we have adopted a technique known as *relationally-parametric polymorphic contracts* [GMFK07] and extended this technique for a prototype-based object model. The difference between a parametric and a parameterized polymorphic contract is that a parametric contract validates that the function *does not depend* on the type of the parametric variables.

In order to clearly illustrate the difference between a parametrized contract and a parametric contract consider the example shown in figure 6.16. In this example a function `id_function` is defined and exported with a contract `forAll(a)=> (a > a)`. This contract specifies that the `id_function` should work for *all* instantiations of the contract variable `a`. In the example this is clearly not the case as the `id_function` always returns the number five. As expected when using this function from another module as shown in figure 6.17 blame is assigned to the definitions module. The big difference is that a `parametrized` contract where the contract variable `a` is instantiated with the `Int` contract would *not* assign blame as parametrized contracts do not validate the "relational" part of the contract.

```
def id_function(x) {
  5;
};

def interface := object: {
  def id := provide: id_function withContract: forAll(a) => (a -> a);
};
```

Figure 6.16: Parametric polymorphic identity function.

```
>id(5)
"defs.at" violated the contract id_function is not polymorphic
(created a parametric return value) [5]
```

Figure 6.17: Applying the parametric polymorphic identity function.

In AmbientTalk/C parametric contracts can also be inferred similarly as described by Guha et al. [GMFK07]. Unlike the contracts in Guha et al. [GMFK07] the inference mechanism of AmbientTalk/C can not be detached from the `forAll` form.

Figure 6.18 shows the `map` example shown in section 5.1.2, but now specified as a relationally-parametric polymorphic contract in AmbientTalk/C. The contract should be read as: for all contracts `a` and `b` given a function from `a` to `b` and an array of `a`'s, the map function returns an array of `b`'s. In this contract, `a` and `b` are contract variables and for each invocation of the `map` function the contract system infers contracts for `a` and `b`. These contracts can be flat contracts, higher-order contracts or higher-order object contracts. The limitation of the inference mechanism is that the flat contracts that are combined in order to make higher-order contracts and method contracts are all primitive contracts such as `Int` and `String`. For example, the contract system cannot infer that a certain value has to be within a certain range.

```
def map := provide: map withContract: forAll(a,b) => ((a -> b) * arrayOf(a)) -> arrayOf(b);
```

Figure 6.18: Parametric polymorphic contract for the `map` function.

An example of using the contracted `map` function is shown in Figure 6.19. In this example the user applies the `map` function with a closure and a table containing the numbers 1 till 4. The closure that is passed to the `map` function is a function that excepts as argument a number (`x`) and returns a function. This function expects another number `y` and either gives the number back or when the number is bigger than `3` returns the string `"wrong"`. The result of applying the `map` function is thus a table of functions. In our example there is a contract violation as the return value should be an array of values all obeying the same contract `b`. The reason why it is a contract violation is that the functions in the table return either a string or an integer.

Let us show this in more detail by explaining how the contract system validates the contract. In this case the contract system deduces that the contract variable `b` is a higher-order contract (x->y) where `x` and `y` are again contract variables. When one of these functions (x->y) is applied to a number, the contract system determines that `x` should be an integer and when the function returns an integer the contract system infers that `y` has to be an integer. However, when a function of the returned array is applied to a number higher than `3` the function returns the string `"wrong"`, at that moment in time the contract system can not unify the two different contracts for the contract variable `y` and signals an error.

In the example the function in the array at index one is applied to the number two. Subsequently, the function at index two of the array is applied to the number four. The contract system infers that this is an error and presents the programmer with a blame assignment message indicating that the contract variable `b` is not obeyed by the client.

```
>def table := map( { |x| { |y| if: y > 3 then: { "wrong" } else: { y } } }, [1,2,3,4]);
>table[1](2);
table[1](4);
Check parametric varibale name -result--from-functional-parameter :: --b--
uses.at violated the contract expected Integer given string: "wrong".
```

Figure 6.19: Applying Parametric Polymorphic Contract Functions

Figure 6.20 shows a second example of applying the contracted `map` function. The difference is that instead of creating functions an array of objects is created. All these objects have a method `foo`, which has the same implementation as the function returned in the previous example. Line 6 shows how the `foo` method of the object sitting at index `1` is applied to the number `2`. Line 7 shows another invocation of the method `foo` but now on the object sitting at index 4. As expected, the contract system infers a contract violation. The reason is that the objects in the array do not behave in a uniform way. The contract system signals a contract violation against the use of contracted map function and assigns blame to the `uses.at` module. Note that the contract system presents the programmer with a stack-trace of the inferred contracts. It first hints the programmer to look at the return value of the method `foo`. The second line of the error output hints the programmer to look at the parametric variable `b`, which the contract system has inferred to be an object with the method `foo`. This helps the developer in finding out what went wrong so that he/she can fix the problem.

```
1  >def table := map( { |x|
2    object: {
3      def foo(y) {  if: y > 3 then: { "wrong" } else: {y}; };
4    }
5   },
6   [1,2,3,4]);
7  >table[1].foo(2);
8  2
9  >table[3].foo(4);
```

```
(Check parametric varibale name -result--from-method- :: foo )
(Check parametric object varibale name --b-- )
uses.at violated the contract expected Integer given string: "wrong".
```

Figure 6.20: Applying Parametric Polymorphic Contract with Objects

6.3.3 Executable Semantics

The implementation of relational parametric polymorphic contracts is presented in Figure 6.21. Every contract maintains a variable `inferred_contract` to keep track of the inferred contract (line 2). Like higher-order and flat contracts, parametric contracts are created by passing the blame labels `pos` and `neg` (line 3). The implementation also keeps track of whether the variable is instantiated by the supplier or the client explicitly. The contract system does this by keeping track that the variable is used in an even or an odd position with the variable `even` (line 3). This is an important addition because parametric variables can be used both as provided by the supplier or by the client in the same contract. In order to make this more clear consider the contract signature of the `map` where the variables are annotated with subscripts s and c to indicate whether this variable is provided by the client or the supplier: $\forall\alpha, \beta :: (\alpha_s \rightarrow \beta_c) * array(\alpha_c) \rightarrow array(\beta_s)$. As can be seen in this signature the variable α is both used as the responsible of the client and of the supplier. The first occurrence α_s is in the function f[1] given by the client to the map function. Here it indicates that the supplier passes values that satisfy the α contract to the function f. The second occurrence α_c indicates that the map function receives from the client an array of values all satisfying the α contract. What the contract system ensures is that the values passed from the client α_c are the same variables that the supplier inputs to the function f (i.e. this part of the validation checks that the function is truly polymorphic). In our implementation this is conceptually done by putting the values α_c into a coffer and unpacking these values before giving them back to the function f. If at a certain moment a value that is not packed into a coffer is given to the function f a contract violation is detected and blame is assigned.

Contract validation is split up in two conceptual cases, either the contract is in an even (supplier) or an odd position (client) as annotated to the right of Figure 6.21. We first explain the case when a contract, in an odd position, is applied over an array of values `val` (line 5). For simplicity, assume that there is exactly one argument value, and thus this value can be taken out of the array (line 7). When there was a contract inferred already we simply apply that contract over the value (line 9). Otherwise we check if the argument is a simple type, for example a number or a string (line 12-13). If this is the case the contract system has inferred that the contract should be an `Int` contract or a `String` contract. It could also be the case that the value was a closure (line 16). In case the value is a closure, it is impossible to derive how the contract of this closure looks like. Therefore, a new higher-order contract is created where the arguments and return values are again parametric contracts variables. The concrete contracts for these variable are deduced when the higher-order function is used.

Finally, `val` can also be an object (line 22). The creation of such a contract is very similar to the creation of a function contract. However, instead of creating a higher-order contract a parametric higher-order object contract is created with the function `parRef`. This function returns a proxy object that intercepts all the method invocations and works very similar to how a parametric contract works. Every time a method is invoked on the proxy it checks if there was already a contract deduced for that method. If this is not the case then a new higher-order contract is created for that method. Once again the argument and return value of that contract are parametric contracts. If a contract was already deduced it is simply applied.

Irrespectively of which contract is applied over the value the contracted value is wrapped into a `coffer` (line 27). This is just a normal object with a method `get` that returns the original value and a `name` method that returns the name of the contract. Note that wrapping values into a coffer as shown in the code is not entirely secure. However, it is our expectation that programmers using contracts do so to detect errors

[1]We arbitrarily gave it the name f to easily refer to it in the continuation of our explanation.

```
1   def makeParametricContract(name) {
2     def inferred_contract;
3     { |pos, neg, even|
4       { |val|
6           if: !even then: {
7             def val := val[1];
8             if: (inferred_contract != nil) then: {
9               val := inferred_contract.contract(pos,neg,even)(val);
10            } else: {
11
12              if: (is: val taggedAs: Number)  then: { inferred_contract := Int; };
13              if: (is: val taggedAs: Text)    then: { inferred_contract := String; };
14              //..
15
16              if: (is: val taggedAs: Closure) then: {
17                inferred_contract:=
18                makeParametric("-argument-to-functional-parameter:: "+name) ->
19                makeParametric("-result--from-functional-parameter :: "+name)) );
20                val := inferred_contract.contract(pos,neg,even)(val);
21              };
22              if: (is: val taggedAs: Object) then: {
23                inferred_contract := parRef(val);
24                val := inferred_contract.contract(pos,neg,even)(val);
25              };
26            };
27            object: { def get() { val }; def name() {name;} } taggedAs: [coffer];
28          } else: {
29            if: (is: val[1] taggedAs: coffer.and:{ coffer.name() == name }) then: {
30              inferred_contract.contract(pos, neg, even)(val[1].get());
31            } else: {
32              blame(pos);
33            }
34          }
35        };
36      };
37    };
38  };
```

Primitive

Closure

Wrap Objects

Unwrap

Odd

Even

Figure 6.21: Parametric Polymorphic Contract Inference

and are not deliberately trying to circumvent the contract system. In our code there are a few measurements taken in order to prevent the programmer from *accidentally* breaking the contract validation. First the variable `val` is looked up in the lexical scope of the wrapper object. Therefore base-level programmers can not change this value by accident [2]. Second the type tag `coffer` is local to the module where the contracts are defined. Therefore, the chance that the programmer accidentally defines objects tagged with the same type-tag are minimal.

We now focus on the case where the contract is in an even position (line 29). An even position means that the value is given by the supplier. As this computation should itself behave in a polymorphic way, it is impossible for the value to be created inside the supplier function itself. To make sure that it only returns polymorphic values it must be verified that even positioned values are wrapped `coffers`. Note that coffers can be only unwrapped by contracts with the same name. If this is not the case blame is assigned. Otherwise the inferred contract is applied over the value.

[2]In AmbientTalk, the meta programmer can still do this through the reflective layer.

6.4 Blame Assignment in OO Higher-Order Contracts

In this section we highlight the adjustments that we have made to the blame assignment mechanism shown in section 5.4 in order to incorporate contracts into a prototype-based language. Previous contract systems do not validate recursive applications. This implies that when a entity under contract applies itself recursively those recursive applications are delegated to the original value which is not under contract. The blame assignment mechanism explained so far does not verify these invocations. For the definition of module contracts the idea is that self applications within the same module can be easily validated by the programmer [SF10]. However, in the context of object protocols the temporal ordering of the applied methods is often crucial. Failing to monitor self applications can easily lead to a contract system that blames the wrong party. In the following sections we focus on explaining recursive blame assignment and why it is important.

6.4.1 Recursive Higher-Order Object Contacts

As illustrated in Figure 6.22, a contract is active between the boundaries of the value under contract and the client who uses the value. When a client module invokes a method on a contracted object exported by the supplier module, the contract intercepts this invocation. The contract then applies the contracts defined over the argument of the invocation and forwards the invocation to the contracted object.

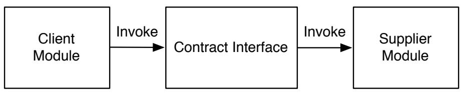


Figure 6.22: Data structure under contract.

The way the supplier module interacts with contracted entities is shown in Figure 6.23. Current contract systems only affect the way external components act with the entity under contract. This implies that self applications are not subject to any contract and thus are allowed to violate the same contracts that the client has to satisfy. As can be observed self applications resulting from an external application are not subject to contract verification.

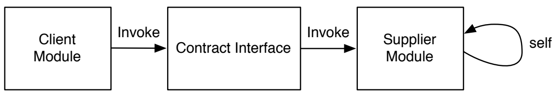


Figure 6.23: Unfair contract validation.

In the context of object protocols this is troublesome because internal self applications can leave the object protocol in an inconsistent state. For example, one of the methods of a file object could close the file. Because internal self applications are not monitored the state machine of the object protocol is *not* transitioned. When the client module at a later time invokes the write function, this application would not be seen as a contract violation. For the contract system the file was never closed. In the context of object protocols it is thus important to also monitor self applications. Because there is different judging strategy for the value under contract and the client who uses the value under contract we call these contracts *unfair*.

In order to achieve a *fair* contract validation, self references *within* the recursive structure have to be routed through the contract interface. In order to assign blame in case of a violation, the contract system needs to differentiate between recursive applications and applications coming from the client. This situation is shown in Figure 6.24. The following sections outline how we have achieved this in our contract system.

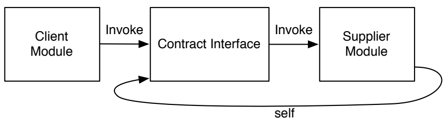


Figure 6.24: Fair contract validation.

6.4.2 Blame Assignment for Recursive Higher-Order Contracts

Assigning blame in the face of recursion is fairly straightforward. When a client module invokes a contracted method of the supplier, **self** is rebound to refer to the contract interface. In a sense, the module instantiates a contract with itself. It is important to note that the recursive invocations can sometimes pass on higher-order values passed from a client in a recursive call. In this section, we show that our recursive blame assignment strategy assigns blame correctly even when functions or objects passed from the client are used as arguments of recursive method invocations.

In order to make this more concrete consider the example shown in Figure 6.25. In this example, an object math is defined. This object has two methods, even and odd. Both methods expect two arguments: a function and an integer, they apply the given function to the given integer and check whether the returned value is even or odd. In the example implementation, even is implemented by means of a recursive invocation of the odd method. This object is then provided by the module. At the end of the example, the even method of the math object is invoked (from within another module) with a function that always returns the string "wrong" and the number four. In this example, the contract systems assigns blame to the uses.at module as expected.

We now show that the order in which the contracts are validated is crucial in order to assign blame correctly. The precedence of the contracts over the function f within the body of the odd function is shown in Figure 6.26. As can be seen in this example there are two contracts defined over the function f. A first contract is applied when the function f moves the module boundaries from the uses module to the defs module. The second contract is applied when the function f is applied to the odd function. As said before the contract system also monitors **self** sends and so another contract is applied. When the function f is applied within the body of the odd function the contract system first validates all the preconditions. *Preconditions are validated from newest contract to the oldest contract.* Therefore, first the **self** contract is validated. The supplied integer (4) passes the contract and the contract systems moves the validation to the second contract. This contract is also satisfied and the function is applied. When the function returns the result "wrong" the contract system validates the contracts. *Post conditions are validated from oldest deployed contract to newest deployed contract.* The contract thus starts with evaluating the oldest deployed contract and assigns blame to the uses.at module. In case this order would be the other way around the math module would be blamed.

This concludes our exposition of recursive blame assignment. We have shown that

```
//defs.at
def mathC := ObjectContract: {
  def even(f,i)  { (int -> int) * int -> boolean };
  def odd(f,i)   { (int -> int) * int -> boolean };
};

def math := object: {
  def even(f,i)  { not: { self.odd(f,i); }  };
  def odd(f,i)   { isOdd( f(i) ); };
};

//uses.at
math.even( { |x| "wrong" }, 4);
```

Figure 6.25: Object contract definition.

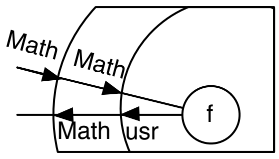


Figure 6.26: Blame order in the math example.

the order in which higher-order functions are evaluated is important for the correct validation of recursive contracts. As we have shown in this section pre-conditions have to be evaluated from newest to oldest deployed contract while this is the other way around for post-conditions. If the contract system validates contracts in that order, passing along higher-order functions in recursive invocations does not pose any problem for the correct validation of the contracts.

6.5 Operational Semantics of Object Contracts

The semantics of object contracts is not very different from the semantics as shown in chapter 5. However, in this chapter we showed that recursive calls should be captured by the contract system. In order to illustrate this point we present the semantics of recursive object-level contracts.

In chapter 4 we introduced communicating event loops and their semantics. In figure 6.27, we extend this model with the syntax extension for object contracts. Guarding an expression with a contract is expressed by a monitor construct $\texttt{mon}_i^j(\kappa,\ e)$. The two labels i, j respectively indicate the supplier of the expression and the client of the expression. Blame is expressed by $\texttt{error}^l$. There are three syntactical expression for the definition of contracts. Flat contracts are represented by $\texttt{flat}(e)$ where e is expected to evaluate to a predicate function. For example, the definition of a flat contract that verifies that a value has to be greater than `10` can be represented as follows: $\texttt{flat}(\lambda x.x > 10)$. Composing flat contracts in order to define a functional contract is done by using the arrow operator, $c_a \to c_r$. In this representation c_a is the contract defined over the arguments of the function under contract and c_r the contract over the return value of the function under contract. Note that the definition of contracts is recursive, therefore both c_a and c_r can be functional contracts too. Object protocols are expressed with $\texttt{protocol}\{\overline{\texttt{m(x)}\{\kappa_\texttt{m}\}}\}$ where κ_m represents the contract over the individual methods.

Terms	e	$::=$	$... \mid \texttt{mon}_l^l(\kappa, e) \mid \texttt{error}^l$
Contracts	κ	$::=$	$\texttt{flat}(e) \mid \kappa \mapsto \kappa \mid \texttt{protocol}\{\overline{m(x)\{\kappa_m\}}\}$
M. Contracts	κ_m	$::=$	$\kappa \mapsto \kappa$
E. Contexts	E	$::=$	$... \mid \texttt{mon}_l^l(E, e) \mid \texttt{mon}_l^l(\kappa, E) \mid \texttt{flat}(E)$

Figure 6.27: Extension for Contract Syntax.

Reductions Rules In this section we describe the reduction rules that govern the evaluation of object contracts. There are three groups of rules that govern objects, higher-order contracts and object contracts. We do not repeat the standard rules shown in chapter 4 that govern if statements, function application etc.

Higher-Order Objects Contracts Note that there is no explicit higher-order contract reduction rule for functions in AT-LITE. Functions are represented as objects with an apply method, the use of a higher-order contract over functions is thus expressed in terms of object contracts.

- FLAT: The reduction rule for flat contracts replaces the expression in the hole with an if expression. It either returns the monitored value untouched or assigns blame.

- OBJECT CONTRACT: An object contract creates a new proxy object that forwards all method invocations to the contracted object. Note that the contract defined over the object κ_p is saved in the monitor. As shown in the next reduction rule this is necessary to reapply the contract.

- METHOD CONTRACT: A method contract is similar to a higher-order contract. It places a new monitor over the argument value x in order to validate the contract κ_1. Note that again the blame labels l and k are switched. The return value of executing the method is guarded with a monitor for the contract κ_2. Further a method contract rebinds the variable `this` to a newly created contracted version. This rebinding is necessary to ensures that internal contract violations are also validated in the face of recursion (as shown in section 6.4.1).

This concludes the overview of the semantics for object contracts. We have not shown how to incorporate aspects in our object-oriented language as the introduction of future contracts requires a few extensions to the aspect language that we have shown in chapter 5. In the following sections we first show the concepts of future contracts and subsequently show their operational semantics in section 6.8.

6.6 Future Contracts for Event Loop Concurrency

In the previous sections, we showed how to combine the concepts of computational contracts with a prototype-based object model. Now we turn our attention to the abstractions of the event loop concurrency model and show how computational contracts can be integrated with those abstractions as introduced in section 2.3.2.

(FLAT)

$$\mathcal{A}\langle \iota_a, O, Q, e_\Box[\texttt{mon}_l^k(\texttt{flat}(e), v)]\rangle\rangle \to_a \mathcal{A}\langle \iota_a, O, Q, e_\Box[\texttt{if}\ (e\ v)\ v\ \texttt{error}^k]\rangle\rangle$$

(OBJECT CONTRACT)

$$\frac{\kappa_p = \texttt{protocol}\{\overline{m(x)\{\kappa_1 \mapsto \kappa_2\}}\}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\Box[\texttt{mon}_l^k(\kappa_p, \iota_o)]\rangle\rangle \to_a \\ \mathcal{A}\langle \iota_a, O, Q, e_\Box[\texttt{object}\{\overline{m(x)\{\texttt{mon}_l^k(\kappa_p, \kappa_1 \mapsto \kappa_2, \iota_o.m(x))\}}\}]\rangle\end{array}}$$

(METHOD CONTRACT)

$$\frac{o = \mathcal{O}\langle \iota_o, \texttt{o}, \overline{f := v}, M\rangle \in O \qquad m(\overline{x})\{e\} \in M}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, e_\Box[\texttt{mon}_l^k(\kappa_p, \kappa_1 \mapsto \kappa_2, \iota_o.m(\overline{v}))]\rangle \to_a \\ \mathcal{A}\langle \iota_a, O, Q, e_\Box[\texttt{mon}_l^k(\kappa_2, [\texttt{mon}_k^k(\kappa_p, \iota_o)/\texttt{this}][\texttt{mon}_k^l(\kappa_1, \overline{v})/\overline{x}]e)]\rangle\end{array}}$$

Figure 6.28: Recursive Contract Semantics

6.6.1 Contracts for Futures

Recall from section 2.3.2 that the result of sending an asynchronous message in an event loop concurrency language is a future [YBS86]. A future acts as the placeholder for the return value of the asynchronous message. This mechanism greatly reduces the programing effort of the programmer because he no longer needs to write code to explicitly retrieve the result of an asynchronous message. As futures are an essential mechanism used in any ambient application, the ambient contract system has direct support for future values. A future contract is denoted as $Fut(C_r)$. In this contract Fut is a contract constructor and C_r denotes the contract that the resolved value of the future must satisfy. Future contracts can be used both for the domain and for the range of a function. For example, a contract defined over a function that expects a future that resolves to a string and returns a future that resolves to a natural number can be expressed as: $Fut[string] \to Fut[nat]$.

Future Contracts: Example An example of a contract making use of future contracts is shown in Figure 6.29. The supplier first defines a function sum. This function expects two future values and returns the sum of the two resolved values. The exact workings of this code is explained after the client code is shown. The contract defined over the function sum (line 9) states that the function must be applied to two future values and returns a future value. All these futures are futures that must be resolved with an integer value.

```
1  def sum(f1, f2) {
2    when: f1 becomes: { |a|
3      when: f2 becomes: { |b|
4        a+b;
5      };
6    };
7  };
8
9  def sum := provide: sum withContract: Fut[int] * Fut[int] -> Fut[int];
```

Figure 6.29: Server providing a function with a future contract.

Figure 6.30 shows a client that imports the `sum` function from the supplier. The client first creates a future-resolve pair `f1` and `r1` with the function `makeFuture` (line 1). The result of the function `makeFuture` is a table where the first element is a fresh future and the second element the corresponding resolver, as outlined in section 4.1.2. The result of this table is deconstructed using a parallel assignment. The client then proceeds and makes a second future-resolver pair (line 2). Then the `sum` function is applied and the returned future is stored in the variable `result`. When the `result` future resolves to the sum of the resolved futures `f1` and `f2` the client prints this value (line 5). At line 7 and 8 the client explicitly resolves the futures with the values `5` and `10`. Notice that futures are typically created transparently (see section 2.3.2). However in AmbientTalk futures are first class objects. Here we use them explicitly for the sake of the example.

```
1 def [f1,r1] := makeFuture();
2 def [f2,r2] := makeFuture();
3 def result :=  sum(f1,f2);
4 when: result becomes: { |sum|
5   system.println("The sum is " + sum);
6 };
7 r1.resolve(10);
8 r2.resolve(5);
```

Figure 6.30: Client using a future contract.

Reconsider the `sum` function shown in Figure 6.29. Note that this code makes extensive use of future pipelining. First, the outer **`when:becomes`** block returns a first future, which we will call f_o (line 2). This future is returned as the result of applying the `sum` function and bound to the variable `result`, as shown in Figure 6.30. Second, when the client resolves the future `f1` to a value the inner **`when:becomes`** block (line 3) is registered which returns a new future lets say f_i. At that moment in time the future f_o is resolved to the future f_i. Finally, when at a later moment in time the client also resolves the future `f2`, the inner block is triggered and the future f_i is resolved. This triggers a cascading effect where the future f_i is resolved to the value `15` which triggers resolving f_o with the value `15` which in his turn triggers the **`when:becomes:`** block of the client shown in figure 6.30 (line 4). As future pipelining is used extensively when writing ambient applications, it is important to take into account future pipelining in the definition of an ambient contract system.

In the example all the contracts are satisfied and thus the client simply prints the “The sum is 15”. When the client would resolve the future `f2` with the value `"wrong"` the contract system detects the violation and assigns blame as shown in Figure 6.31. Note that the contract system needs to be aware of the fact that future values are the responsibility of the client. How blame assignment and future contract validation is performed is detailed in section 6.7.1.

Note that the contracts that are used in a future contract can be any contract that we have previously defined. Therefore the values which futures can be resolved to are not limited to simple integers. They can be functions, objects and even other futures.

```
>1:8:uses.at violated the contract, expected integer given wrong
```

Figure 6.31: Future blame.

6.6.2 Callbacks and Computational Contracts

In event loop concurrency languages such as AmbientTalk and E, the use of callbacks is omnipresent. Every time a **`when`**`:`**`becomes`**`:` block is registered there is an entry point created in the code that can be triggered in a later turn of the actor (see 2.3.2). It is important to take such callbacks into account when specifying computational contracts. In order to see why, consider the example shown in Figure 6.32. The function `makeMove` has a computational contract defined over it that disallows applications of `system.println`. Inside of the definition of the `makeMove` function a callback is registered that awaits the answer of an asynchronous message. At a later time when this future resolves to a value the callback is applied. However, this future is always resolved in a different turn than the application of the `makeMove` function. Therefore the computational contract that was active during the application of the `makeMove` function is no longer active. This means that the `makeMove` function can violate the computational contract simply by evaluating it in a callback function.

The use of a **`when`**`:`**`becomes`**`:` block in combination with sending an asynchronous message is the equivalent of a remote method invocation. Therefore the code in the **`when`**`:`**`becomes`**`:` block is a logical extension of the ongoing program. Programmers thus expect that contracts defined over such blocks are also active when the **`when`**`:` **`becomes`**`:`block is triggered, i.e. right after the remote method invocation. *In most cases the programmer expects computational contracts to be active within the body of the registered callbacks. Therefore the contract system needs to be able to capture violations beyond single turns of the actor.* How computational contracts are validated and how blame is assigned in the presence of callbacks and futures is the subject of the next section.

```
def makeMove(player) {
  when: player<-getName() becomes: {
    ...
  };
};

...
def makeMove := provide: makeMove withContract: any -prohibit_c(system.println)-> any;
...
```

Figure 6.32: Example of callbacks in combination with a computational contract.

6.7 Contract Validation and Blame Assignment in Event Loop Contracts

In this section we give a detailed overview of how contracts are validated and how blame is assigned in the context of event loop contracts.

6.7.1 Future Type Message Contract: Blame Inversion

The use of futures inverts the responsibility of the supplier and the client compared to higher-order functions. To make this clear consider Figure 6.33. To the left of this figure, the client module passes a future over the module boundaries to the server module. For example, by applying a function exported by the server module. Assume that the server module has a contract defined over this function and thus wraps the future value into a contract. The server then registers callbacks that are triggered when the future resolves (not shown in the figure). Later in time the client module resolves the future with a value as shown to the right of Figure 6.33. At that moment in time the resolved value also crosses the module boundaries and is applied to the callbacks in the server module.

It is crucial to realize that the value under contract (the future) is resolved from outside the module that has a contract defined over it. Normally, contracted values are applied from inside the module which have a contract over them. For future values this is the other way around. In the case of futures, it is always the external module that resolves the future. It is also the external module that should be blamed when the resolved future does not satisfy the contract C_r.

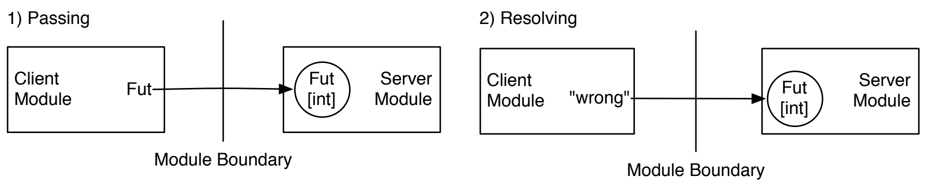


Figure 6.33: Passing (left) and resolving (right) a future from the client module to the server module.

As shown in the previous examples, it is the responsibility of the client that passes the future value to resolve the future as specified by the contract. Because of the reversed responsibility, the current blame assignment strategy for higher-order values is no longer applicable for the definition of future contracts. *Future values and their callbacks not only invert the control flow of the application, they also invert the blame assignment strategy*. While current contracts defined over higher-order functions switch around the blame labels contracts for future values do not require the blame to be switched.

6.7.2 Executable Semantics of Future Contracts

In order to show the internal workings of the blame assignment strategy for future values we extend the didactical contract system shown in section 5.3. As future contracts do not effect the definition of the guarding functions, higher-order contracts and computational contracts we do not repeat them here.

The definition of the future contract is built on the concept of future pipelining. When a future contract $Fut(C)$ is defined over a future f, the contract installs a callback on this future, and returns another future f_c. When the future f resolves to a value v the future contract applies the contract C over the value v. It then resolves the future f_c with the contracted value v or assigns blame.

The contract constructor function for future constructs is shown in Figure 6.34. It takes as an argument a contract that restricts the values to which the future resolves. It returns a closure that expects the blame assignment labels pos and neg. When this function is applied it instantiates the contract with the correct blame labels. *Note that although a future contract is a higher-order contract the blame labels are not switched, i.e. blame is inverted.* After creating the contract it returns a new closure that is applied by the contract system with the future value over which the contract is defined. When

```
 1  def fut(contract) {
 2    { |pos,neg|
 3      def contractp := contract(pos, neg);
 4      { |future|
 5        when: future becomes: { |val|
 6          contractp(val);
 7        };
 8      };
 9    };
10  };
```

Figure 6.34: Future contract constructor.

this closure is applied it registers a callback on the future. The return value is the result of the **when**:**becomes**: block (line 5). Recall that this is a future value on its own that resolves when the **when**:**becomes**: block is triggered. When this **when**:**becomes**: block is triggered the `contract` is applied over `val`, i.e. the value to which the future resolves to. The contracted value is then propagated in the future pipeline.

6.7.3 First-Class Aspect Environments for Validating Callbacks

In the definition of computational contracts the validation of temporal properties relies on aspect-oriented techniques as shown in section 5.3.2. When using callbacks the control flow in which the callback is registered and the control flow in which the callback is triggered are different. The reason for this is that futures are always resolved in a different turn of the actor. Computational contracts as defined before, are only active in the dynamic extent of the applied contracted entity (a function or a method). In many applications this limited reach of computational contracts is counter intuitive. Therefore we have extended our aspect language with language constructs that allow the programmer to arbitrary extend the reach of aspects, even beyond a single turn of the actor. The way in which this is made possible is by allowing the programmer to capture the aspect environment and reify it as a first class value. It is also possible to re-apply these aspects at a later moment in time. In order to guarantee the programmer that his contracts can not be removed, re-applying an aspect environment does not remove the currently active aspects. Applied aspects can not be removed by the programmer they can only be extended, or in terms of contracts we allow the programmer to make the temporal contracts more restrictive but we do not allow the programmer to loosen the contracts.

The aspect language provides two functions to reify the aspect environment, as shown in Table 6.1. `withCurrentAspects` expects a block and applies this block to an array containing all the aspects in the current aspect environment. `restoreAspects` expects an array of aspects and a code block. It deploys the aspects in the dynamic scope of applying the block. Note that the aspect environment is a set and that reapplying the same aspect multiple times will not result in weaving of the same aspect multiple times. This is similar (both in spirit and implementation) to statically-scoped aspects in AspectScheme [DTK06].

The formulation of a specialized **when**:**becomes**: variant that captures the aspect environment and restores this environment when its closure is triggered is shown in Figure 6.35. When loading the contracts in AmbientTalk the normal **when**:**becomes**: function is overwritten with this definition. As can be seen in the figure the aspect

Operation	Operation Signature
withCurrentAspects	$arrayOf(Aspects) \rightarrow \alpha \rightarrow \alpha$
restoreAspects	$arrayOf(Aspects) \rightarrow \emptyset \rightarrow \beta \rightarrow \beta$

Table 6.1: Aspect Extensions for First Class Aspect Environments.

```
def with_aspects_when: future becomes: block {
    withCurrentAspects: { | currentAspects |
      when: future becomes: { |value|
        restoreAspects: currentAspects in:  {
          block(value);
        };
      };
  };
};
```

Figure 6.35: Definition of a variant on the **when:becomes:** block.

environment is captured when the callback function is registered and before executing the callback `block`, the aspects are restored. As computational contracts are encoded as aspects and aspects are captured, the callback block of this **when:becomes:** variant is also validated.

In order to support such contracts, the underlying aspect language, shown in chapter 5, needs to be extended so that aspects outlive multiple turns of the actor. In Figure 6.36 an example of the this construct is shown. In this example the programmer first defines an aspect that intercepts all applications of `system.println`. Then a new future value is made and the aspect is deployed over the registration of a **with_current_aspects :becomes:** block. The operation of this construct is exactly the same as the **when: becomes:** block but it captures all the aspects. As shown in the output at the end of Figure 6.36 an error is thrown even though the **when:becomes:** block is triggered in a different turn of the actor.

In brief, AmbientTalk allows the registration of callbacks on a future which will be triggered when the future is resolved to a value. The combination of callbacks and computational contracts requires a scoping mechanism that allows the computational contracts that were active during the creation of the callback to be restored when the callback is invoked. We have shown that by adapting the underlying aspect language with an aspect capturing **when:becomes:** block the semantics of computational contracts is as would be expected from the programmer.

6.8 Operational Semantics of Higher-Order Event Loop Contracts

The definition of future contracts does not require us to adjust the operational semantics as it can be implemented with the current constructs in the language. Callbacks however, rely on a mechanism to reify the aspect environment which is not possible in the semantics shown so far. In this section we show how to support first class aspect environments.

In figure 6.37 the syntactic extensions for the definition of aspects are shown. `fluidBefore`$(e, e)\{e\}$ is used to deploy an aspect, `withCurrentAspects`$\{e\}$ reifies

```
def c := aspect: call(&system.println)  advice: { |proceed, arguments|
  error: "Printing text is not allowed in this code block!";
};

def [future, res] := makeFuture();

fluid: c deploy: {
  with_current_aspects: future becomes: { |value|
    system.println(value);
  };
};

res.resolve(2);
```

```
[futures.at] Warning: future has no listeners to catch exception:
"Printing text is not allowed in this code block!"
origin:
at error:("Printing text is not allowed in this code block!")  (43:16:AspectsTest.at)
at system.println(value)  (9:17:AspectsTest.at)
```

Figure 6.36: Aspect capturing registration construct.

the aspect environment and $\texttt{deployAspects}(e)\{e\}$ redeploys the aspects.

Terms	e	$::=$	... | $\texttt{fluidBefore}(e, e)\{e\}$
			| $\texttt{withCurrentAspects}\{e\}$
			| $\texttt{deployAspects}(e)\{e\}$
E. Contexts	E	$::=$	... | $\texttt{fluidBefore}(E, e)\{e\}$
			| $\texttt{fluidBefore}(v, E)\{e\}$
			| $\texttt{deployAspects}(E)\{e\}$

Figure 6.37: Extension for Event Loop Contracts Syntax.

Object-Oriented Aspect Language Extending the object-oriented language with the necessary aspect-oriented language extensions is similar to how the functional language show in chapter 5 is extended. The reduction rules that make up the basic object-oriented aspect language are shown in Figure 6.38.

- FLUID-BEFORE: Extends the aspect environment with the pointcut descriptor and the advice. It then continues to evaluate the corresponding closure e_{body}.

- INVOKE-PRIM: In order to make the distinction between a regular method invocation and a method invocation coming from the underlying system there are two rules for method invocations. This rule is used for method applications by the contract system.

- INVOKE-WEAVE: This rule weaves the aspects into the method application. It only weaves in those aspects that are in the aspect environment.

- W: The weaving function `W` weaves in the current aspects by applying each of the pointcut descriptors and when the result is positive the corresponding advice is applied. After matching the aspect environment to the current point cut the original function method is applied.

In Figure 6.39 the rules for reifying the aspect environment are shown.

(FLUID-BEFORE)

$$\frac{r_{pc} \in O \qquad r_{adv} \in O}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, A, e_\Box[\texttt{fluidBefore}(r_{pc}, r_{adv})\{e\}]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, A \cup \{(r_{pc}, r_{adv})\}, e_\Box[e]\rangle\end{array}}$$

(INVOKE-PRIM)

$$\frac{\begin{array}{c}\mathcal{O}\langle \iota_o, t, F, M\rangle \in O \\ r = \iota_a.\iota_o \qquad m(\overline{x})\{e\} \in M\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, A, e_\Box[prim(r.m(\overline{v}))]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, A, e_\Box[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\end{array}}$$

(INVOKE-WEAVE)

$$\frac{\begin{array}{c}\mathcal{O}\langle \iota_o, t, F, M\rangle \in O \\ r = \iota_a.\iota_o \qquad m(\overline{x})\{e\} \in M\end{array}}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, A, e_\Box[r.m(\overline{v})]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, A, e_\Box[W(A, r, m, \overline{v})]\rangle\end{array}}$$

Figure 6.38: Object-Oriented Aspect Language

(WITH-ASPECT-ENVIRONMENT)

$$\frac{r_{block} \in O}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, A, e_\Box[\texttt{withCurrentAspects}\{r_{block}\}]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, A, e_\Box[r_{block}.apply(A)]\rangle\end{array}}$$

(DEPLOY-ASPECTS)

$$\frac{r_{block} \in O}{\begin{array}{c}\mathcal{A}\langle \iota_a, O, Q, A, e_\Box[\texttt{deployAspects}(A')\{r_{block}\}]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, O, Q, A \cup A', e_\Box[r_{block}.apply()]\rangle\end{array}}$$

Figure 6.39: Extensions for reifying the aspect environment.

- WITH-ASPECT-ENVIRONMENT: This reduction rule applies the block with the current aspect environment.

- DEPLOY-ASPECTS: This reduction rule moves the evaluation to the given block in an extended aspect environment. Note that the aspect environment is extended and not overwritten. This is important when evaluating contracts because overwriting the aspect environment would give the programmer the power to remove or loosen contracts. In our semantics we do not allow such an operation.

6.9 Conclusion

In this chapter we have shown extensions to the computational contract model shown in chapter 5. First we have studied the concepts of computational contracts for object-oriented and synchronous message passing. Important to note is that violations against the defined contract can also be caused by self applications. Traditional contract systems based on wrappers do not take into account self applications which can lead to misleading blame assignments. Later we have shown how computational contracts can deal with asynchronous concepts introduced by futures and their callbacks. This has lead to the notion of blame inversion. The main contributions presented in this chapter are:

- An expressive mechanism to specify and verify computational contracts based on scoped aspects in combination with prototype based objects.
- Relational parametric polymorphic contracts for prototype based objects
- The concept of blame inversion as a consequence of the inversion of control associated with the use of futures and callbacks.
- The introduction of first class aspects environments and their application for the specification of computational contracts that can be active over multiple turns of an actor.

In the next chapter we give an overview of how contracts can be applied in a distributed manner and more specifically over far references.

Chapter 7

Client Side Ambient Contracts

The contracts defined in previous chapters of this dissertation express properties over the objects owned by a single actor. In chapter 5, we showed how higher-order contracts can be extended in order to also specify behavioral properties over the contracted value leading to the notion of computational contracts. Subsequently we detailed how computational contracts can be extended in order to validate properties in an object-oriented event loop concurrency model. In this chapter we show how the concepts of flat, higher-order and computational contracts can be extended in order to validate contracts that express properties over objects owned by *another* actor. With these contracts, clients can protect themselves from getting misbehaving far references from the server. Before giving a detailed overview of these concepts we give a sneak preview of each of these extensions:

Flat Far Reference Contracts We start our overview of the distributed aspects of ambient contracts with flat far reference contracts, i.e. predicates over far references. *A flat far reference contract validates a predicate over an object hosted by another actor.* Validating flat reference contracts thus requires asynchronous communication with the remote object over which it is defined. This is in contrast with flat contracts in a local synchronous model where predicates can be validated immediately. Because communication is involved in the validation of flat far reference contracts, it is significantly more complex to guarantee inference-free contract validation. In section 7.2, we show that contract validation algorithms that make use of blocking can introduce deadlocks and when they do not introduce deadlocks they can alter the order in which messages are processed. We propose a novel *asynchronous* contract system where the validation of flat contracts over far references are *postponed* until a message is sent to the contracted remote object. By postponing the verification of the flat far reference contract, deadlocks can be avoided and the message ordering semantics can be preserved.

Higher-Order Far Reference Contracts Sending a message to a far reference can result in transitively receiving more far references therefore there is a need to support higher-order far reference contracts. A higher-order far reference contract is similar to a higher-order object contract but specifies properties over a *remote* object, i.e. it validates the messages that are sent across actor boundaries. When an asynchronous message is sent to a remote object the result is a future. Higher-order far reference contracts also validate the values to which these futures resolve to. Finally, we also show how computational contracts can be defined over far references.

Computational Message Contracts We give an overview of how computational contracts can be used in order to validate the messages that are being sent during the execution of a contracted entity. This is significantly different than specifying a contract over a far reference. When a contract is defined over a far reference it only validates the messages that are sent to that far reference. With computational contracts we verify the outgoing messages as the effect of evaluating a message. These messages can be sent to arbitrary nodes in the network. With computational contracts we allow the programmer to validate these outgoing messages and prevent that these messages are being sent in the first place.

With flat, higher-order and computational contracts over far references we fill the last parts of the ambient contracts design space as shown in figure 7.1. In this figure, green squares indicate contract system that have been discussed in previous chapters. The yellow squares show the novel contract system presented in this chapter.

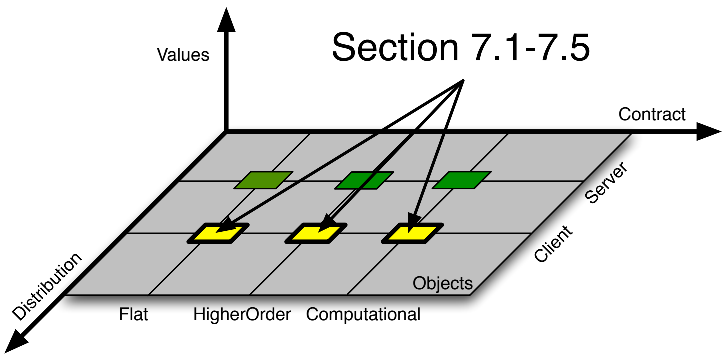


Figure 7.1: Situation in the ambient contracts design space.

7.1 Flat Far Reference Contracts

In this section, we give an overview of the extensions we have made to the computational contract system in order to define flat contracts over far references. We show a number of variations of flat far reference contracts with respect to the contract lifetime. We show how our contract system allows the programmer to express these different variations of flat far reference contracts depending on the requirements of the application at hand.

7.1.1 Flat Far Reference Contracts in Action

We explain the use of flat far reference contracts by means of a small example application. Consider an ambient game where the players register themselves by sending an asynchronous message to a coordinator. Assume that in this game, one of the requirements is that players are not allowed to participate in two different games at the same time (the player could be involved in a game hosted by another coordinator). We now show that validating this requirement can be done by making use of a flat far reference contract.

The relevant snippet of the coordinator of the ambient game is shown in figure 7.2. As shown in the figure, the coordinator is implemented as an object which has a method `addPlayer` (line 2). The coordinator actor sends an asynchronous message `getName` to the peer and prints information to the screen that a new player has been added to the game (line 4).

```
1   Coordinator := object: {
2     def addPlayer(p) {
3     ...
4       when: p<-getName() becomes: {|name|
5         system.println("Added Player: " + name);
6       };
7     ...
8     };
9   };
10
11  def CoordinatorContract := objectContract: {
12    ...
13    def addPlayer(p) {
14      Assert: {|p| p<-notInGame() } -> any;
15    };
16    ...
17  };
18
19  deftype CoordinatorType;
20
21  export: Coordinator as: CoordinatorType withContract: Ref(CoordinatorContract);
```

Figure 7.2: Defining and exporting the coordinator object on the server actor.

The `Coordinator` object is contracted with an object contract `CoordinatorContract` shown at line 11. This object contract specifies that the contracted object should have at least an `addPlayer` method. It also specifies a flat contract over the `addPlayer` method. Flat far reference contracts are expressed with the keyworded message `Assert:` which — just like a flat contract — expects a predicate. However, as flat reference contracts are defined over remote objects the result of evaluating the predicate is *not a boolean but a future that resolves to a boolean*. The flat far reference contract shown in the example, expresses that the `addPlayer` method expects a player `p` which is not involved in another game. Validating this predicate is done by sending an asynchronous message `notInGame` to the newly added player. Further, the contract over the `addPlayer` method specifies that the return value can be anything, indicated by the `any` contract.

The server then defines a type tag, `CoordinatorType`, for exporting the coordinator (line 19). Finally, the coordinator object is contracted by the `CoordinatorContract` and exported with the type tag `CoordinatorType` (line 21).

```
Player := object: {
  def notInGame() { ... };
  def getName() { "Player1" };
  ...
};

when: CoordinatorType discovered: { |coordinator|
  coordinator<-addPlayer(Player);
};
```

Figure 7.3: Discovering and registering with the coordinator by the client actor.

Figure 7.3 shows the definition of a player object defined in the client actor. The player object has two methods `notInGame` and `getName`. When the coordinator object is discovered in the proximity the **`when:discovered:`** block is applied to a far reference pointing to the coordinator object hosted by the server actor. Inside the **`when :discovered:`** block an asynchronous message is sent that adds the player object to the newly discovered coordinator.

When the object on the server actor processes the `addPlayer` message it sends an asynchronous message `getName` to the player. As there is a contract defined over this far reference, the contract system validates that the player is not yet in game. Assume that in our example the `notInGame` method returns that the player is already in a game. In this case a violation of the contract is detected by the contract system. The blame message as presented to the programmer is shown in Figure 7.4. Blame is assigned to the module that hosts the player object.

```
<78:5:uses.at> violated the contract,expected
the far reference to pass: {|p| p<-notInGame()} given
<far ref:<obj:984103443{notInGame,getName}>>
```

Figure 7.4: Asynchronous contract violation.

In this section we have shown an example of a flat far reference contract. Contrary to traditional flat contracts a flat reference over a far reference requires the asynchronous communication. In section 7.2, we explain why validating such contracts is difficult. In the next sections we first show how the programmer can control the lifetime of flat far reference contracts.

7.1.2 The lifetime of a Far Reference Contract

Flat contracts in a synchronous world are validated only *once*. Normally this validation is performed before the contracted message is processed. As we will show in section 7.2 this is not always feasible in an asynchronous world. Therefore, in order to validate flat far reference contracts the asynchronous contract system exploits the asynchronous nature of the event-loop system and validates the flat contract at the moment the contracted far reference is *used*.

Delaying contract validation in an asynchronous message passing model poses a number of questions which do not manifest themselves in a synchronous world. Validating the flat far reference contract once might be enough in certain situations, in other situations it might be worthwhile to validate the asynchronous contract *every time* the far reference is used, i.e. when an asynchronous message is sent to the contracted far reference. Therefore we have opened up the interface of the flat far reference contracts to allow for variations in the lifetime of the contract. In Figure 7.5 two contract constructors, `ForeverAssert` and `OnceAssert` are specified. `ForeverAssert` validates the flat contract whenever an asynchronous message is sent to the contracted far reference. `OnceAssert` validates the flat contract only once, i.e. the first time an asynchronous message is sent to the contracted far reference.

In order to define a flat far reference with a custom validation lifetime the programmer can create a new variation by using the keyword message `Assert:withLifeTime:`. This function expects the predicate over the far reference and an object which has at least one method called `validate`. A new lifetime object is initialized for every flat contract that is applied to a far reference. From that moment on, the lifetime object's

```
def ForeverAssert: block {
    def lifeTimeObject : = object: {
      def validate(message) { true; };
    };
    Assert: block withLifeTime: lifeTimeObject;
};

def OnceAssert: block {
    def lifeTimeObject : = object: {
      def count := 0;
      def validate(message) {
        count := count + 1;
        count == 1;
      };
    };
    Assert: block withLifeTime: lifeTimeObject;
};
```

Figure 7.5: Specifying remote-object assertion constructors with a custom validation lifetime.

method `validate` is invoked for every asynchronous message which is sent to the contracted far reference.

This validation lifetime of `ForeverAssert`, is implemented by always returning `true` when the `validate` method of the `lifeTimeObject` is invoked. This validation strategy is also the default validation lifetime when creating a far reference assertion with the keyworded function `Assert`. Note that our default strategy is not implemented by means of a lifetime object as in this particular variation the test can be omitted entirely leading to a more efficient implementation. `OnceAssert` keeps track of the amount of messages that are being sent to the contracted remote object. Every time a message is sent to the far reference the local variable `count` is increased. By comparing the count variable with 1 the only time that validation takes place is when the first message is sent.

The variations related to when to validate the assertion, once or forever, are extremes of the spectrum and other factors can play an important role in determining the lifetime of the flat contract over the far reference. One factor is time, for example when the programmer knows that certain guarantees are subject to a lease. Another factor is the messages which are sent to the far reference. For example, in a virtual shop a flat far reference could validate a certificate only before sending the payment message. By opening up the interface of the flat far reference contracts the programmer can implement his own variations in the lifetime of the contract.

7.1.3 Atomicity

The actor model of AmbientTalk guarantees that when an actor `A` sends two successive messages (M_{a1} and M_{a2}) to an actor `B`, the actor `B` processes these two messages in the same order i.e. M_{a1} is always processed before message M_{a2}. However, the AmbientTalk actor model does not guarantee that the actor `B` processes these messages without interleaving. For example, when a third actor `C` also sent a message to the actor `B` there are three ways in which the messages can be processed: $M_c.M_{a1}.M_{a2}$, $M_{a1}.M_{a2}.M_c$ and $M_{a1}.M_c.M_{a2}$. In this last ordering the message M_{a1} and M_{a2} are interleaved with the message M_c. Interleaving of messages poses a problem for the validation of far reference assertions that does not manifest itself in a synchronous world.

```
AtomicAssert: {|p| p.notInGame() } -> any;
```

Figure 7.6: Specifying a contract with an atomically executed assertion.

Recall that flat far references are validated when they are used, i.e. when a message M_r is sent to the contracted far reference. Instead of sending this message directly the contract system first sends a validation message M_v, when this validation message is processed the actual message M_r is sent. Therefore it is possible that a message alters the state of the actor between the validation of the assertion and sending the asynchronous message M_r. The message that is processed in between can potentially violate the assertion. Only when the assertion is defined over an immutable value this is not a problem. However, in case the assertion depends on mutable values it is necessary to guarantee that validating the assertion and processing the first asynchronous message is executed without interleaving.

Therefore in our contract system we have defined a language construct that allows the programmer to define assertions that are atomically evaluated at the server side. The way we have done this is by embedding the validation of the far reference contract in the message M_r. In figure 7.6 the contract for the coordinator is rewritten by making use of such `AtomicAssert`. Note the use of the synchronous dot operator instead of the asynchronous arrow operator. Because, the assertion is evaluated at the server side there is direct access to the coordinator and the assertion can be executed synchronously.

The use of non-atomic assertions is still useful for the specification of assertions that involve multiple client objects. An example of such an assertion is a predicate that defines that two players can only shoot each other when they are in opposite teams. As this assertion depends on more than one far reference it is impossible to execute it synchronously. Note that in such a case the teams of the contract should be immutable.

7.1.4 Flat Far Reference Contracts Overview

There are two dimensions that we have explored for the formulation of flat far reference contracts, the validation lifetime and the atomicity guarantees. Each dimension leads to a different variation of the basic flat far reference contract. In Table 7.7 these two dimension are summarized.

Flat atomic, means that the contract is validated once and that the system guarantees that when the first message is processed the contract holds. For flat asynchronous assertions there is no guarantee from the system that the contract still holds when the first message is processed. It is the programmer's responsibility to ensure that the assertion only depends on immutable values.

Invariant atomic means that the contract is validated whenever an asynchronous message is sent and the system guarantees that the contract holds when the asynchronous message of the client to the server is processed. Finally, invariant asynchronous is a contract that is validated whenever an asynchronous message is sent. The system does not guarantee that the contract holds when the flat contract is defined over mutable values. Note that this table only shows those contracts which are supported out of the box. Our implementation allows the meta programmer to make variations on these flat contracts by specifying a custom validation lifetime object.

	Atomic	Asynchronous
Flat	First message No interleaving	First message No guarantees for interleaving
Invariant	Each message No interleaving	Each message No guarantees for interleaving

Figure 7.7: Assertion variations supported by flat far reference contracts.

```
def coordinator := object: {
  // (p.team() == this.team) -> bool;
  def makeMove( p ) { ... };
  def getTeam() { ... };
};
//exporting code omitted
```

Figure 7.8: Coordinator (Actor)

7.2 Far Reference Contracts: The Problems

In this section we give an overview of the problems that arise when validating a contract over a far reference. We show three (naive) approaches and verify whether they introduce deadlocks or change the message ordering. From these approaches we infer that the validation of assertions defined over far references *before* executing a contracted message is not possible. In section 7.3 we present our solution to validate far reference contracts.

7.2.1 Running Example

Consider the ambient game application again. After enough players have joined the game, players can make a move by sending a message `makeMove` to their coordinator. Pseudo code for the `makeMove` method of the coordinator and the contract defined over this method is shown in Figure 7.8. The precondition of the contract `p.team()== this.team` specifies that the provided player should be from the same team as the coordinator (assume that in this variation of the game the player could be involved in a game hosted by another coordinator with another team). The postcondition of the contract, `bool`, specifies that the return value has to be a boolean. Recall that while the precondition of this contract is a simple predicate, in an distributed asynchronous model verifying a precondition over a remote object requires an asynchronous message to be sent to this object.

7.2.2 Coarse-Grained Blocking: Delay all Message Processing

A first approach for the validation of an asynchronous contract consists of blocking all incoming messages of the actor until the contract is validated [1]. Unfortunately this validation strategy can introduce deadlocks which are not present in the non-contracted code. A client that introduces a deadlock in the validation of the contract over the `makeMove` method can be easily created as shown in Figure 7.9. To clearly show why there is a deadlock in this example we step through the validation process step by

[1]Note that this would require us to break the AmbientTalk concurrency model.

```
1  //...
2  def player := object: {
3    def getTeam() { currentCoordinator<-getTeam() };
4  };
5
6  //discovery code omitted
7  currentCoordinator<-makeMove(player);
```

Figure 7.9: Deadlock Example: Client (Actor)

step. In this example we deliberately use server instead of supplier to emphasize the distributed nature of the example.

1. The client discovers the server (`currentCoordinator`) (line 6).
2. The client sends an asynchronous message `makeMove` to the server (line 7).
3. The server receives the message `makeMove`(line 3, of the server).
4. The contract system of the server blocks all incoming messages to validate the contract.
5. In order to validate the contract, the contract system sends a message `getTeam` to the player hosted by the client in order to retrieve the team of the player.
6. Upon receiving the message `getTeam` the player object — hosted by the client — sends a message `getTeam` back to the coordinator (line 3).
7. The contract system however, blocks all incoming messages and the system is in a deadlock.

We conclude that *coarse-grained blocking is not a viable option for the validation of flat reference contracts* because they can easily introduce deadlocks. Worse, even more clever coarse-grained blocking techniques to support reentrance would not provide a viable option because the remote player could disconnect and leave the coordinator in a blocked state.

7.2.3 Fine-Grained Blocking: Delay the Contracted Message

A less radical approach does not block the entire actor but only delays processing the asynchronous message until the contract is validated. The advantage of this approach is that, when we assume that there are only two parties, it correctly validates the contract shown in the previous section i.e. (in step 7) the message `getTeam` from the player to the coordinator is processed and the deadlock situation is resolved. However, most actor systems [Yon90, DVM^{+}05, MTS05] provide a minimal guarantee on the ordering of the messages which is known as the *transmission ordering law* as shown in section 3.1.1.

This law is violated when the contract system delays processing the contracted message until the contract is verified. In order to illustrate this issue, consider the code to add and remove players from a game, shown in Figure 7.10 and 7.11. In this example `addPlayer` has a flat far reference contract that states that the (remote) player should not be in a game when added. Further, the contract also states that the return value of the `addPlayer` method has to be a boolean value. When a client sends a message `addPlayer`, the contract system validates the contract. In order to validate this contract, the contract system is forced to send an asynchronous message back to the (remote) player. As this approach tries to avoid the deadlock situation the server (coordinator) starts processing the rest of its inbox until the answer is retrieved.

```
def coordinator := object: {
  // p.notInGame() -> bool
  def addPlayer( p ) { ... };
  def removePlayer( p ) { ... };
};
//exporting code omitted
```

Figure 7.10: Transmission ordering: Server side (Actor)

Consider now the situation in Figure 7.11, the client sends two messages to the coordinator: `addPlayer` and `removePlayer`. The coordinator postpones the execution of the `addPlayer` message until the contract is validated. Therefore, the coordinator processes the message `removePlayer` first. This is clearly unwanted behavior because removing a player that was not added could lead to errors or at least to unexpected behavior when at a later moment in time the `addPlayer` message is processed.

With this small example, we have illustrated that this validation strategy can alter the semantics of the underlying programs in a significant way. We conclude that *fine-grained blocking is not a viable option to validate flat far reference contracts because this validation strategy violates the transmission ordering law.*

```
//discovery code of omitted
currentCoordinator<-addPlayer(p);
currentCoordinator<-removePlayer(p);
```

Figure 7.11: Transmission ordering: Client side (Actor)

7.2.4 Contract Decomposition

The last implementation strategy consists of decomposing the asynchronous contract into a number of synchronous contracts. Instead of validating the contract at the server-side of an object the contract is decomposed into two synchronous contracts, one for the sender and one for the receiver of the message. The contract verifies the preconditions synchronously at the sender side and the postconditions are validated at the receiver of the asynchronous message.

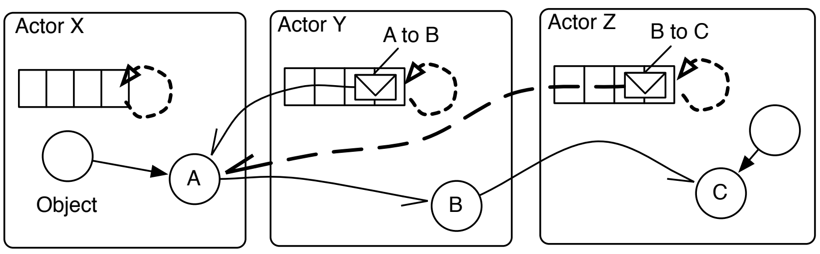


Figure 7.12: Transitively passing far references.

The advantage of this strategy is that it allows the previous asynchronous code to be executed synchronously. This approach works well in the examples that we have given but fails when the arguments passed in an asynchronous message are also remote to the sender of the message. Figure 7.12 illustrates this situation by showing the interactions between 3 actors `X`, `Y` and `Z`. An object reference of the object `A` owned by actor `X` is transitively passed from actor `X` to actor `Y` and finally to actor `Z`. When a contract is defined over the message sent from object `B` to `C` it is impossible to decompose the contract in a synchronous part which is executed on actor `Y` before the message is sent. The reason is that the object `A` is not local for the sender, i.e. actor `Y`. We conclude that *decomposing contracts into synchronous parts is also not a viable option*.

7.2.5 Conclusion

A summary of all the options in order to validate assertions over remote object references is shown in Figure 7.13. The only approach that can offer non-inference with respect to deadlocks and message ordering is contract decomposition. However this approach is not a general solution as it can not deal with transitively passed references. The main problem of the approaches shown is that they try to validate the contract *before* processing the asynchronous message. *To solve these issues we propose a contract system that postpones the validation of far reference contracts until they are used.* As we show in the next section such a contract system can deal with assertions over far references while staying inference-free with respect to deadlocks and message ordering. Moreover our system does not have any problems with transitively passed far references.

	Deadlock Free	Ordering	Transitivity
Blocking Actor	no	yes	yes
Blocking Message	yes	no	yes
Contract Decomp.	yes	yes	no

Figure 7.13: Validating asynchronous contracts *before* processing a contracted message.

7.3 Validation and Blame Assignment of Flat Far Reference Contracts

In this section we introduce the semantics of the flat far reference contract system by presenting a minimal implementation.

7.3.1 Far Reference Contracts Overview

They key point of the asynchronous message contract systems is to *postpone* the validation of far reference assertions until the far reference is actually used in a message expression. Concretely this implies that a function that has a far reference contract defined over one of its arguments is always executed immediately without validating the contract immediately. The contract system postpones the validation of far reference assertions until the first asynchronous message is sent to the contracted far reference.

Figure 7.14 shows the sequence diagram of a typical asynchronous contract. In this example the server is the actor that hosts the object over which the client defines a contract. The server first sends a message M to an object over which the client has defined a contract. This contract defines an assertion over the far reference R contained in the received message. Before processing the message from the server, the client first processes the messages in its inbox which were received before the server's message. Afterwards the client takes the server's message from his inbox and starts processing the message. Processing of the message proceeds without verifying the assertion defined over the far reference R. The assertion is verified as soon as the first asynchronous message (M_f) is sent to the far reference R. Instead of sending this message (M_f) immediately the contract system verifies the assertion by sending an asynchronous message M_c to the contracted far reference. When the assertion is verified the contract system either signals a violation of the contract or continues with sending the original asynchronous message M_f. Successive messages to the same contracted entity are handled in a FIFO manner in order to preserve the transmission ordering law.

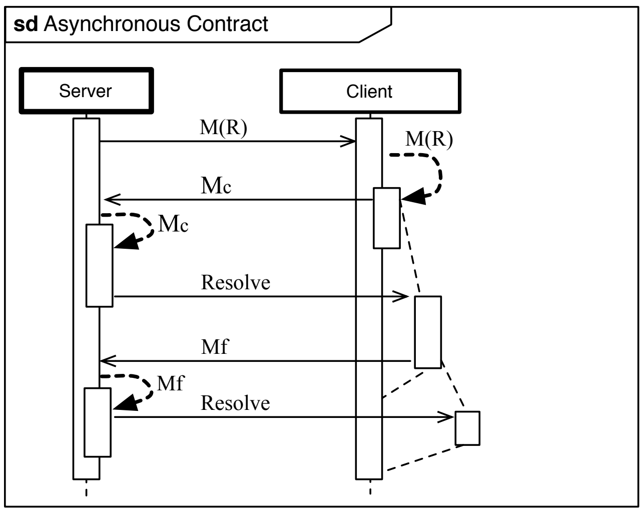


Figure 7.14: Asynchronous Contract: Sequence Diagram

The programmer can not distinguish the execution with contract validation from a run without contract validation other than observing a small delay in which the asynchronous messages are processed. As our contract system does not block the reception of successive messages at any moment in time there is no possibility that deadlocks are introduced nor that the transmission ordering law is violated. Finally as our validation strategy postpones the validation of the assertion until the far reference is actually used our approach scales for multiple communication partners that transitively exchange far references.

7.3.2 Flat Far Reference Contracts Implementation

The constructor function of the flat reference assertion is shown in Figure 7.15.

```
1  def Assert: pred  {
2   { |pos, neg|
3    { |val|
4     if: (is: val taggedAs: FarReference) then: {
5      eventualref: { |rcv,msg|
6       when: ( pred(val) ) becomes: { |res|
7        if: (res) then: {
8         val <+ msg;
9        } else: { blame(pos); };
10      };
11     };
12    } else: { blame(pos); }
13   };
14  };
15 };
```

Figure 7.15: Flat far reference contract constructor.

Analogous to the constructor function for a flat contract (c.f. section 5.1) the return value of this function is a closure which expects two blame labels (line 2). When this closure is applied it returns a new closure that expects a far reference as argument (line 3). The far reference assertion verifies that the argument `val` is a far reference by inspecting the type-tag of `val` (line 4). When a non-far reference value is supplied it assigns blame to the positive label (line 12). Instead of verifying the predicate immediately the flat far reference contract returns a proxy to the reference constructed by the keyword message `eventualref:` (line 5). The closure passed as an argument is applied whenever a message is sent to the newly created reference. The arguments of this closure are the receiver (`rcv`) of the message and the message itself (`msg`). The proxy reference verifies the predicate (line 6) and only when the predicate returns a future that resolves to true, the message is sent [2] (line 8). Otherwise blame is assigned (line 9). The flat far reference contract shown in Figure 7.15 is thus the non-atomic verify-forever variant.

Step-by-step Example In order to exemplify the internal working of the flat far reference contracts consider the example shown in Figure 7.2 again. We now give an overview of how this contract is validated by using the box notation introduced in section 5.3.1. In this notation the contract defined over the `addPlayer` method can be represented as follows:

$\boxed{l_1, l_2 \mid \texttt{!InGame} \rightarrow \texttt{any?} \mid \texttt{addPlayer}}\texttt{(p)}$

The blame labels l_1 and l_2 represent the supplier and the consumer of the contracted value respectively. When this method is applied to a far reference `p` it verifies the domain contract, namely that the supplied far reference is a player which is not yet part of a game. Because this can not be checked immediately when `addPlayer` is applied, the far reference `p` is wrapped in a new contracted far reference as shown below. Recall that inside of the body of the `addPlayer` method an asynchronous message `getName` is sent.

$\boxed{l_2, l_1 \mid \texttt{!InGame} \mid \texttt{p}}\texttt{<-getName()}$

When sending the message `getName` the code is conceptually expanded to the code

[2]The `<+` operator is used to sent first class messages.

```
when: (p<-notInGame ) becomes: { |res|
  if: (res) then: {
    p<-getName();
  } else: { blame(l₂); };
};
```

Figure 7.16: AddPlayer contract validation example.

shown in Figure 7.16. As can be seen in this figure, in case that the player was already in a game blame is assigned to blame label l_2, namely the user of the coordinator object. This is correct as it was the client of the coordinator object that supplied a player who was already in a game.

This concludes the basic exposition of flat far reference assertions. In the next section we show how flat far reference contracts can be extended to support higher-order far references.

7.4 Higher-Order Far Reference Contracts

In the previous sections we explored the definition of *flat* contracts over far references. In contrast to flat contracts in a synchronous world, the validation of flat contracts in an asynchronous world needs a validation strategy that is similar to the validation of higher-order contracts. Instead of validating the contract immediately, validating the contract is postponed until the contracted value is used. The specification of flat far reference contracts has been limited to validating simple predicates over a far reference. Now we show how we can also define properties over the messages that are sent to a far reference. Higher-order far reference contracts are very similar to object contracts in a synchronous program as shown in chapter 6. In this section, we give an overview of how to define and validate higher-order far reference contracts.

7.4.1 Higher-Order Far Reference Example

A higher-order far reference contract specifies the *asynchronous* messages that need to be supported by the far reference over which the contract is defined i.e. its interface. Contrary to the contracts shown in chapter 6, a higher-order far reference contract is specified by the client, i.e. the client defines a contract over an object hosted by a remote actor. We explain higher-order far reference contracts by means of a distributed example where a far reference to a stack of integers is sent from the server to a client. Imagine that the client wants to protect himself from getting a bad far reference. Therefore, the client defines a higher-order far reference contract over the stack received from the client. In the next sections we explain how such a contract can be encoded in AmbientTalk/C.

Client receiving a stack in an asynchronous message The code of the client that defines a higher-order far reference contract over the remote stack is shown Figure 7.17. At lines 6 till 10 a client object `c` is defined, it has one method `receiveStack` which takes a single argument, a far reference to a stack `s`. At line 8, the client sends an asynchronous message `push` with the argument `1` to the remote stack.

At line 13 till 17, a contract for the elements that can be pushed and popped from a stack is shown. The definition of this contract follows exactly the same syntax as the definition of an object contract as shown in chapter 6. It is only when instantiation this contract that the difference between a local object contract and a higher-order far reference contract becomes visible. The contract specifies that there should be two methods available namely `push` and `pop`. The contract defined over the `push` method states that the argument should be an integer. Similarly, the values that are popped from the stack should be integers.

The client needs to export the object `c` in order for the server of the stack to discover the client. The client thus specifies a second object contract (`clientContract`) for the exported object `c` (lines 19-24). This contract specifies that the object over which the contract is define needs to support the method `ReceiveStack`. The `ReceiveStack` method contract specifies that the argument must be a far reference to an object obeying the `StackContract`. The higher-order far reference contract is created with the contract constructor `FarRef`. *The programmer first specifies object contracts with* ***`ObjectContract`*** *but in order to create either a local object contract or a higher-order far reference contract he needs to apply the* `Ref` *contract constructor seen in chapter 6 or the* `FarRef` *contract constructor shown here.*

Finally, at the end of the module the client object is exported to the network with the previously defined client contract (line 27). Note that exporting the object is done with a `Ref` contract not a `FarRef` contract (line 27). The reason is that the object `c` is local to the client, which exports the contract. From that moment that the object is exported, collocated nodes in the network can discovered this object and send messages to it, given that they follow the contract. In the next paragraph, we show how the server discovers the client and sends an asynchronous message to the client.

Server sending an asynchronous message to the client. The server side of the higher-order far reference contract example is shown in Figure 7.18. In this example, first a stack object is defined (line 4 till 11). Then a **`when:discovered:`** listener is registered to discover the client. Once the client object is discovered the server sends this client object a message `receiveStack` with the newly created stack. When this message is sent the client receives a far reference to the stack. At the client side a contract is applied to this far reference in order to validate which messages are sent over the far reference.

Blame Imagine that in the example we made a trivial error by sending a message `push` with the argument `"Wrong"` (line 8 in Figure 7.17). As the higher-order far reference contract specifies that this argument should be an integer the contract system stops the ongoing computation and throws a contract validation exception. Blame is assigned to the "Far-ref-Uses.at" module.

7.4.2 Computational Far Reference Contracts

Recall from chapter 5 that a computational contract allows the programmer to define properties over the internal execution of the contracted entity. AmbientTalk/C allows computational contracts to be defined over far-references. This allows the programmer to prohibit or ensure that certain functions and methods are applied when a asynchronous message to a far reference is processed. Figure 7.19 shows an example where the `push` method of a far reference stack is prohibited to apply the `system.println` method. This contract is very similar to the one shown in Figure 7.17. However, now the `push` contract has a computational contract defined over it. This contract is

```
//Far-ref-Uses.at


//Implementation
deftype Client;
def c := object: {
 def receiveStack(s) {
  s<-push(1);
 };
};

//Contracts
def StackContract :=
ObjectContract: {
 def push(arg) { int -> any };
 def pop()     { any -> int };
};

def clientContract :=
ObjectContract: {
 def receiveStack(s) {
  FarRef(StackContract) ->  any;
 };
};

//Exporting the client
export:  c as: Client withContract: Ref(clientContract);
```

Figure 7.17: Far Reference Contracts: Stack Example (Uses)

```
//Far-ref-Defs.at
deftype Client;

def stack := object: {
 def storage := [];
 def isEmpty() { storage.size == 0; };
 def push(arg) { storage := [arg] + storage; };
 def pop() {
  //...
 };
};

when: Client discovered: { |c|
 c<-receiveStack(stack);
};
```

Figure 7.18: Far Reference Contracts: Stack Example (Defs)

created with the `prohibit_c` constructor function and prohibits `push` to invoke `system.println` (line 3). Note that the selector of a far reference computational contract needs to be an isolate in order to be sent by copy to the server. When the `prohibit_c` contract constructor receives such an isolate it creates a computational contract that is copied to the client when an asynchronous message to the contracted far reference is sent.

```
1  def StackContract :=
2  ObjectContract: {
3   def push(arg) { int -prohibit_c( isolateClosure: { system.&println } )-> any };
4   def pop()     { any -> int };
5  };
```

Figure 7.19: Computational Far Reference Contract: Stack Example (Defs)

In order to show an example of a violation of this contract, we added an extra line in the `push` method at the server side (figure 7.18) that prints the objects that are being pushed on the stack (line 7). Figure 7.20 shows that when pushing elements on this stack, blame is correctly assigned to the `"FarrefDefs.at"` module.

```
14:21:Far-ref-Defs.at violated the contract computational contract.
```

Figure 7.20: Computational far reference contract violation.

7.5 Validation and Blame Assignment of Far Reference Contracts

The implementation of higher-order far reference contracts is shown in Figure 7.21[3]. Higher-order far reference contracts closely resemble the implementation of object contracts. The difference is that instead of making a wrapper object a wrapper reference is created. Such a wrapper reference is created with the keyworded message `eventualref: block to: value` (line 4). Every time a message is sent to the wrapper reference the block is applied with the receiver (`rcv`) of the message (the original reference) and the message (`msg`) itself.

```
1  def farRef( ObjectContract ) {
2   def contract(pos, neg) {
3    { |val|
4     if: (is: val taggedAs: FarReference) then: {
5      /.at.lang.firstclassrefs.eventualref: { |rcv,msg|
6        def mContract := ObjectContract.getContract(msg.selector);
7        def dom       := mContract.domain.contract(neg, pos);
8        def rng       := mContract.range.contract(pos, neg);
9        def cc        := mContract.computation.contract(pos, neg);
10       msg.arguments := dom(msg.arguments);
11       msg           := cc(msg);
12       when: (rcv <+ msg) becomes: { |value|
13         rng(value);
14       };
15      };
16     } to: val;
17    } else: {
18     this.blame(pos, " A far reference contract over a non far reference:: " + val);
19    }
20   }
21 }
```

Figure 7.21: Implementation of Far Reference Contracts

When the contract is applied to a value `val` (line 3), the type tags of the value are inspected in order to make sure that the value is indeed a far reference (line 4). If this is not the case positive blame is assigned (line 18). When the value was a far reference a wrapper reference is created to intercept all asynchronous message sends (line 5). Every time a message is sent to the wrapper reference the contract defined over this message is extracted from the **`ObjectContract`** (line 6). From this contract the domain, range and computational contract are initialized with the appropriate blame

[3]This is pseudo code but it closely follow the actual implementation.

labels (line 7-9). Then the domain contract is applied over the arguments of the message (line 10). A modified asynchronous message is created, which when received by the server applies the computational contract (line 11). The modified message is sent to the original reference `rcv` with the first class message operator (<+) (line 12). Sending an asynchronous message returns a future which will be resolved when the message is processed. The **`when:becomes:`** construct is used in order to retrieve this value. The range contract is applied when the **`when:becomes:`** block is triggered (line 13).

Recall that the result of a **`when:becomes:`** block is a future. This future resolves to the value of evaluating the callback block. Since the **`when:becomes:`** block at line 11, is the last expression of the wrapper reference interception block, this is also the value that is returned when sending a contracted asynchronous message. By making use of future pipelining, the use of the wrapped reference is made transparent for the programmer.

7.6 Computational Message Contracts

Recall from chapter 5 that a computational contract allows the programmer to define properties over the internal execution of the contracted entity. In a distributed context one important aspect over the execution of a method is which messages are being sent during the execution of this method. Just like making certain method calls mandatory or disallowing method calls, it is important to be able to disallow or validate that certain messages are sent asynchronously. In order to support such contracts we have extended our aspect language from section 5.3.2 with a special point-cut which allows the programmer to define aspects that intercept message sends. This hook makes it possible to define computational contracts that validate which messages are sent. We first give an overview of message contracts from a programmers perspective and then show the aspect extensions that are needed in order to define message contracts.

7.6.1 Message Contracts

The AmbientTalk/C framework provides constructor functions in order to specify message contracts similar to the ones shown in chapter 5. An overview of these constructor functions is shown in Figure 7.1. Ambient Contracts allow the programmer to specify contracts for mandatory message sends with `ensure_m`. Similarly, the programmer can create a contract to disallow sending a certain message by means of the function `prohibit_m`. These functions also make it possible to create a contract that monitors a sequence of outgoing messages by means of a protocol. We start our exposition of message contracts with prohibit message contracts in section 7.6.2. Subsequently we show ensure message contracts in section 7.6.3. Ensure and prohibit message contracts by means of a protocol are shown in section 7.6.4.

	A single message	Sequence of messages
Mandatory	`ensure_m`($\mathtt{M}_{pc}$)	`ensure_m`($M_{protocol}$)
Disallow	`prohibit_m`($\mathtt{M}_{pc}$)	`prohibit_m`($M_{protocol}$)

Table 7.1: Computational contracts over messages.

7.6.2 Prohibit Message Contract

With AmbientTalk/C creating a contract to prohibiting a message from being sent is done with the contract constructor function `prohibit_m`. In order to create a contract that prohibits a message to be sent during the execution of a contracted method, the developer needs to specify which message is disallowed. This is done by means of the keyworded function `message:`, which expects a string that is matched against the name of the outgoing message. Figure 7.22 shows an example of a prohibit message contract. In this example, `prohibit_m( message: "disallowed")` prohibits the message `disallowed` to be sent within the dynamic extent of the contracted function. Recall that there are two main mechanism active during the enforcement of a prohibit contract: the interception mechanism and the blame assignment mechanism. The interception mechanism intercepts all prohibited messages *before* they are actually sent. At that moment, control is transferred to the blame assignment mechanism which stops the entire computation and presents the programmer with an error message explaining who violated the prohibit contract.

```
provide: f withContract: any -prohibit_m(message: "disallowed")-> any;
```

Figure 7.22: Providing a function `f` with a prohibit computational contract for messages.

When the contracted entity sends a message which is disallowed by the computational message contract, blame is assigned and the programmer is presented with an error message similar to the one shown in Figure 7.23.

```
65:5:defs.at violated the contract. Computational message contract violation:
Intercepted sending message "disallowed".
```

Figure 7.23: Computational message contract violation.

7.6.3 Ensure Message Contracts

The dual of prohibiting that a certain message is sent, is to *ensure* that a message is sent. An ensure contract verifies that a certain promise is kept during the dynamic extent of the contracted function. For example, a function `g` can promise to sent a message `foo`. An ensure contract does not specify the exact moment when the message `foo` needs to be sent, it only specifies that at some moment during the execution of the contracted function `foo` needs to be sent. Verifying an ensure contract is thus more subtle than verifying a prohibit contract as blame can only be assigned *after* the contracted method has been entirely executed (after all sending the message `foo` could be the last statement of the contracted method). From a programmer's perspective, defining an ensure contract with computational contracts is as simple as defining a prohibit contract. Ensure contracts are created with the function `ensure_m`. Figure 7.24 shows an ensure message contract that verifies that the message `buy` is sent within the dynamic extent of the function under contract.

When the contracted method does not sent the mandatory message the computational message contract blame assigns blame and the programmer is presented with an error message similar to the one shown in Figure 7.25.

```
provide: f withContract: any -ensure_m(message: "buy")-> any;
```

Figure 7.24: Providing a function `f` with an ensure computational contract for messages.

```
65:5:defs.at violated the contract. Computational message contract violation:
Did not sent message "buy".
```

Figure 7.25: Computational message contract violation.

7.6.4 Protocol Message Contracts

The message transmission ordering law states that the order in which messages are being sent to a certain receiver is also the order in which these messages are being received. Therefore it might be important for the programmer to explicitly capture the order in which message are transmitted. Outgoing messages can be validated by a message ordering protocol which is similar to the way in which object protocols are encoded. An example of a sequence protocol to buy a product is shown in Figure 7.26. First, there is sequence protocol defined with the keyworded message `MessageProtocol` (line 1-11). This protocol has four states `init`, `next`, `end` and the mandatory `start` state. The `init` state has one transition `login` that is triggered when there is an outgoing message with the same name (line 3). The `next` state transitions the finite state machine to the `end` state when a message `logout` is sent (line 7). The `next` state transitions again to the `next` state when a message `buy` is sent (line 8). The `end` state does not allow any messages mentioned in the protocol to be send.

A function `buyforme` that satisfies this protocol is defined at lines 14 till 18. After sending the messages `login`, `buy` and `logout` messages, the message protocol is in the end state. In this state no messages are allowed to be sent. Finally, the function `buyforme` is exported with the `sequenceProtocol` defined earlier.

```
1  def sequenceProtocol :=  MessageProtocol: {
2    def init() {
3       (on: "login") =>  { next()  };
4    };
5
6    def next() {
7      (on: "logout") => { end() };
8      (on: "buy") => { next()  };
9    };
10   def end() { };
11   def start() { init(); };
12 };
13
14 def buyforme(user,item) {
15  o<-login(user);
16  o<-buy(item);
17  o<-logout(user);
18 };
19
20 def Interface := object: {
21  def seq := provide: seq withContract: any -ensure_m(sequenceProtocol)-> any ;
22 }
```

Figure 7.26: Message ordering contract example.

In case the `buyforme` function would send a `buy` message before sending the `login` message blame would be assigned as shown in Figure 7.27.

```
154:5:uses.at violated the contract
Message protocol violation possible transitions: [ login ], while sending message: buy
```

Figure 7.27: Message ordering contract example.

7.7 Validation of Messages Contracts

The validation and blame assignment of message contracts is exactly the same as the computational contracts outlined in section 5.3.3. However, we have not shown how messages can be intercepted by the underlying aspect language. In this section we show how the underlying aspect language can intercept outgoing messages and subsequently give the definition of a `prohibit_m` contract.

AmbientTalk has a strong reflective layer which allows the core language to be extended with new programming constructions. The reflective layer of AmbientTalk is based on mirrors [BU04, VMG$^+$07], meta-level objects that allow the programmer to reflect upon the actors state and behavior. In this reflective layer there is a method called `send` which allows the programmer to intercept the outgoing message of an actor. This function is applied whenever a message is sent. It receives two arguments: the receiver and the outgoing message.

The way to intercept message sends is thus exactly the same as intercepting a regular method invocation. An example of defining an aspect that throws an error when a message is sent is shown in Figure 7.28. At line 1, the aspect specifies to intercept all invocations of the `send` method of the actor mirror. This actor mirror is obtained by calling the function **`reflectOnActor`**. Next this aspect is deployed (line 5) over a block of code that sends an asynchronous message `buy` to an object `o` (o is not shown in the figure). As expected an error is thrown when executing this piece of code.

```
1  def a := aspect: call(reflectOnActor().&send)  advice: { |proceed, arguments|
2    error: "Sending messages is not allowed in this code block!";
3  };
4
5  fluid: a deploy: {
6   o<-buy();
7  };
```

Figure 7.28: Aspect example of intercepting message sends

By making use of the reflective layer of AmbientTalk there are no special constructions needed in order to define message contracts. The implementation of message contracts thus follows the implementation of computational contracts as shown in section 5.3.3. The didactical implementation of a prohibit message contract is shown in figure 7.29. As can be seen on line 2, it is a computational contract created with the contract constructor `cc` shown in section 5.3.2. This computational contract intercepts all outgoing messages sends. When a message is intercepted the contract validates that the name of this message equals to the prohibited message. If this is the case blame is assigned. Otherwise the proceed function `p` is applied to the original arguments.

```
def prohibit_m(name) {
 cc(call(reflectOnActor().&send),
  {|pos,neg|
   {|p,a|
    if: ( a[2].selector == name) then: {
     blame(pos)
    }else: {
     p(a)
    };
   };
  },
  flat( {|x| true;}),
  flat( {|x| true;}))
};
```

Figure 7.29: Prohibit message contract constructor.

7.8 Conclusion

In this chapter we have shown that the use of contracts in an asynchronous context poses a number of problems when flat contracts are defined over far references. Subsequently we have shown that our solution validates far reference contracts without introducing deadlocks while preserving the order in which messages are processed. We have also shown that the ideas of higher-order object contracts can be extended to support higher-order far reference contracts. Finally, we have shown how computational contracts can be extended in order to validate the trace of outgoing messages.

The main contributions presented in this chapter are:

- Flat contracts over far references.
- Higher-order far references contracts in an asynchronous distributed object model.
- Computational message contracts.

In the next chapter we give an overview of a concrete application of the contract system developed in the context of this dissertation.

Chapter 8

Ambient Contracts at Work

In this dissertation, we propose AmbientTalk/C as a framework used to monitor the usage of object references that are exchanged between the distributed modules of an ambient application. Ambient contracts allow the specification of behavioral constraints such as: are messages allowed to be sent during the execution of a certain method, are files being accessed when applying certain methods, etc. In previous chapters we have shown numerous small didactical examples which usually involved toy modules that should or should not print some text on the prompt. These examples contribute to the understanding of the basic concepts but are not representative of the actual use of our contracts in a concrete application. In this chapter we evaluate the use of the AmbientTalk/C framework for a concrete use case. The examples of computational and ambient contracts that are shown in this chapter all revolve around the implementation of a framework for the development of ambient social applications called UrbiFlock [GBLCS$^+$11]. The development of the UrbiFlock framework is a joint effort of the Ambient Group at the Software Languages Lab. The explanation of the UrbiFlock framework in this chapter follows the explanation as given in [GBLCS$^+$11].

Our choice for applying ambient contracts over this framework is motivated by the fact that this is the most sophisticated framework developed so far in the AmbientTalk language. Not only is it the biggest (in lines of code) framework developed, it also combines several previous research efforts w.r.t connectedness and context-awareness into a single *smart* application.

In this chapter, we report on our experiences from applying ambient contracts over the most important modules of the UrbiFlock framework. We show the different modules involved and give an overview of the contracts that are applied. As will be shown, applying our contracts in UrbiFlock can be compared to writing detailed documentation. In some cases it was even possible to make a one-to-one mapping of code comments to actual contracts. We also give an initial assessment of the complexity of using ambient contracts and show where they have aided in finding bugs in the UrbiFlock framework. Finally, we also mention those part of the AmbientTalk/C framework where our contracts could be improved or have been improved by applying them to the UbriFlock framework.

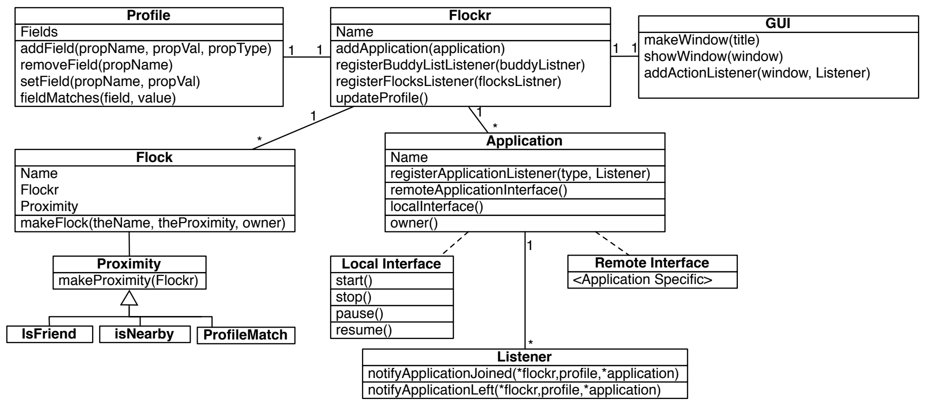

Profile
Fields
addField(propName, propVal, propType)
removeField(propName)
setField(propName, propVal)
fieldMatches(field, value)
1
1
Flockr
Name
addApplication(application)
registerBuddyListListener(buddyListner)
registerFlocksListener(flocksListner)
updateProfile()
1
1
GUI
makeWindow(title)
showWindow(window)
addActionListener(window, Listener)
1
*
Flock
Name
Flockr
Proximity
makeFlock(theName, theProximity, owner)
1
*
Application
Name
registerApplicationListener(type, Listener)
remoteApplicationInterface()
localInterface()
owner()
Proximity
makeProximity(Flockr)
IsFriend
isNearby
ProfileMatch
Local Interface
start()
stop()
pause()
resume()
1
Remote Interface
<Application Specific>
*
Listener
notifyApplicationJoined(*flockr,profile,*application)
notifyApplicationLeft(*flockr,profile,*application)

Figure 8.1: UrbiFlock Design Diagram

8.1 UrbiFlock Framework

UrbiFlock is a framework designed for the development of applications that enable spontaneous interactions of people by exploiting new technologies such as wireless networks and mobile devices. As in Facebook, users that join Urbiflock (called *flockrs*) can meet other users and interact with them, for example by sending each other messages. Flockrs have a profile which can be browsed by other flockrs that are in proximity. The Urbiflock framework takes care of managing a flockr's friends lists, called *flocks*. A flock can be compared to a Facebook group (for example, a group of old classmates). Urbiflock additionally allows for the definition of groups of *proximate* flockrs (for example, a group of all friends that are currently nearby). Unlike current mainstream social network sites, Urbiflock allows the *specification* of flocks both in terms of *physical* proximity (defined by for example the bluetooth communication range of the flockr's cellular phones) and *semantic* proximity (e.g. in terms of being friends of other flocrks).

Similar to Facebook, third parties can build applications and plug them into the Urbiflock framework. Several core applications are currently available in the Urbiflock framework, such as flock creators and profile viewers. In the remainder of this section, we describe the main concepts of the Urbiflock framework. Subsequently, we describe how ambient contracts are applied to the different modules of the framework.

8.1.1 Architecture

Figure 8.1 shows a UML diagram of the parts of the UrbiFlock framework to which we have applied ambient contracts. Note that in this diagram many of the details are omitted.

Flockr. As can be seen the `Flockr` abstraction plays a central role in this design. A flockr has exactly one profile and can be registered to multiple `Flocks`. In addition, a `Flockr` can have various applications installed. Applications need to be explicitly added to a `Flockr` before they can be used. When an application is added to the `Flockr` with `addApplication`, the application is added to a dashboard (similar to the Home screen on the Apple iPhone). Running applications have controlled access to flockr [1] information via the framework such as the flockr profile. This is a common functionality found in a wide range of social networking applications. In addition, Urbiflock applications have access to the user's flocks (so they can talk to nearby flockrs) and the flockrs who have installed the same application on their devices.

Profile. Profiles in UrbiFlock are highly extensible and besides a number of mandatory fields, flockrs can add as many custom fields as they like (for example, they could add their year of graduation). The fields of a profile can be used to match other users in the proximity by grouping them in flocks. For example, a flockr could create a flock of nearby flockrs which graduated in the same year. When adding custom fields to the profile, the user can specify the type of the field (e.g. a number, a piece of text, a date, a choice etc.). Furthermore, the framework provides some infrastructure that makes it easy to add new custom types without having to write too much boiler plate code.

Flock. A `Flock` consists of a list of flockrs and a proximity function that determines whether or not a certain `Flockr` belongs to that list. There are several predefined proximities in the Urbiflock framework: `isFriend` encodes a friendship relationship (i.e. if a flockr is a friend of another flockr), `isNearby` encodes physical proximity relationship (currently defined by the communication range of their cellular phones) and `doesProfileMatch` tests an attribute of a flockr's profile.

Users can define their own proximity functions as combinations of existing proximities by combining them using logical operators. For instance, a user could specify a flock consisting of all female people in the neighbourhood who like to drink belgian beers. This can be encoded in Urbiflock with a proximity function that combines the physical proximity to discover nearby flockrs and matches their profile to select the nearby flockrs that are female and like drinking belgian beers.

A proximity function is recomputed whenever there is an event that alters the type of encoded relationship. These events become visible to the user as an addition or removal of a flockr in a flock. For example, if a flockr moves out of communication range, the proximity function in the nearby flock is recomputed and as a result the disconnected flockr is removed. The same happens when any of the connected flockrs adapts his or her profile, since this change may cause a flockr to enter or leave the proximity as defined by the corresponding proximity function.

The Urbiflock framework provides programmers with the necessary infrastructure to deal with the highly dynamic environment to which pervasive social networking applications running in mobile ad hoc networks are exposed. Programmers do not need to manually track the appearance and disappearance of flockrs in their environment (by means of the AmbientTalk service discovery constructs), or changes in their profiles. In addition, plug-in applications themselves can be notified of the appearance or disappearance of nearby flockrs running the same application, making it easy to have small applications interact with each other.

Application. As shown in figure 8.1 every application has two interfaces: a local and a remote one. This distinction between local and remote interfaces has been introduced for reasons of security: methods defined in the local interface can only be called by

[1] Here we refer to flockr as the user not the module.

local objects. Remote objects can only invoke methods defined in the application's remote interface. Applications can have a local interface, which can be trusted, to call different operations on the application than remote objects (e.g. changing the application's settings). Applications can register listeners that are notified when other flockrs enter or leave communication range, when they change their profile (e.g. when they update their status) or when flockrs running the same application appear in the proximate environment (e.g. to enable application-specific interaction). The latter event can be detected by registering a listener by using the `registerApplicationListener` method on an `Application`. The listener must implement two methods `notifyApplicationJoined` and `notifyApplicationLeft` which are invoked by the underlying framework when applications move in and out of reach.

8.1.2 Software Specifications in UrbiFlock before AmbientTalk/C

It has to be noted that the development of the UrbiFlock framework was developed before the AmbientTalk/C framework. After developing the AmbientTalk/C we applied ambient contracts in order to validate the applicability of our contract system by a concrete use case. However, before applying contracts for the development of the UrbiFlock framework programmers were already aware of the importance of specifying the rights and obligations of the objects that flow from one module to another. This is reflected in the code documentation dispersed all over the UrbiFlock framework. Many of this documentation are simple specifications that can easily be encoded as contracts. We have transformed these specifications into contracts and added more sophisticated (behavioral) contracts that were not documented as well.

A representative example of the documentation found in UrbiFlock is shown in Figure 8.2. This piece of documentation is taken from the application module and more specifically it is defined over the function `registerApplicationListener`. The comment informs the programmer which properties an application listener object (`l`) should adhere to. It specifies that such a listener object should implement two methods `notifyApplicationJoined` and `notifyApplicationLeft`. Furthermore, it details that the first and last arguments are far-references. Such documentation is very useful for the programmers because from this documentation they can deduce that his code should not contain any synchronous method invocations to the first and last arguments.

There are two major drawbacks of using such documentation as specifications of the software. First, the code comments are not subject to validation, which means that the code does not necessarily implements the documentation. The second problem with this code documentation is that it is mainly written as a reminder for the developer of the module. Therefore, it is written in the middle of the implementation of the module where the callback is registered. This implies that the user of the application module still needs to browse through the code in the hope that there is some interesting information in the comments that helps understanding how the module works. Moreover, when changes are made to the code, the comments can get out of sync with the implementation.

In the next sections, we show that the programming effort for applying ambient contracts is comparable with the effort required from the programmer to write documentation such as the one shown in Figure 8.2. Moreover when using ambient contracts, the programmer is less likely to give an incomplete specification such as the one shown in Figure 8.2. In this example the programmer did not specify that both methods are not expected to return any meaningful value. While this might be intuitive, a programmer confronted with the documentation can not be sure that this is the case and is

```
/* l should understand:
invoked whenever an application was discovered
notifyApplicationJoined(flockr : farref to flockr, buddyProfile : profile,
  app : farref to application)

invoked whenever an application is no longer present
notifyApplicationLeft(flockr : farref to flockr, buddyProfile : profile,
  app : farref to application)
*/
```

Figure 8.2: Comments in the Application Module.

forced to obtain this information from the implementation. In the next section we start by showing how contracts are applied in the code that implements the graphical user interface of UrbiFlock.

8.2 Contracts over the Graphical User Interface Module

The graphical user interface of the UrbiFlock framework is implemented by making use of the foreign function interface between AmbientTalk and Java. The GUI module of the UrbiFlock framework provides a small layer between the AWT classes and the rest of the UrbiFlock framework. The example of creating a window and showing it to the user is shown in Figure 8.3.

```
import /.urbiflock.gui;
def window := makeWindow("Hello World!");
showWindow(window);
```

Figure 8.3: Graphical User Interface, Hello World example.

First the `gui` module is imported and then a new window is created with the `makeWindow` function. The argument of this function is the string `"Hello World"`!. The last line of the example presents the window to the user by applying the `showWindow` function. Besides creating windows the GUI layer provides abstractions to create buttons, labels, etc. A screenshot of the window that is created when evulating the code in figure 8.3 is shown in figure 8.4.

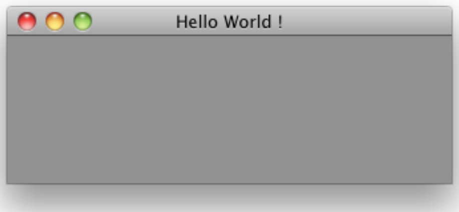


Figure 8.4: Graphical User Interface, Hello World example screenshot.

The contracts applied to the graphical user interface module are shown in Figure 8.5. We first show the custom made flat contracts followed by the computational contracts used in the module. At the end we show the *"module contracts"*, i.e. those contracts that are applied over the provided objects of the module.

```
 1  def isWindow(w)   { w.getClass == jlobby.java.awt.Frame; };
 2  def window := flat(&isWindow);
 3
 4  def actionListener := ObjectContract: {
 5    def actionPerformed(event) {  any -ensure_c(&updateWindow)-> any; };
 6  };
 7
 8  def guiInterface := object: {
 9    //...
10    def makeWindow := provide: &makeWindow  withContract: string -> window;
11    def showWindow := provide: &showWindow  withContract: window -> bool;
12    def addActionListener := provide: &addActionListener
13                             withContract: window * Ref(actionListener)  -> bool;
14    //...
15  }
```

Figure 8.5: Contracts defined over the graphical user interface module.

Flat contracts. Flat contracts are used in order to make sure that the correct Java classes are used. For example the `window` contract (line 1 and 2) is a flat contract that tests whether a given value is an instance of the Java `Frame` class. Note that in the definition of the `isWindow` predicate no explicit error handling code is installed. This is because the contract framework assumes that all exceptions thrown during the evaluation of a *flat* contract are due to improper values. For example, if the user would pass an object that does not understand the `getClass` accessor an exception is thrown. This exception is caught by the contract framework and blame is assigned.

By using flat contracts for the development of the UrbiFlock framework we found out that writing correct predicates is difficult. For example the `isWindow` predicate should actually first check that the given value is an object. Subsequently it should (through reflection) validate that the object has a field `getClass` and finally do the test that this field equals to `jlobby.java.awt.Frame`. Needless to say such code would complicate the process of writing simple flat contracts.

We also found that it is still useful to make the programmer aware that the contract fails either due to an error in the evaluation of a flat contract or because the contract failed normally.

Computational Contracts In the GUI module, programmers can register event listeners to be notified when an interesting event happens. For example, the programmer can register an event listener to be notified when a user clicks in a window. Contracts in the GUI module are used to verify that the event listeners are correctly applied i.e. that they implement the correct callback method. The `actionListener` contract shown in Figure 8.5 (line 4 – 6) specifies that an `actionListener` should have at least one method called `actionPerformed`.

Often it is important to update the GUI (repaint) when the `actionListener` event handler is triggered. This protocol is validated by a computational contract over the `actionPerformed` method of the `actionListener` (line 5). This computational contract *ensures* that in the dynamic extent of applying the `actionPerformed` method the method `updateWindow` is applied. While this contract is specified for the UrbiFlock framework, this pattern is often seen when developing GUI frameworks. Therefore, it could be reused for example, to validate that the GUI is redrawn when the model changes in the MVC design pattern.

Module Contracts. The module exports the functions `makeWindow` with a function contract (line 10) that validates that the first argument is a string and that a window is returned. The `showWindow` function also has a functional contract defined over it. This contract (line 11) specifies that the argument should be a window and that a boolean value is returned. Finally, the `addActionListener` contract (line 12) validates that the first argument is a window and the second argument an object obeying the `actionListener` contract. The return value is a boolean.

8.2.1 Experiences

The use of contract in the graphical user interface has been useful in a number of ways.

- The use of the flat contracts during the development of the library has led us to detect that the order of the arguments when registering a listener was wrong in the original code. The use of contracts helps to detect errors early. In a fully dynamic library without contracts the error would have been manifested itself in the GUI library. The use of contracts showed us immediately which module to fix.

- The use of the AmbientTalk/C framework detected a spelling mistake in the source code. The method name `actionPerformed` of one of our event listeners was written without uppercase p. The contract system detected this error and assigned blame to the `Chat` module. This was extremely useful because without the contract system an error is triggered by a callback from within the Java AWT framework. Event listeners can be registered from various places within code which means that without the contract system the programmer has to search the whole code for the wrong event listener.

- The use of computational contracts aided for detecting errors in the mandatory calls in the callbacks to the `updateWindow` function. This is was mainly because this requirement was introduced later in time to make sure that the GUI is always updated. When introducing this requirement, some places did not implement it yet and those cases were detected by the contract system during testing.

- In the testing phase of applying contracts in the UrbiFlock framework it became clear that writing flat contracts is not as trivial as it might seem. In order to specify a flat contract over the field of an object the predicate `o| o.field == 10|o| o.field == 10` is not sufficient. The programmer should actually first validate that the given value is indeed an object. Subsequently, the programmer should validate through reflection that this object has the field that he wants to inspect. And finally the programmer can access that field and to test that it equals to 10.

Usually, the programmer is concerned with writing a (partial) predicate that works for a value that satisfies the predicate i.e. `o| o.field == 10|o| o.field == 10`. It illustrated writing a (complete) predicates that work for every possible value is not trivial. Therefore, exceptions thrown from the evaluation of the predicate associated with the flat contract are treated as contract violations.

- Finally, some of the contracts have currently not aided in finding bugs but they have certainly helped the programmer in understanding the responsibilities of the graphical user interface elements.

8.3 Contracts over the Application Module

The application module of the UrbiFlock framework provides the constructor function to make new UrbiFlock applications. This module is not explicitly shown in Figure 8.1. It is a module that acts as a factory to create the kind of applications shown in the UML diagram. In order to create an application, the programmer has to provide a name for the application and the flockr that owns the application. When an application is created it is registered to the list of applications of which the user can choose. After an application is registered, the user can open the application and at later moment in time close the application. In the Android version of the UrbiFlock framework, applications can also be paused. Applications are paused by the Android operating system when the user switches back and forth between the UrbiFlock application and other applications on his phone. Whenever an application is paused, it is the responsibility of the programmer to save the state of the application and to restore this state at the moment the application is resumed.

In the remainder of this section we give an overview of the contracts applied to the applications module. Since the application module serves as template code offered to the programmer in order to develop his own UrbiFlock applications. Therefore, many of the contracts defined in this module are quite general and should be refined by the programmer when implementing custom applications.

Computational Protocol Contracts A computational contract is applied in order to validate the lifetime of an application. The state diagram that depicts the lifetime of an UrbiFlock application is shown in Figure 8.6. When the application is initialized it can be transitioned to the `running` state by applying its `start` method. In the running state the application can be either stopped or paused by applying the `stop` or `pause` method respectively. In the paused state the application can be either resumed by invoking the `resume` method or stopped by invoking the `stop` method.

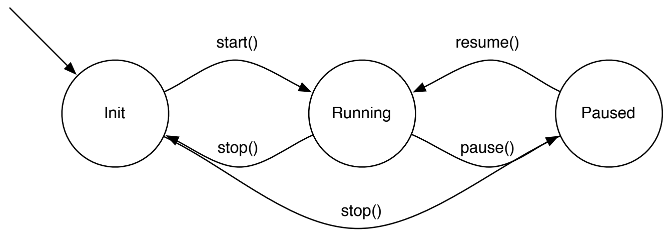


Figure 8.6: State diagram of an UrbiFlock application.

```
def ApplicationProtocol() {
  ObjectProtocol: {
    def init() {
      (on: start) => { running()  };
    };

    def running() {
      (on: stop)  => { init()   };
      (on: pause) => { paused() };
    };

    def paused() {
      (on: stop) =>  { init()    };
      (on: resume) => { running()  };
    };

    def start() {
      init();
    };
  };
};
```

Figure 8.7: Application contracts protocol implementation.

A protocol contract that implements this protocol by means of the object protocols shown in section 6.2.2 is shown in Figure 8.7. As can be seen, the states and the transitions of the diagram are in a one-to-one relationship with the transitions shown in the contract code.

Ambient Contracts UrbiFlock applications can register a callback object in order to be notified when other applications are discovered in the network. This object has to understand two methods `notifyApplicationJoined` and `notifyApplicationLeft` . These callback functions are invoked when applications join or leave the proximity of the user. Figure 8.2 shows that the developers of the UrbiFlock framework already specified some requirements over such callback objects by using code comments. The direct translation of this contract from the code comments into AmbientTalk/C, is shown in Figure 8.8 (lines 1 – 8). For the `notifyApplicationJoined` method (line 3), it specifies that the first argument is a far-reference obeying the `flockr` contract, the second argument is a locally accessible object obeying the `profile` contract and finally the last argument is a far-reference to an object implementing the remote interface of a custom application. The contract over the `notifyApplicationLeft` method is identical.

Note that the return values of both `notifyApplicationJoined` and `notifyApplicationLeft` should be the value **`nil`** indicated by the `nothing` contract. Remember that AmbientTalk does not have an explicit return statement and any value that happens to be the the last statement of the function or method is returned to the caller. Therefore there is always a return value and another correct (weaker) contract could have been the `any` contract. The use of the `nothing` contract thus makes it explicit that the application listener callback functions should not accidentally leak any return values.

What the code comments did not specify is how the profile itself should behave. Figure 8.8 shows an object contract for the profile object (lines 10 – 13). It specifies that a profile contract should have at least two methods, `setField` and `fieldMatches`. The profile contract validates an important property of the fields of a profile, namely that all the elements must be passed by copy. This is validated by the **`isolate`** contract

```
 1 def applicationListener := ObjectContract: {
 2   def notifyApplicationJoined(flockr, buddyProfile, app) {
 3     farRef(flockr) * IsolateRef(profile) * far -> nothing;
 4   };
 5   def notifyApplicationLeft(flockr, buddyProfile, app) {
 6     farRef(flockr) * IsolateRef(profile) * far -> nothing;
 7   };
 8 };
 9
10 def profile := ObjectContract: {
11   def setField(propName, propVal) { symbol * isolate -> any;    };
12   def fieldMatches(field, value)  { symbol * isolate -> boolean;  };
13 };
```

Figure 8.8: Contracts defined over the application listener.

as shown on lines 11 and 12.

Object Contracts. Figure 8.9 first shows two helper contracts (line 1 – 10). The first helper contract, `cancelObject` is used in order to cancel a subscription (see section 2.3.3). It simply specifies that objects over which it is defined need to support a `cancel` method. The second helper contract, `remoteAppInterface` is used in order to specify how to retrieve the flockr and its profile by remote applications. This is done by sending an asynchronous message `getOwnerAndProfile`. This method returns an array with two objects: the flockr and its profile. Note that both contracts are object contracts. The profile is always passed by copy to remote peers, in the contract this behavior is validated by the `IsolateRef` contract (line 8).

Figure 8.9 also shows how the application contract can be constructed with the two helper contracts shown before (line 12 – 22). The contracts defined over the application object make sure that the name is a string (line 13) and that the owner is a flockr (line 14). The contract defined over the **`export`** method of the application object specifies that an object has to be exported. This is validated with an ensure contract (line 15). The rest of the contracts defined over the application object define contracts about retrieving the remote interface and registering application listeners.

Module Contract. Finally, the previously defined contracts are applied to the constructor function that is exposed to other modules. The contract defined to the `makeApplication` function (line 26) validates that the arguments are a string and a flockr. It also validates that the return value satisfies the application protocol and finally that the application obeys the `application` contract shown before.

8.3.1 Experience

The use of AmbientTalk/C to specify the behavior of the application module has aided both in the development of the contract system itself and to find bugs in the development of the UrbiFlock framework.

- Homogeneous table contracts have always been part of the contract framework by means of the contract constructor `arrayOf`. In the application module it became clear that tables are often used to return multiple values from a function. Hence, in many cases these tables contain *heterogeneous* values. For example, in the application module the `getOwnerAndProfile` function returns two values:

```
def cancelObject := ObjectContract: {
  def cancel() { void -> nothing ; }
};

def remoteAppInterface := ObjectContract: {
  def name() { void -> string };
  def getOwnerAndProfile() {
    void -> arrayOf([Ref(flockr), (IsolateRef(profile)) ]);
  }
}

def application := ObjectContract: {
    def name()  { void -> string };
    def owner() { void -> Ref(flockr) };
    def export(asType) { typeTag -ensure_c(&export:as:)-> exportPublication; };
    def remoteApplicationInterface() {
      void -> Ref(remoteAppInterface);
    };
    def registerApplicationListener(type, l) {
      typeTag * applicationListener -> Ref(cancelObject);
    };
};

// the module object of this file
def Interface := object: {
  def makeApplication := provide: &makeApplication withContract: string * Flockr ->
                   (ensure_c(AppliactionProtocol) + Ref(application));
  ...
}
```

Figure 8.9: Contracts defined over the application module.

a remote interface and a profile. As this pattern is often used by AmbientTalk programmers, the primitive contract constructor `arrayOf` was extended to also support heterogeneous tables. The `arrayOf` contract constructor when given a table of contracts T_c creates a contract that verifies that a table of values T_v pairwise obey the contracts in T_c.

- When writing generic code, it is sometimes the case that the contract defined can be very general. These contracts then need to be refined by the programmer when possible. For example, the interface of an application is specific for each custom application. As we show in the next section, in order to export (contracted) applications in the network the programmer needs to manually overwrite the contract of the parent object. It would be interesting to find a more general mechanism to overwrite such generic contracts by the child object.

8.4 Contracts over the Application IR8U Module

As explained in section 8.1, UrbiFlock is a toolkit for the rapid development of pervasive social network applications that run on mobile ad hoc networks. When making use of UrbiFlock framework, programmers do no longer have to be concerned about discovery of services or failures in the network layer. Instead they can work with different notions of proximity that make sense for pervasive social networking applications. To plug additional applications into the framework that make use of its offered infrastructure, UrbiFlock developers only have to implement a small set of methods.

In this section we first explain the implementation of a simple UrbiFlock application called *I rate you* (IR8U) and then we show how ambient contracts are used in order

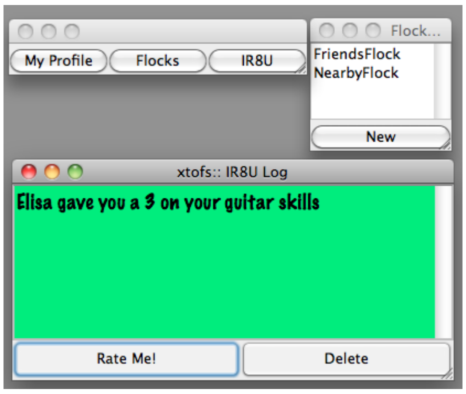


Figure 8.10: Screenshot of IR8U application in Urbiflock

to validate the correct execution of the application.

8.4.1 IR8U Overview

This IR8U application allows users to ask proximate users to rate them on a certain subject. Figure 8.10 shows a screenshot of the IR8U application in Urbiflock. It depicts the Urbiflock screen launcher for a flockr called “xtofs”, which consists of buttons to access its profile, its defined flocks and its installed applications (only IR8U in this case). The figure also shows the flock viewer (launched when the user clicks the flocks button) with the two predefined flocks (corresponding to the `isNearby` and `isFriend` proximities). The bottom part of the figure shows the GUI for IR8U which consists of a list of pending ratings. In this example, the flockr has an ongoing rate about his guitar skills (with one reply from a flockr called Elisa). In IR8U, other users can rate the subjects by giving a rating between 0 and 5 stars.

The first step in the creation of the IR8U application is to extend the prototypical UrbiFlock application with custom infrastructure as shown in the code snippet below. We define the necessary data structures to keep track of who is connected in the proximity and who rated certain subjects. A vector `connectedRaters` stores the flockrs who are connected in the proximity while a hashmap `ratingSubjects` stores the subjects (as keys) and their ratings (as values). Each rating itself consists of a pair of a far reference referring to the flockr who rated the subject and an integer between 0 and 5 representing the flockr’s rating. Notice that in order to identify applications in Urbiflock, every application is associated with a type tag. Therefore, we create one for IR8U with the same name as the application itself. Last we define a variable to contain a reference to the GUI. This variable is not initialized here yet.

```
def localInterface := extend: makeApplication("IR8U", aFlockr) with: {
 def connectedRaters := Vector.new();
 def ratingSubjects := HashMap.new();
 deftype IR8U;
 def ui;
 ...
}
```

The next step is to implement the two mandatory methods `start` and `stop` which are called by the framework when the user starts and stops the application (see section 8.3). The main purpose of these functions is to initialize and clean up the application's listeners and internal data structures. The code snippet below shows the original `start` method for the IR8U application.

```
1  def start() {
2   super^start();
3   ui := jlobby.at.urbiflock.ui.ir8u.IR8U.new(self);
4   self.export(IR8U);
5   subscription := self.registerApplicationListener(IR8U,
6    object:{
7     def notifyApplicationJoined(flockr, profile, ir8uApp){
8      connectedRaters.add(ir8uApp);
9     };
10    def notifyApplicationLeft(flockr, profile, ir8uApp){
11     connectedRaters.remove(ir8uApp);
12    };
13   });
14 };
```

First the GUI is initialized (line 3) after which the application is exported to the network by calling the **`export`** method with its type tag (line 4). The framework takes care of exporting the application on the network and notifying listeners for this application. Finally, a listener is registered that updates the vector when connected IR8U users enter or leave the proximity (lines 5 – 13). This is done by calling the `registerApplicationListener` method with the type tag `IR8U` and a listener object. This listener object implements two methods `notifyApplicationJoined` and `notifyApplicationLeft` which are called when another application in the proximity is discovered or decides to leave, respectively. Both of these methods are called with a reference to the flockr in the proximity, a copy of his profile, and a reference to the remote interface of the IR8U application of the remote flockr. The code snippet below shows the implementation of the `stop` method.

```
1 def stop(){
2   super^stop();
3   if: (subscription != nil) then: {
4     subscription.cancel();
5     subscription := nil;
6     connectedRaters := nil;
7   };
8 };
```

The `stop` method is responsible for cleaning up the IR8U application. It first issues a super-send to invoke the default cleanup code defined in the prototypical UrbiFlock application (which takes the application offline by unexporting it). Application listeners are then removed (by invoking `subscription.cancel()`) and its data structures are set to **`nil`** such that they can be garbage-collected (lines 4 – 6).

Now that we have explained how to start and stop the application as well as the data structures that it uses, we can explain the implementation of the basic functionality of IR8U. A user can ask all proximate users to rate a certain subject as implemented in the `askRatingFor` method. When this method is called it first creates a new subject and adds it to the `ratingSubjects` hashmap. Next, it sends an asynchronous message `rateMe` to all connected raters asking them to give a rating on the subject. A flockr can give this rating by calling the `rateFlockr` method. This method sends the asynchronous message `rate` to the remote application. Note that `rateMe` and `rate` must be send asynchronously to the connected `ir8uapp` applications since they are remote objects.

```
def askRatingFor(subject) {
 ratingSubjects.put(subject, []);
 connectedRaters.each: { |ir8uapp|
  ir8uapp<-rateMe(remoteInterface, aFlockr.getProfile(), subject);
 };
};
```

```
def rateFlockr(ir8uApp, subject, rating) {
 ir8uApp<-rate(aFlockr, aFlockr.getProfile().username, subject, rating);
};
```

All previously shown method definitions are contained in the local interface as these methods and data structures should not be accessible from remote devices. Now we describe the remote interface by which remote devices can interact with the applications in the proximity. As mentioned earlier, this remote interface defines the methods `rate` and `rateMe`. The `rate` method is invoked when a remote Flockr issues a rating on this flockr's subject. It simply records the rating given by the Flockr and updates the GUI. The `rateMe` method is called when a remote Flockr asks this Flockr to rate him on a subject. The method sends a message to the GUI to ask the user for a rating.

```
def remoteInterface := extend: localInterface.remoteApplicationInterface with: {
 def rate(ratingFlockr, ratingFlockrName, subject, rating) {
  ratingSubjects.put(subject, prevRatings + [[ratingFlockr, rating]]);
  ui.updateRating(ratingFlockrName, subject, rating);
 }
 def rateMe(ir8uApp, profileToRate, subject) {
  ui.askToRate(ir8uApp, profileToRate, subject);
 };
};
```

8.4.2 Contracts

The IR8U application has many connections to other modules in the UrbiFlock framework. An overview of all the contractual bindings between the IR8U application and the rest of the framework is shown in Figure 8.11. The UrbiFlock module creates the application and adds it to the list of applications. From this module the application is started and stopped. Further the IR8U application crosses the language boundaries by passing a reference to itself to the graphical user interface which is written in Java. And finally IR8U applications communicate with other instances of the same application by their remote interface. At all these module interactions there are opportunities to validate the operation of the IR8U application against the contracts as we show in the remainder of this section.

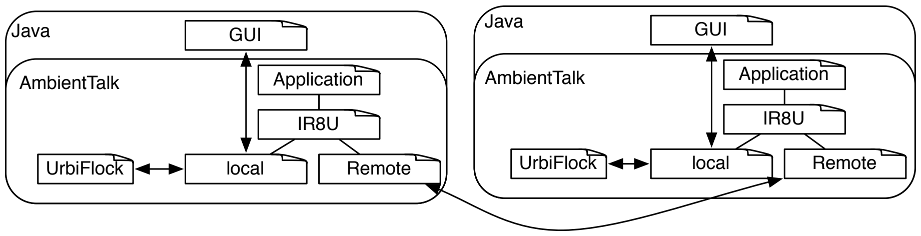


Figure 8.11: Overview of contractual bindings of the IR8U application.

Contractual Bindings with remote Applications The IR8U application exposes itself to other devices in the network proximity. This is done by applying the **`export`** method of the application that it extends. When the inheriting application — the IR8U application in this case — wants to define a contract over the exported module it can do this by setting the field `contract`. The code for exporting an application over the network as defined in the application module is shown in Figure 8.12. As can be seen in the code, the application module exports the module with a contract when it is not **nil**. Note that this object contract is exported with the Ref contract constructor as seen in chapter 6.

```
def export(asType) {
  if: (self.contract != nil ) then: {
    exportPublication := export:  self.remoteApplicationInterface
              as: asType withContract: Ref(self.contract);
  } else: {
    exportPublication := export: self.remoteApplicationInterface  as: asType;
  };
};
```

Figure 8.12: Exporting an application.

Once exported, other devices that discover the IR8U application can send messages to the newly discovered application. The contract used when exporting a IR8U application (exportIR8U) is shown in Figure 8.13. It defines a contract over the rateMe method of the application. The first argument should be a far reference to a remote IR8U application. Note that from the perspective of the exporting IR8U application the receiving argument app is a far reference. Therefore, the contract constructor FarRef (shown in chapter 7) is used to create this object contract. The contract also validates that the name of the remote IR8U application equals "IR8U" with a flat far reference contract. This contract is combined with the contract operator + which applies both contracts.

The second argument of the rateMe method is a profile object which is passed by copy. Finally, the subject is a string. Note again the use of the nothing contract to express that the rateMe method does not returns the value **nil**. In this case, the use of the nothing contract is more important as it prevents accidental leaking of objects across the network.

A second contract (lines 9-13) describes how a far reference to a remote IR8U application should behave. Note that the arguments that are being sent to this remote reference have to be seen from the perspective of the local IR8U application. Therefore the first argument to the rate message is a *local* object flockr. The arguments ratingFlockrName and subject are both strings and the rating is a number between 0 and 5. This last contract is created with the interval function as shown from lines 15 till 17 of Figure 8.13. This interval function returns a flat contract that given a number x verifies that the number is between the min and maximum values.

Contractual Bindings on the local IR8U application. The contract defined on the application module seen in section 8.3 makes sure that the start and stop methods are applied in the correct order. However, the IR8U application must not forget to make the mandatory super calls to the application it inherits. By making use of computational contracts it is possible to ensure these mandatory super calls. The computational contract which can validate such super calls is shown in Figure 8.14. Such mandatory super-calls are also interesting in other cases aside from the UrbiFlock framework and a new computational contract constructor function superCall was added to the set of primitive contract constructor functions. This contract constructor function makes use of reflection in order to grab the method of the super object of the contracted object (line 2). Once this method has been determined an new ensure contract is created. Two

```
def isIR8U := Assert:{|r| r<-getName() == "IR8U"; };

def exportIR8U := ObjectContract: {
  def rateMe(app, profileToRate, subject) {
    (FarRef(remoteIR8U)+isIR8U) * IsolateRef(profile) * string -> nothing;
  };
};

def remoteIR8U := ObjectContract: {
  def rate(ratingFlockr, ratingFlockrName, subject, rating) {
    Ref(flockr) * string * string * interval(0,5) -> nothing;
  };
};

def interval(min,max) {
  flat({|x| (min <= x).and:{ max >= x } });
};
```

Figure 8.13: Ambient contracts over the remote application interface of IR8U.

mandatory super calls for the `start` and the `stop` method of the IR8U application are shown at the bottom of the code example (lines 7-8).

```
1  def superCall(method) {
2    occ: { |o| ensure_c( (reflect: o.super).getMethod(method) ) };
3  };
4
5  ObjectContract: {
6    ...
7    def stop()  { void -(superCall(‘stop))-> nothing;  }
8    def start() { void -(superCall(‘start))-> nothing; }
9    ...
10 };
```

Figure 8.14: Computational contracts for mandatory super calls.

Contractual Bindings with the GUI. The IR8U application creates the graphical user interface by directly creating a Java class for a window. From within the GUI, two contracted methods can be applied to the IR8U application (as shown in figure 8.15): `rateFlockr` and `askRating`. The first contract (lines 3-5) specifies that the application should be a far-reference to an IR8U application, the second a string and the rating a number between 0 and 5. The `askRating` contract (lines 7-9) is a bit more interesting. It determines that the number of outgoing messages should equal the amount of connected raters. This is done by the message contract defined over the execution of the `askRating` method.

8.4.3 Experience

- In the definition of the contracts for the IR8U application it became apparent that we needed a contract dedicated to checking whether or not a value falls within a certain interval. While it is not a difficult contract to define oneself, it is a general abstraction and it was decided to add it to the set of standard contracts in AmbientTalk/C.
- For some reason, the comments in the original UrbiFlock framework specified that ratings should be between 0 and 4 while the development team clearly de-

```
def GUIContract := ObjectContract: {
  ...
  def rateFlockr(app, subject, rating) {
    farRef(remoteIR8U) * string * interval(0,5) -> any;
  };

  def askRating(subject) {
    string -(occ:{|o| atMost: { o.connectedRaters } outgoingMessagesFor: 'rateMe })-> nothing;
  };
  ...
}
```

Figure 8.15: Message contract for asking ratings.

cided on having values between 0 and 5. While this is not a severe error, it shows that the use of textual documentation is prone to errors. Further, there was no error checking code installed to make sure that the input from the user was indeed a value between 0 and 5.

- The use of computational contracts for enforcing super calls for the `start` and `stop` functions of the application was extremely useful. Such behavior can be enforced in languages like Java by making the `stop` method in the superclass final, and creating a protected abstract method base_stop which is invoked from the stop method. However, in languages like AmbientTalk which do not have abstract or final methods computational contracts certainly provide a solution [2].

- When starting to use AmbientTalk/C it might be difficult for the programmer to know whether a certain value should have a local object or a far-reference contract. The reason why this might be confusing from time to time is that there are always two sides for a remote interaction. The correct perspective in order to determine that something should be an object contract created with `Ref` or `FarRef` is to take into account *where* the contracted object is hosted. If the object is hosted by the actor applying the contract `Ref` should be used; otherwise, the object is hosted by a remote actor and `FarRef` should be used.

- We found confusing documentation in the IR8U application for the `rate` method:
  // rate(ratingFlockr : farref<Flockr>, subject : Text, rating: int 0-4)
  First the comments imply that the contract should be a far reference over the flockr. Because the contract is applied over a local object this should be an object contract created with `Ref`. When applying our contracts at first we followed the documentation and suddenly got a lot of contract violations. We found out quickly that the previous documentation was in fact wrong. Furthermore, the documentation forgot to specify that the name of the flockr is sent.

8.5 Conclusion

In this chapter, we applied ambient contracts to UrbiFlock, framework for the rapid development of social ambient applications. We applied flat far reference contracts, local object contracts (with `Ref`) and remote object contracts (with `FarRef`). Furthermore, we have applied computational contracts in order to monitor that object protocols are

[2] A similar effect as the Java solution can be encoded by making use of traits but that would have required a substantial redesign of the UrbiFlock framework.

respected. We deduce three main conclusion from applying AmbientTalk/C for the development of UrbiFlock.

- First, we have found that many of the distributed specifications could be easily encoded with the developed contracts. A remarkable example of this can be seen by the direct translation of the documentation in comments into an actual contract.

- Second, it can be concluded that when developing interesting contracts which we did not anticipate will emerge. The proof of the pudding is in the eating. One example of this can be found in the definition of heterogeneous array contracts (with `arrayOf`), it was only when we started applying our contracts in a bigger application that the generalization of such a contract became useful. While the definition of this contract is not too complex for an expert contract programmer we do not expect end users of the contract system to write contracts in that manner. Furthermore, applying contracts to UrbiFlock framework has been an extremely useful tool to understand the flow of objects within the ambient application. For example, we noticed that in the IR8U application a remote object is passed from the AmbientTalk module up to the Java GUI and back to the AmbientTalk module without ever being used.

- A final conclusion is that it is extremely important for the contract programmer to understand where the object over which the contract is hosted. Depending on *where* the object is hosted, the perspective of remote or local reference changes. This also has an impact on the specification of the contracts themselves. When developing ambient application keeping this perspective is always important. With ambient contracts we make this perspective explicit.

Chapter 9

Conclusion

In this final chapter, we reflect on the challenges for ambient contracts stated in the introduction and highlight how the AmbientTalk/C framework address those challenges. We discuss the rough edges of AmbientTalk/C and give an overview of future research directions.

9.1 Summary and Contributions

Weiser envisioned a future where computers are no longer on the foreground but instead embedded in everyday objects [Wei91]. Today, we can say that technological advances have transformed the bulky machine that a computer used to be into a small piece of art which fits in your pocket. Ever since Weiser formulated his vision — now almost two decades ago — technological advances in software and hardware have rapidly improved. An everyday mobile phone today is much more powerful than a desktop computer 10 years ago and consumes only a fraction of the energy.

The development of software for this new technology has been confronted with three main challenges. The first challenge was how to connect all those devices over a wireless network. The second challenge was the question of how to make devices adapt themselves to their environment. And finally, the third unsolved challenge is how to make bigger and smarter ambient applications. Ambient contracts are a research contribution in context of the third generation of research challenges for ubiquitous computing. It is motivated by the observation that contract systems have not focused on the particular language abstractions that have been proposed for the development of ambient applications. This includes the use of futures, first-class remote references and event-driven language constructs. As a result, it was unknown how a contract system to validate applications for mobile ad hoc networks should look like. In this dissertation, we have formulated a first answer to this question under the form of ambient contracts. We now revisit the contributions of ambient contracts outlined in this dissertation.

Survey of existing contract systems. Our first contribution is an extensive survey of related contract systems presented in chapter 3. We started from the fact that the landscape of contract systems can be roughly divided into four categories: static contracts, behavioral contracts, synchronization contracts, and contracts for quality of service. We first concluded that static systems are ill fitted for the hardware properties of mobile ad hoc networks as they would either require the programmer to use a lot of annotations and/or require a lot of computation. Secondly, we found that synchronization contracts are less interesting for the development of an ambient contract system because ambient-oriented languages already exclude many synchronization violations by design. Finally, we also conclude that quality of service as understood in the context of mobile ad hoc networks is highly related to context awareness. The focus in this dissertation is on contract systems that monitor the flow of values between the different (distributed) modules of the system.

From the survey it became clear that most contract systems that adhere to certain properties needed for the development of ambient applications fall in the category of behavioral contracts. Such contracts specify pre and post-conditions which are validated at runtime. The work which turned out to be the most interesting for the development of an ambient contract system is the work inspired by Findler and Felleisen [FF02]. The work described in their papers form the foundations on top of which ambient contracts have been built. Before the start of our research it was unclear how this contract system could be combined with ambient-oriented language constructs. From the survey of related work in chapter 3 we concluded that previous contract systems are not *expressive* enough to deal with *complex higher-order values* in the presence of highly *volatile connections*.

The Communicating Event Loop Calculus. The second contribution outlined in this dissertation is the communicating event loop calculus. This model gives a formal overview of the event-loop concurrency model in which we have formulated ambient contracts. This event-loop concurrency model forms the basis of the concurrency and distribution model that adheres to the AmOP criteria presented in chapter 2. We believe this is the first formal account of an actor language that is based on communicating event loops. Hence, it is not only the formal foundation for AmbientTalk but also for other languages based on the event loop model like for example E [MTS05]. To a lesser extend it also forms the formal foundation for the many event loop frameworks found in for example OS X Core Foundation run loops, GLib event loops, Ruby's Event Machine, Python's Twisted etc.

A novelty of our semantics is the operational description of *non-blocking futures*. A future is a first-class value that acts as the placeholder for a value that is asynchronously awaited. While the future is unresolved, any messages sent to the future are queued. When the future becomes resolved, all messages in the queue are forwarded to the resolved value and the queue is emptied. In the semantics, we give a formal account of future pipelining and show how futures can be used in a distributed environment. Our semantics also includes the primitives necessary for service discovery, i.e. the ability for objects in different actors to discover one another by means of a publish/subscribe-style mechanism. Finally, our semantics shows how to encode robust time-decoupled message transmission. The semantics of the communication event loop calculus has been validated by using PLT-Redex [FFF09] tool.

Capturing Ubiquitous Behavioral Properties. The third contribution is the design and implementation of a novel contract framework called AmbientTalk/C. This framework combines several types of contracts amongst which flat and higher-order contracts. These contracts can be define over data values, functions, objects and future values and even work in the presence of event handlers. In AmbientTalk/C, contracts can be defined both over objects hosted by the actor or objects hosted by remote actors.

In order to capture the internal behavior of a certain entity, we proposed a novel contract type called *computational contracts*. A computational contract validates the execution of a contracted entity. With computational contracts the developer can define a function contract that verifies a single event or a sequence of events during the execution of the contracted function. We have defined computational contracts that can be used in combination with object-oriented language concepts and event-loop language constructs. Computational contracts are built upon aspect-oriented language techniques that we precisely describe by presenting an operational semantics.

When an actor receives a message, it does some computation and as a response can sent new messages in its turn. The AmbientTalk/C framework has support to monitor the trace of outgoing messages that are a consequence of processing an incoming message. Analogous to how a single method invocation or a sequence of method invocations can be monitored by a computational contract, a message contract monitors the trace of outgoing messages.

Scoping of Behavioral Contracts The fourth contribution deals with the scope of behavioral contracts. In event loop concurrency languages such as AmbientTalk and E [MTS05], the use of callbacks is omnipresent. Every time a callback is registered there is an entry point created in the code that can be triggered in a later turn of the actor. The validation of computational contracts in combination with an event-driven programming model poses questions about the scope of behavioral contracts. In most cases the programmer expects computational contracts to be active within the body of the registered callbacks. Therefore, the contract system needs to be able to capture violations that transcend a single turns of an actor.

In order to achieve such a scoping mechanism the underlying aspect-oriented programming language had to be extended with a novel mechanism called *first-class aspect environments*. First-class aspect environments allow the programmer to capture the currently active aspects. It is possible to re-apply these aspects at a later moment in time. As contracts are encoded with aspects, allowing the programmer to arbitrary alter the deployed aspects would be dangerous. Overwriting the aspect environment would give the programmer the power to remove or loosen contracts. In our semantics we disallow such an operation.

Complex Values and Blame Assignment. In an expressive language such as AmbientTalk, distributed programmers are able to exchange objects, functions, futures, and remote references between distributed communication partners without having to worry about the serialization or about the way messages are actually distributed by the underlying runtime. The AmbientTalk/C framework is able to monitor such higher-order functions, futures, remote references etc. The definition of our contract system can even deal with objects hosted by remote actors. While we defined contracts over such complex values we have found that some of these values require special attention.

First, we showed how a contract system can deal with the use of non-blocking futures. We found that futures conceptually invert the responsibility of the server and the client leading to the notion of *blame inversion*. The main reason is that the contracted value (the future) is resolved from outside the module that has a contract defined over it. Normally, contracted (higher-order) values are applied from inside the module which have a contract over them. For future values this is the other way around. In chapter 6 we described a technique — called blame inversion — to validate future contracts which is closely related to future pipelining. In our contract system a future contract forms one of the pieces of a future pipeline. When the resolved value obeys the contract it simply forwards this value in the pipeline. Otherwise blame is assigned.

Second, we showed how to deal with assertions over far references in chapter 7. In order to account for such assertions we introduced the notion of *flat far reference* contracts. A flat far reference contract validates a predicate over an object hosted by *another* actor. Validating flat reference contracts requires asynchronous communication with the remote object over which it is defined. This is in contrast to flat contracts in a local synchronous model where predicates can be validated immediately. We showed how flat far references can be validated and how to assign blame in case of a violation.

Third, we showed how a contract system can deal with higher-order far reference contracts. A higher-order far reference contract is similar to a higher-order object contract but specifies properties over a *remote* object. When an asynchronous message is sent to a remote object the result is a future. A higher-order far reference contract monitors all outgoing messages to the contracted far reference and monitors the resulting future values as the result of sending messages. Note that the values that are being sent in these messages can again be complex first-class values such as objects or futures.

Validation Our final contribution is the validation of the AmbientTalk/C contract system by deploying it in the context of an existing framework called UrbiFlock (shown in chapter 8). From this use case, we found that many of its informal specifications could be easily encoded with the developed contracts. We found that the specification of ambient contracts are similar to writing extensive documentation in comments. In certain cases we could almost make a one to one mapping from code comments onto an actual contract. The use of contracts during the development of the UrbiFlock framework has been an extremely useful tool to understand the flow of objects within the framework. We also found out that it is extremely important to understand *where* the contract is validated. Depending on where the contract is validated the perspective of remote or local reference changes i.e. whether the concepts of chapter 6 or chapter 7 apply.

9.2 Shortcomings and Future Work

After mentioning our achievements, it is time to reflect on what can be done next. We present future research opportunities and show where ambient contracts can be improved.

Ownership. In recently published work Dimoulas [DF11] proposes a correctness verification for contracts based on *ownership*. It would be interesting to see how these techniques can be adopted in an ambient contract system. It could, for example, warn the programmer of values that are not subject to contract verification — and thus dangerous objects — received from remote parties.

Interaction Protocols. Ambient contracts allow the programmer to monitor the outgoing messages and even implement a protocol in order to see which messages are being sent. We have not investigated how the programmer can make use of these protocols in order to monitor the traffic between two or more objects in a distributed setting. Currently, it would be up to the programmer to manually encode a contract that transitions a state machine when a message is received and transitions that state machine when a message is sent. There is a large body of research that deals with the specification and verification of complex interaction protocols. As described in chapter 3, one promising technique deals with the interactions between distributed processes. Such interactions can even be captured in the type signature of a communication channel. However, these communication channels are difficult to reconcile with object-oriented languages. Nevertheless, there has been a significant amount of work that has tried to combine session types and object-orientation [DCMYD06, DCGDY07, DDCC07, CCDC$^+$09, GVR$^+$10]. However, none of them have managed to completely eliminate the concept of the communication channel. It would be interesting to combine session types and object-oriented model in a unified framework.

Performance. In the definition of our contract system we have not focused on efficiency. A large part of the performance overhead of the ambient contract system is due to the inefficient implementation of the underlying aspect language. Fortunately, there has been a significant body of related work which has focused on the efficient implementation of aspect-oriented languages. For example, in the work of Masuhara et al. [MKD03] they make use of partial evaluation to remove unnecessary run-time checks of programs that make use of aspects. Hilsdale et al. [HH04] show that automatic advice weaving in AspectJ can be as efficient as a hand-crafted approach. This work can thus serve as a starting point for optimizing ambient contracts. However, our aspect language has support for first class aspect environments in order to be able to express contracts that span more than one turn of the actor. How these optimization techniques can be combined in order to support first-class aspect environments is future work.

Static, Dynamic and Hybrid Verification. In this thesis, we have argued for the dynamic validation of contracts. As shown in chapter 3, there is a large body of work which deals with the static validation of contracts. For a limited subset of flat contracts it would indeed be possible to validate them statically in a dynamic language. However, it is clear that purely static techniques today are not expressive enough. One avenue of future work could be to adopt hybrid validation techniques as proposed by Flanagan [Fla06].

Higher-Order Message Contracts. In the current state of our work we have provide the programmer with message contracts that allow the programer to intercept outgoing messages. The programmer can prohibit or ensure that certain messages are sent. Many times, however, these messages contain values such as functions and objects which can not be validated with a simple predicate. Therefore, one avenue of future work would be to provide the programmer with abstractions to apply (higher-order) contracts over the arguments of the outgoing messages. As the infrastructure to intercept and alter the arguments of the outgoing messages is already there (see section 7.7), we do not foresee any fundamental difficulty to implementation such a higher-order message contract system. However, it remains to be seen what the impact of such a contract system would have on the application and in which context such contract would really contribute.

9.3 Conclusion

Just like object-oriented techniques have gained popularity in the nineties, the use of ambient-oriented programming is steadily gaining ground in the world of every day software development. For example, the Android platform already has a model of activities that is akin towards an actor like concurrency model. Mobile ad hoc networking technology has provided us with a window on the future. It showed us a glimpse of what future ambient applications might bring. At the same time, it also has shown us the difficulties that future generation programmers will need to deal with. The prior research on ambient-oriented programming has formulated answers for the two fundamental research challenges of connectedness and awareness. Before mainstream programming languages adopt the software abstractions that have been developed in order to overcome the challenges of connectedness and awareness there are still questions to be answered. As put forward in the third generation challenges of ambient applications, ambient programmers are more and more in need for a methodology for ambient-oriented software construct. In the absence of such methodology it is unclear how programmers will be able to write large scale (smart) ambient applications.

In this dissertation, we brought the design by contract methodology to ambient-oriented software development. The ultimate goal of the AmbientTalk/C contract system is to ease the development of large scale ambient applications and to improve the reliability of ambient-oriented software systems. Where reliability should be understood as a combination of correctness and robustness or simply as minimizing the presence of bugs. It is a given that minimizing the amount of bugs in software is a very important property of all software systems. Therefore, it is surprising that few research has specifically focused on how to make the development of ambient-oriented systems more reliable. Our answer is the notion of ambient contracts and their implementation in a framework called AmbientTalk/C.

The main research problem that ambient contracts addresses, is how to build an *expressive* contract system that can deal with *complex first-class values* in the presence of highly *volatile connections*. Ambient contracts can deal with the validation of functions, objects, far references, futures and their callbacks. With this framework we give ambient programmers a set of tools that enable them to specify the requirements of their ambient applications as code contracts.

In the introduction chapter we have shown that there is a need for software tools that help the programmer to write more robust ambient applications. We stress now, that ultimately, our research is about bridging the gap that prevented programmers to apply the design by contract methodology in order to write more robust ambient applications. The contract system described in this dissertation is a state of the art *research* vehicle. However, we argue that when mainstream languages are ready to adopt ambient-oriented programming techniques, ambient contracts can be the basis of a solid design by contract methodology for the development of robust ambient applications.

Appendix A

Communicating Event Loop Calculus in PLT-Redex

This appendix complements the presentation of the communicating event loop calculus that describes the operational semantics of a key subset of the AmbientTalk programming language. The subset focuses on its support for asynchronous, event-driven programming. In this appendix, we present the *implementation* of the operational semantics in PLT-Redex. We first give some basic examples followed by more elaborate examples. Our implementation is freely available and can be run in DrRacket.

A.1 Basic Example of the PLT-Redex Semantics

The parser of the PLT-Redex semantics is not compatible with the syntax of AmbientTalk. In order to try out examples the syntax has to be translated into s-expressions which are understood by the PLT-Redex evaluator. As an example consider the following AmbientTalk code:

```
def x := 3 + 2;
x;
```

Figure A.1: Example AmbientTalk code snippet.

In the example shown in Figure A.2 the variable x is assigned to the value of adding the number 3 to the number 2. Then the variable x is returned. Following the syntax outlined in the semantics, this code snippet can be translated into s-expressions as follows.

```
(let (x  (+ 3 2))
    in
    x)
```

Figure A.2: Plt-Redex: AmbientTalk code snippet.

Currently, the PLT-Redex evaluator can only evaluate actor configurations. Therefore, the example has to be written as an actor configuration. The most simple actor configuration to test the example is a configuration consisting of one actor which is currently evaluating the let expression. This is written down as follows:

```
((actor id  () ()
       (let (x  (+ 3 2))
         in
     x)))
```

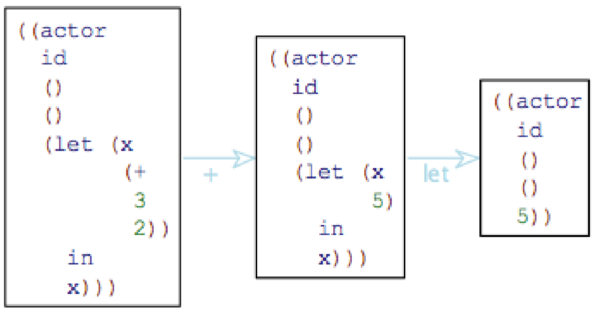
Figure A.3: Plt-Redex: Actor code snippet.

In this actor configuration there is one actor with the id id. Finally in order to show the reduction steps that the evaluator takes in order to reduce this term into a value the function traces can be used. This function takes a reduction relation and a term and shows the reduction graph. The reduction relation defined for AmbientTalk is called AmbientTalk-Red.

```
(traces AmbientTalk-Red
          (term
            ((actor id  () ()
             (let (x  (+ 3 2))
               in
                 x)))))
```

Figure A.4: Tracing the execution of an AmbientTalk program

Evaluating this code results in the following reduction graph.



```
((actor
  id
  ()
  ()
  (let (x
         (+
          3
          2))
   in
   x)))
```

```
((actor
  id
  ()
  ()
  (let (x
         5)
   in
   x)))
```

```
((actor
  id
  ()
  ()
  5))
```

Figure A.5: Plt-Redex Reduction graph.

A.2 Syntactic Sugar

Some expression of AmbientTalk are syntactic sugar for more basic expressions. For example, lambda expressions are transformed into objects with an apply method. Applying a lambda expression to a value is consequently transformed into invoking the apply method of the object. This syntax expansion is modeled in the PLT-Redex semantics by the function expand-syntax. As an example consider the application of a function r to the value p, i.e. r(p).

In order to use the syntactic sugar as defined in the paper make sure to apply the function expand-syntax to the evaluating term.

```
(expand-syntax (term ((actor id () () (r(p))))))
;evaluates to
((actor id () () ((r $ apply) p)))
```

Figure A.6: Syntactic sugar

A.3 Advanced Example of the PLT-Redex Semantics

In this section we show that the semantics models the creation of actors, isolates, asynchronous message sends with futures and registering code blocks that are triggered when a future resolves to a value. More concretely, first a new actor is created with a method `foo`, then a isolate is created and a future message is send to the newly created actor. When the actor receives this isolate it takes the `myfield` field of the isolate and returns the double. Finally, when the future resolves, the value 24 is added to the result and printed. The AmbientTalk code for this example is shown in Figure A.7:

```
import /.at.lang.futures;
  enableFutures(true);

  def newactor := actor: {
    def x := 2;
    def foo(isolate) {
      isolate.myfield + isolate.myfield;
    }
  };

  def newisolate := isolate: {
    def myfield := 9;
  };

  when: newactor<-foo(newisolate) becomes: { |x|
    system.println(x + 24);
  };
```

Figure A.7: AmbientTalk, sending an asynchronous message

Again we need to translate this example to s-expressions in order to be able to run the AmbientTalk PLT-Redex semantics as shown in Figure A.8.

```
(let (new-actor
       (actor
         (field x 2)
         (method foo x (+ ( x $ my-field) (x $ my-field)))))
  in
  (let (new-isolate (isolate (field my-field 9)))
    in
    (when (sendf new-actor foo new-isolate)  x (+ x 24))))
```

Figure A.8: Plt-Redex: sending an asynchronous message

Evaluating this expression with the trace functionality of PLT-Redex results in the following reduction graph. Note that after sending the asynchronous message the reduction graph splits up. This exemplifies the non-deterministic nature of AmbientTalk when more than one actor is active at the same time. As our example involves only two actors and one message all these interleaving eventually merge together into a single solution. This can be seen in the graph as all solutions merge together at the end. [1]

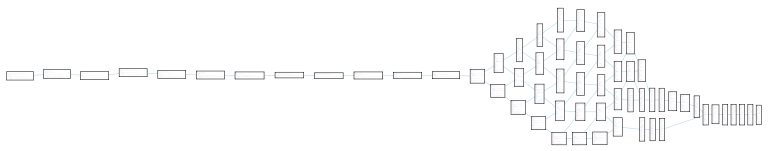


Figure A.9: Plt-Redex Reduction graph of an asynchronous message.

A.4 Reduction Rules

The reduction rules as implemented in the PLT-Redex semantics are shown in figure A.10. While we wanted to keep them as close as possible to the reduction rules shown in the dissertation they are currently differing in the following aspects.

- The PLT-Redex semantics of AmbientTalk only support single argument methods and lambda's.
- Initialization of the self variable with lambda expression is currently wrong.
- We currently have two rules for sending asynchronous messages, this is a technical detail and should be resolved.
- Differentiating between a future message send and a normal asynchronous message send is currently clumsy, we plan to write a meta-function to make this more elegant.
- We make use of the function variable-not-in, it would be more elegant to make use of the fresh function.

[1] In the digital version of this graph, the code in the boxes and the labels on the arrows can be made clear by zooming in.

```
                        K[(/ number_1 0)] ⟶ (error /)                                                        [error/]
            K[(+ number_1 number_2)] ⟶ K[(+ number_1  number_2)]                                      [+]
            K[(* number_1 number_2)] ⟶ K[(* number_1  number_2)]                                      [*]
            K[(/ number_1 number_2)] ⟶ K[(/ number_1  number_2)]                                      [/]
            K[(- number_1 number_2)] ⟶ K[(- number_1  number_2)]                                      [-]
                  K[(seq any_1 any_2)] ⟶ K[(let (x_1 any_1) in any_2)]                                 [syntax-expansion-seq]
                                          where x_1 = (variable-not-in e_2  some-var)
                 K[(let (x_1 v_1) in e_1)] ⟶ K[subst[[(x_1 v_1 e_1)]]]                                 [let]
 K[(actor id objects q E[(object f ... m ...)])] ⟶ K[(actor id U[[objects, (new-object)]] q E[field-assigners[[(f ...), reference, reference]]])]   [new-object]
  where io_new = (variable-not-in                  , reference = (ref id io_new), new-object = (object io_new obj initialise-fields[[(f ...), ()]]  m ...)
                 (actor id objects q E[(object f ... m ...)])
                 id)
 K[(actor id objects q E[(isolate f ... m ...)])] ⟶ K[(actor id U[[objects, (new-object)]] q E[field-assigners[[(f ...), reference, reference]]])]   [new-isolate]
  where io_new = (variable-not-in                  , reference = (ref id io_new), new-object = (object io_new iso initialise-fields[[(f ...), ()]]  m ...)
                 (actor id objects q E[(isolate f ... m ...)])
                 id)
K[(actor id_a                                                              ⟶ K[(actor id_a                                    [invoke]
      (obj-entity_0 ...                                                           (obj-entity_0 ...
       (object id_o any f ... (method x_mn0 x_m0 e_0) ... (method x_mn x_m e) (method x_mnn x_any e_n) ...)     (object id_o any f ... (method x_mn0 x_m0 e_0) ... (method x_mn x_m e) (method x_mnn x_any e_n) ...)
       obj-entity_n ...)                                                           obj-entity_n ...)
      q                                                                           q
      E[(((ref id_a id_o) $ x_mn)  v)])]                                          E[subst[[(this (ref id_a id_o) subst[[(x_m v e)]])]]])]
K[(actor id_1                                                              ⟶ K[(actor id_1                                    [field-access]
    (obj-entity_0 ... (object id_2 any (field x_0 v_0) ... (field x_1 v_1) (field x_n v_n) ... m ...) obj-entity_n ...)     (obj-entity_0 ... (object id_2 any (field x_0 v_0) ... (field x_1 v_1) (field x_n v_n) ... m ...) obj-entity_n ...)
    q                                                                             q
    E[((ref id_1 id_2) $ x_1)])]                                                  E[v_1])]
K[(actor id_1                                                              ⟶ K[(actor id_1                                    [field-update]
      (obj-entity_0 ... (object id_2 t (field x_0 v_0) ... (field x_1 v_1) (field x_n v_n) ... m ...) obj-entity_n ...)     (obj-entity_0 ... (object id_2 t (field x_0 v_0) ... (field x_1 v_2) (field x_n v_n) ... m ...) obj-entity_n ...)
      q                                                                           q
      E[(set! ((ref id_1 id_2) $ x_1) v_2)])]                                     E[v_2])]
        K[(actor id_a objects q E[(let (x_f x_r) future in e)])] ⟶ K[(actor id_a                              [make-future]
                                                                      U[[U[[objects, ((fut i_f () null))]], ((res i_r i_f))]]
                                                                      q
                                                                      E[subst[[(x_f (ref id_a i_f) subst[[(x_r (ref id_a i_r) e)]])]]])]
                                                                   where i_f, i_r fresh
                  K[(actor id_a                                    ⟶ K[(actor id_a                                    [local-asynchronous-send]
                           objects                                      objects
                           q                                            U[[((message (ref id_a id_object) x_message (v ...))), q]]
                           E[(send (ref id_a id_object) x_message (v ...))])]     E[null])]
          K[(actor id_a                                            ⟶ K[(actor id_a                                    [process-message]
                (obj-entity_0 ... (object id_o t f ... m ...) obj-entity_n ...)     (obj-entity_0 ... (object id_o t f ... m ...) obj-entity_n ...)
                (msg_actor ... (message (ref id_a id_o) x_method (v_2 ...)))       (msg_actor ...)
                 v_1)]                                                      process[[(ref id_a id_o), x_method, (v_2 ...)]])]
            K[(actor id_a                                          ⟶ K[(actor id_a                                    [process-message-resolve]
                  (obj-entity_0 ... (res id_o id_f) obj-entity_n ...)             (obj-entity_0 ... (res id_o id_f) obj-entity_n ...)
                  (msg_actor ... (message (ref id_a id_o) x_method (v_2 ...)))     (msg_actor ...)
                   v_1)]                                                    process[[(ref id_a id_o), x_method, (v_2 ...)]])]
         K[(actor id_a                                             ⟶ K[(actor id_a                                    [process-message-to-future]
               (obj-entity_0 ... (fut id_fut (msg_fut ...) v_fut) obj-entity_n ...)     (obj-entity_0 ... (fut id_fut U[[(msg_fut ...),  message]] v_fut) obj-entity_n ...)
               (msg_actor ...    (message (ref id_a id_fut) x_method (v_args ...)))    (msg_actor ...)
                v_actor)]                                                   e_r)]
                                 where message = (first  store[[x_method, (v_args ...), v_fut]]), e_r = (second store[[x_method, (v_args ...), v_fut]])
       K[(actor id_a                                               ⟶ K[(actor id_a                                    [resolve]
             (obj-entity_0 ... (fut id_f q_fut null) (res id_r id_f) obj-entity_n ...)     (obj-entity_0 ... (fut id_f () v) (res id_r id_f) obj-entity_n ...)
             q_actor                                                        q_actor
             E[(((ref id_a id_r) $ resolve-mu) v)])]                        E[fwd[[(ref id_a id_f), q_fut]]])]
                   ((actor id_0 objects_0 q_0 any_0) ...           ⟶ ((actor id_0 objects_0 q_0 any_0) ...          [new-actor]
                    (actor id_a objects_a q_a E[(actor f ... m ...)])        (actor id_a objects_a q_a E[(ref id_new id_o)])
                    (actor id_n objects_n q_n any_n) ...)                     (actor id_new ((object id_o obj f ... m ...)) () null)
                                                                              (actor id_n objects_n q_n any_n) ...)
                                                                              where id_new, id_o fresh
    ((actor id_0  objects_0  q_0  any_0) ...                       ⟶ ((actor id_0  objects_0  q_0  any_0) ...       [asynchronous-send-d]
     (actor id_a1 (obj-entity_a1 ...)  q_a1 E[(send (ref id_a2 id_object) x_message (v_a1 v_2 ...))])     (actor id_a1 (obj-entity_a1 ...)  q_a1 E[null])
     (actor id_1  objects_1  q_1  any_1) ...                          (actor id_1  objects_1  q_1  any_1) ...
     (actor id_a2 objects_a2  q_a2 any_a2)                            (actor id_a2 U[[pass-isolates, objects_a2]] U[[(new-message), q_a2]] any_a2)
     (actor id_n  objects_n  q_n  any_n) ...)                         (actor id_n  objects_n  q_n  any_n) ...)
 where reachable-isolates = reach[[(obj-entity_a1 ...), (v_a1)]], pass-isolates = sigma[[reachable-isolates, (fresh-id ...)]],
     pass-value = (first sigma-v[[(v_a1), (), id_a1, id_a2, reachable-isolates, (fresh-id ...)]]), new-message = (message (ref id_a2 id_object) x_message (pass-value v_2 ...)), (fresh-id...) fresh
    ((actor id_0  objects_0  q_0  any_0) ...                       ⟶ ((actor id_0  objects_0  q_0  any_0) ...       [asynchronous-send-u]
     (actor id_a2 objects_a2  q_a2 any_a2)                            (actor id_a2 U[[pass-isolates, objects_a2]] U[[(new-message), q_a2]] any_a2)
     (actor id_1  objects_1  q_1  any_1) ...                          (actor id_1  objects_1  q_1  any_1) ...
     (actor id_a1 (obj-entity_a1 ...)  q_a1 E[(send (ref id_a2 id_object) x_message (v_a1 v_2 ...))])     (actor id_a1 (obj-entity_a1 ...)  q_a1 E[null])
     (actor id_n  objects_n  q_n  any_n) ...)                         (actor id_n  objects_n  q_n  any_n) ...)
 where reachable-isolates = reach[[(obj-entity_a1 ...), (v_a1)]], pass-isolates = sigma[[reachable-isolates, (fresh-id ...)]],
     pass-value = (first sigma-v[[(v_a1), (), id_a1, id_a2, reachable-isolates, (fresh-id ...)]]), new-message = (message (ref id_a2 id_object) x_message (pass-value v_2 ...)), (fresh-id...) fresh
```

Figure A.10: Plt-Redex reduction rules for the AmbientTalk language.

Bibliography

[ABZ10] Lucia Acciai, Michele Boreale, and Gianluigi Zavattaro. Behavioural contracts with request-response operations. In *Proceedings of the 12th international conference on Coordination Models and Languages*, COORDINATION'10, pages 16–30, Berlin, Heidelberg, 2010. Springer-Verlag.

[ADB$^+$99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1996.

[BBC$^+$06] Laurent Baduel, Francoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BC06] Paolo Bellavista and Antonio Corradi. *The Handbook of Mobile Middleware*. Auerbach Publications, Boston, MA, USA, 2006.

[Ben86] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, August 1986.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, July 1999.

[BKA11] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, pages 2–26. Springer-Verlag, July 2011.

[BM06] Matthias Blume and David McAllester. Sound and complete models of contracts. *Journal of Functional Programming.*, 16:375–414, July 2006.

[BRLM11] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 141–152, 2011.

[Bro87] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–343, 2004.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCDC$^+$09] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theoretical Computer Science*, 410:142–167, 2009.

[CD96] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 3–14, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[CDDS94] Wim Codenie, Koen D'Hont, Theo D'Hondt, and Patrick Steyaert. Agora: Message passing as a foundation for exploring OO language concepts. *SIGPLAN Notices*, 29(12):48–57, 1994.

[CGP08] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 261–272, New York, NY, USA, 2008. ACM.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *DLS '05*, pages 1–10, New York, NY, USA, 2005. ACM.

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. A calculus of global interaction based on session types. *Electron. Notes Theor. Comput. Sci.*, 171(3):127–151, June 2007.

[Cla77] Edmund Melson Clarke, Jr. Programming language constructs for which it is impossible to obtain good hoare-like axiom systems. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 10–20, New York, NY, USA, 1977. ACM.

[CMR03] Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy assertions. In *Implementation of Functional Languages, 15th International Workshop, IFL 2003*, volume 3145 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[Com11] Th. Sc. Community. Towards Social-ICT Organisms. In *Pervasive Adaptation. The Next Generation Pervasive Computing Research Agenda*. Institute for Pervasive Computing, Johannes Kepler University Linz, Austria, May 2011.

[CP09] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*, pages 211–228. Springer Berlin Heidelberg, 2009.

[CR06] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006.

[DCGDY07] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *Proceedings of the 5th international conference on Formal methods for components and objects*, FMCO'06, pages 207–245, Berlin, Heidelberg, 2007. Springer-Verlag.

[DCMYD06] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 328–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[DDCC07] Sophia Drossopoulou, Dezani Dezani-Ciancaglini, and Mario Coppo. Amalgamating the Session Types and the Object-Oriented Programming Paradigms. In *Multiparadigm Programming with Object-Oriented Languages 2007 (an ECOOP workshop)*, August 2007.

[Ded06] Jessie Dedecker. *Ambient-Oriented Programming*. PhD thesis, Faculteit Van Wetenschappen, Programming Technology Lab, Vrije Universiteit Brussel, May 2006.

[DF11] Christos Dimoulas and Matthias Felleisen. On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(5):16:1–16:29, November 2011.

[DFFF11] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, volume 46, pages 215–226, New York, NY, USA, January 2011. ACM.

[DFM11] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *International Conference on Functional Programming*, pages 176–188, 2011.

[DGM$^+$07] Jessie Dedecker, Elisa Gonzalez Boix, Stijn Mostinckx, Stijn Timbermont, Jorge Vallejos, and Tom Van Cutsem. The Ambienttalk/2 Tutorial, 2007. `http://soft.vub.ac.be/amop/at/tutorial/tutorial`.

[DH98] Andrew Duncan and Urs Hoelzle. Adding contracts to Java with HandShake. Technical Report TRCS98-32, University of California at Santa Barbara, Santa Barbara, CA, USA, 1998.

[DJ83] Werner Damm and Bernhard Josko. A sound and relatively complete hoare-logic for a language with higher type procedures. *Acta Informatica*, 20:59–101, 1983.

[DJLS08] Nikhil Dinesh, Aravind Joshi, Insup Lee, and Oleg Sokolsky. Reasoning about conditions and exceptions to laws in regulatory conformance checking. In *Proceedings of the 9th international conference on Deontic Logic in Computer Science*, DEON '08, pages 110–124. Springer-Verlag, 2008.

[DSA01] A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, December 2001.

[DTK06] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63(3):207–239, December 2006.

[DVM$^+$05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, and Theo D'Hondt. Ambienttalk : A small reflective kernel for programming mobile network applications. Technical report, Vrije Universiteit Brussel, 2005.

[ENO82] George W. Ernst, Jainendra K. Navlakha, and William F. Ogden. Verification of programs with procedure-type parameters. *Acta Informatica*, 18:149–169, 1982.

[FB06] Robert Findler and Matthias Blume. Contracts as pairs of projections. *Functional and Logic Programming*, 3945:226–241, 2006.

[FBT06] Yishai A. Feldman, Ohad Barzilay, and Shmuel Tyszberowicz. Jose: Aspects for design by contract. *IEEE International Conference on Software Engineering and Formal Methods*, pages 80–89, 2006.

[FF01] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 48–59, 2002.

[FFF04] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP'04*, volume 3086 of *Lecture Notes in Computer Science*, pages 364–388. Springer, June 2004.

[FFF09] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[FG09] Lidia Fuentes and Nadia Gámez. Modeling the context-awareness service in an aspect-oriented middleware for AmI. *3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008*, pages 159–167, 2009.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[FHRR04] Cédric Fournet, Tony Hoare, SriramK. Rajamani, and Jakob Rehof. Stuck-free conformance. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer Berlin Heidelberg, 2004.

[FII00] Clemens Fischer, Fachbereich Informatik, and Des Fachbereichs Informatik. Combination and Implementation of Processes and Data: from CSP-OZ to Java, 2000.

[FK99] Svend Frølund and Jari Koistinen. Quality of service aware distributed object systems. In *Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems - Volume 5*, pages 69–83, Berkeley, CA, USA, 1999. USENIX Association.

[FL11] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 international conference on Formal verification of object-oriented software*, FoVeOOS'10, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[Fla06] Cormac Flanagan. Hybrid type checking. *SIGPLAN Not.*, 41(1):245–256, January 2006.

[FLF01] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 229–236, New York, NY, USA, 2001. ACM.

[GBLCS$^+$11] E. Gonzalez Boix, A. Lombide Carreton, C. Scholliers, T. Van Cutsem, W. De Meuter, and T. D'Hondt. Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–25, 2011*, volume 1, pages 425–432. ACM, 2011.

[GF99] R. Guerraoui and M. E. Fayad. OO Distributed Programming is *Not* Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GMFK07] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 29–40, New York, NY, USA, 2007. ACM.

[GPW10] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 353–364, 2010.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[GSL$^+$10] Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Scripting mobile devices with ambienttalk. pages 202–224. IGI Global, 2010.

[GVR$^+$10] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th symposium on Principles of programming languages*, POPL '10, pages 299–312, New York, NY, USA, 2010. ACM.

[HBG$^+$01] Franz J. Hauck, Ulrich Becker, Martin Geier, Erich Meier, Uwe Rastofer, and Martin Steckermeier. Aspectix: A quality-aware, object-based middleware architecture. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 115–120, 2001.

[HBT12] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. *SIGPLAN Not.*, 47(1):111–122, January 2012.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pages 169–180, New York, NY, USA, 1990. ACM.

[HJL06] Ralf Hinze, Johan Jeuring, and Andres Loh. Typed contracts for functional programming. In *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[HO07] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proc. of the 9th inter. conf. on Coordination models and languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hol] Bernhard Hollunder. Interface contracts for wcf services with code contracts. *International Journal On Advances in Software*, 4:275–285.

[Hol92] Ian M. Holland. Specifying reusable components using contracts. In *European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, pages 287–308, Berlin, Heidelberg, 1992. Springer-Verlag.

[KCR98] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998 and ACM SIGPLAN Notices, Vol. 33, No. 9, September, 1998*, 1998.

[KHB99] Murat Karaorman, Urs Hölzle, and John L. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 175–196, London, UK, 1999. Springer-Verlag.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[LCC$^+$05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55:185–208, March 2005.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 214–223. ACM Press, 1986.

[LKP02] Martin Lackner, Andreas Krall, and Franz Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3):57–76, 2002.

[LN99] Baochun Li and Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, 17:1632–1650, 1999.

[Lom11] Andoni Lombide Carreton. *Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Software Languages Lab, October 2011.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20:564–576, August 1977.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[Mey91] B. Meyer. *Eiffel : The Language*. Prentice Hall Object-Oriented Series, 1991.

[Mey92] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, October 1992.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, March 2000.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1st edition, June 1999.

[Mil06] M. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, John Hopkins University, Baltimore, Maryland, USA, May 2006.

[MKD03] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 46–60, Berlin, Heidelberg, 2003. Springer-Verlag.

[MM09] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[MSP$^+$07] Stijn Mostinckx, Christophe Scholliers, Eline Philips, Charlotte Herzeel, and Wolfgang De Meuter. Fact spaces: Coordination in the face of disconnection. In *Proceedings of 9th International Conference on Coordination Models and Languages*, volume 4467 of *Lecture Notes* *in Computer Science*, pages 268–285, Heidelberg, June 2007. Springer-Verlag.

[MTS05] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[MY07] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *Proceedings of the 8th international conference on Typed lambda calculi and applications*, TLCA'07, pages 321–335, Berlin, Heidelberg, 2007. Springer-Verlag.

[NMO09] Piotr Nienaltowski, Bertrand Meyer, and Jonathan S. Ostroff. Contracts for concurrency. *Form. Asp. Comput.*, 21(4):305–318, July 2009.

[Ric53] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[RM09] Kay Romer and Junyan Ma. Pda: Passive distributed assertions for sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 337–348, Washington, DC, USA, April 2009. IEEE Computer Society.

[RSA$^+$12] Coen De Roover, Christophe Scholliers, Wouter Amerijckx, Theo D'Hondt, and Wolfgang De Meuter. CrimeSPOT: a language and runtime for developing active wireless sensor network applications. *Science of Computer Programming*, 2012.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. MIT Press, Cambridge, MA, USA, 1993.

[SBMD10] Christophe Scholliers, Elisa Gonzalez Boix, Wolfgang De Meuter, and Theo D'Hondt. Context-aware tuples for the ambient. In *Proceedings of the 12th International Symposium on Distributed Objects, Middleware, and Applications*, pages 745–763. Springer, 2010.

[SF04] Neelam Soundarajan and Stephen Fridella. Incremental reasoning for object oriented systems. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 302–333. Springer Berlin / Heidelberg, 2004.

[SF10] T. Stephen Strickland and Matthias Felleisen. Contracts for First-Class Classes. In *Proceedings of the 6th symposium on Dynamic languages (DLS)*, pages 97–112, 2010.

[SGD09] Christophe Scholliers, Elisa Gonzalez Boix, and Wolfgang De Meuter. Totam: Scoped tuples for the ambient. In *Proc. of the CAMPUS Workshop collocated with DisCoTec'09 federated event*, volume 19, pages 19–34. EASST, 2009.

[SHSDM11] Christophe Scholliers, Lode Hoste, Beat Signer, and Wolfgang De Meuter. Midas: a declarative multi-touch interaction framework. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, TEI '11, pages 49–56, New York, NY, USA, 2011. ACM.

[SHT$^+$11] Christophe Scholliers, Dries Harnie, Éric Tanter, Wolfgang De Meuter, and Theo D'Hondt. Ambient Contracts: Verifying and Enforcing Ambient Object Compositions á La Carte. *Personal Ubiquitous Computing*, 15(4):341–351, April 2011.

[SL04] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 196–197, New York, NY, USA, 2004. ACM.

[SLN07] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 351–368, New York, NY, USA, 2007. ACM.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128, Berlin, Heidelberg, 2008. Springer.

[SNS$^+$11] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, pages 713–732, 2011.

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: generalizing active objects to concurrent components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[Sti04] Marc Stiegler. The E language in a walnut. *www.skyhunter.com/marcs/ewalnut.html*, 2004.

[STM11] Christophe Scholliers, Eric Tanter, and Wolfgang De Meuter. Computational contracts. In *Proceedings of the 2011 Workshop on Scheme and Functional Programming*, Portland, Oregon, October 2011.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[Sza02] V. Szathmary. Barter - beyond design by contract. http://barter.sourceforge.net/, 2002.

[Tan08] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 168–179, New York, NY, USA, 2008. ACM.

[Tan10a] Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 37–48, Rennes and Saint Malo, France, March 2010. ACM Press.

[Tan10b] Éric Tanter. LAScheme. http://pleiad.cl/research/lascheme, 2010.

[TFD$^+$10] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Scoping strategies for distributed aspects. *Sci. Comput. Program.*, 75(12):1235–1261, December 2010.

[TGDB06] Eric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. volume 4089 of *LNCS*, pages 227–249. Proceedings of the 5th International Symposium on Software Composition (SC 2006), 2006.

[THVH12] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 537–554, New York, NY, USA, 2012. ACM.

[TS03] Benjamin Tyler and Neelam Soundarajan. Black-box testing of grey-box behavior. In *FATES'03*, pages 1–14, 2003.

[UBPU08] Aitor Urbieta, Guillermo Barrutieta, Jorge Parra, and Aitor Uribarren. A survey of dynamic service composition approaches for ambient systems. In *SOMITAS '08*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2008.

[UPST08] Naoyasu Ubayashi, Jinji Piao, Suguru Shinotsuka, and Tetsuo Tamai. Contract-based verification for aspect-oriented refactoring. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 180–189, Washington, DC, USA, 2008. IEEE Computer Society.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001*, 36(12):20–34, December 2001.

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[VCSHDM] Tom Van Cutsem, Christophe Scholliers, Dries Harnie, and Wolfgang De Meuter. An Operational Semantics of Event Loop Concurrency in Ambienttalk. Technical Report VUB-SOFT-TR-12-04.

[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 *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.

[Wam06] Dean Wampler. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. *Fifth AOSD Workshop on ACP4IS*, 2006.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.

[XPJC09] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 41–52, New York, NY, USA, 2009. ACM.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* Computer Systems Series. MIT Press, 1990.

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems (TAPOS)*, 3(1):55–73, 1997.

[ZR03] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Fundamental Approaches to Software Engineering (FASE 2003)*, pages 150–165, Berlin, Heidelberg, April 2003. Springer-Verlag.