"Programming in ambience : gearing up for dynamic adaptation to context"

Gonzalez Montesinos, Sebastian Andres

ABSTRACT

In the vision of Ambient Intelligence, people are assisted in their everyday activities through the proactive, opportunistic support of non-intrusive computing devices offering intuitive interaction modalities. The usefulness and quality of delivered services can be improved considerably if the devices are able to adapt their behaviour according to sensed changes in their surrounding environment, both at the physical and logical levels. This interplay between context-awareness and dynamic software adaptability is key to the construction of applications that are smart with respect to user needs. Unfortunately, most current applications do not reach this level of adaptability, due to a lack of appropriate programming technology. Most applications exhibit fixed functionality and seldom do they sense their environment and adapt their services in a context-aware fashion. Many chances of delivering improved services to users and network peers are thus missed. This dissertation presents a p...
Programming in Ambience
Gearing up for dynamic adaptation to context
Dissertation

Programming in Ambience
Gearing up for dynamic adaptation to context

Sebastián González Montesinos

24th October 2008

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor in Engineering Sciences

Thesis Committee:
Prof. Yves DEVILLE (Chair) INGI/UCL, Belgium
Prof. Kim MENS (Promoter) INGI/UCL, Belgium
Dr. Pascal COSTANZA PROG/VUB, Belgium
Prof. Theo D’HONDT PROG/VUB, Belgium
Prof. Oscar NIERSTRASZ SCG/U.Bern, Switzerland
Prof. Peter VAN ROY INGI/UCL, Belgium
Programming in Ambience
Gearing up for dynamic adaptation to context

© 2008 Sebastián González Montesinos
Département d’Ingénierie Informatique
École Polytechnique de Louvain
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work has been supported by a grant from the Special Research Fund† of Université catholique de Louvain, by a grant from the Fund for Research Training in Industry and Agriculture* of the French Community in Belgium, by the MoVES project of the Interuniversity Attraction Poles Programme of the Belgian Science Policy, Belgian State, and by the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIIB).

This manuscript has been set with the help of KOMA-Script, PDFLaTeX, and GNU Emacs (with AUCTeX and REFTeX support). Bugs have been tracked down in the text and squashed thanks to Bugs in Writing by Dupré [45] and to Elements of Style by Strunk and White [108].

† Fonds Spéciaux de Recherche (FSR).
* Fonds pour la formation à la Recherche dans l’Industrie et dans l’Agriculture (FRIA).
To Martín, Sonia and Héctor
Abstract

In the vision of Ambient Intelligence, people are assisted in their everyday activities through the proactive, opportunistic support of non-intrusive computing devices offering intuitive interaction modalities. The usefulness and quality of delivered services can be improved considerably if the devices are able to adapt their behaviour according to sensed changes in their surrounding environment, both at the physical and logical levels. This interplay between context-awareness and dynamic software adaptability is key to the construction of applications that are smart with respect to user needs. Unfortunately, most current applications do not reach this level of adaptability, due to a lack of appropriate programming technology. Most applications exhibit fixed functionality and seldom do they sense their environment and adapt their services in a context-aware fashion. Many chances of delivering improved services to users and network peers are thus missed.

This dissertation presents a programming model to ease the construction of applications that can react to changes in their execution context by adapting their behaviour dynamically. The starting point of our research is the development of novel language abstractions and the adaptation of existing abstractions to render context-aware, self-adaptable applications easier to develop. We demonstrate that a simple yet powerful computation model readily provides the needed support, leading to straightforward application code that is not concerned with context adaptation, behaviour that can be adapted dynamically to different contexts in a non-intrusive fashion, and context-aware applications with software architectures that are not biased towards context adaptation —rather, they can be designed freely according to their domain.

The proposed computation model is realised through the Ambience programming language, and its underlying open implementation, the Ambient Object System. A small-step operational semantics describes it formally. Much in the vein of prototype-based programming, the model has been designed with simplicity and concreteness in mind. It is highly dynamic, featuring dynamic (multiple) dispatch, dynamic inheritance, dynamic typing, and dynamic method scoping. Application logic adaptation is enabled by means of an intuitive, first-class reification of context that is straightforwardly connected to dynamic behaviour selection. We describe needed management techniques for such context, and a few programming guidelines on how to develop context-aware applications using our approach. The approach is validated by showing its application in a number of scenarios inspired on Ambient Intelligence.
Acknowledgements

My deepest gratitude goes to Prof. Kim Mens for his continued support throughout these years. The role of adviser is more demanding than one might think, and —I believe— requires many skills beyond scientific knowledge. Kim gave me the freedom I needed to develop my own ideas, while keeping an interest in them and following up closely. He has always been available, ready to discuss, contribute and provide needed advice. As a manager, Kim has been expeditious in securing all needed resources, and following up on administration chores of all kinds. On a personal level, I am lucky to have had the friendliest and most sympathetic promoter I could have hoped for.

I am also indebted to Dr. Pascal Costanza. He has given me valuable advice in various occasions, including pointers to related work which influenced some of the design decisions behind Ambience. As is usually done by program chairs, Pascal invited me to make submissions to the Dynamic Languages Symposium in 2007 and to the European Lisp Symposium in 2008. Both papers were accepted and now make the core of chapters 4 and 5.

I thank all jury members for the time they took in evaluating this work and for their feedback, which helped to increase the quality of the text considerably.

Even though I started out as sole member of Kim’s team, more members have joined in and have contributed to making these years a pleasant experience: Diego Ordoñez, Johan Brichau, Sergio Castro and Alfredo Cádiz. Thanks to this RELEASeD team. Their contributions to this dissertation come from fruitful discussions in our regular research meetings.

The INGI department in general has offered a very agreeable working atmosphere. I thank all fellow researchers for their friendliness and interesting discussions in the cafeteria, train, jogging sessions and otherwise.

Also influential to this work —probably more than it is apparent— are Prof. Wolfgang De Meuter and Prof. Theo D’Hondt. They taught me the concepts that make up the foundations of Ambience, when I followed the European Master on Object-Oriented Software Engineering between 2001 and 2002. In the years to follow, during the journey towards this dissertation, both Wolf and Theo were ready to help when I needed it most.

I am indebted to Prof. Baudouin Le Charlier for taking over most of the workload from a course of which I was an assistant, so that I could spend more time writing this dissertation. Time is probably the most precious gift one can receive. Damien Saucez, Sébastien Combéfis and the members of the RELEASeD team also helped in this regard.

My enthusiasm for doing research was inspired early on by Prof. José Tiberio Hernández. It is thanks to his good judgement that some doors remained open.
and I could arrive to this point in my career. I owe him deep gratitude.

My gratitude extends to the Belgian nation, which has supported my studies through the years with funding ultimately coming from people’s taxes. The Belgian state and particularly the French Community in Belgium were so open as to allow a non-European to apply for the FRIA grant shoulder-to-shoulder with Europeans, and Belgians in particular. The Université catholique de Louvain supported my studies much as openly, by means of a grant and an assistantship.

Last, but not least, I thank my family and friends. They hold my warmest thoughts and constitute an integral part of myself. Without them it would have been impossible to bring this dissertation to an end.

Sebastián González
15th October 2008
# Contents

1. Introduction ................................................. 1  
   1.1. Motivation ................................................ 1  
   1.2. Problem Statement ....................................... 3  
   1.3. Thesis Statement ......................................... 4  
   1.4. Approach .................................................. 5  
   1.5. Contributions ............................................. 6  
      1.5.1. Language Engineering ................................. 6  
      1.5.2. Publications ......................................... 7  
      1.5.3. Implementation ....................................... 7  
   1.6. Roadmap .................................................. 8  

2. Background .................................................. 11  
   2.1. Flexible Behaviour Selection ............................. 12  
      2.1.1. Subject-Oriented Programming ....................... 12  
      2.1.2. Subjective Objects .................................. 14  
      2.1.3. Context-Oriented Programming ....................... 16  
      2.1.4. Predicate Dispatch ................................... 19  
   2.2. Flexible Behaviour Composition .......................... 20  
      2.2.1. Dynamic Inheritance .................................. 21  
      2.2.2. Predicate Objects .................................... 23  
      2.2.3. Traits ................................................ 25  
      2.2.4. Dynamic Aspect Weaving .............................. 26  

3. The Ambience Programming Language .......................... 29  
   3.1. Core Concepts ............................................. 29  
   3.2. Syntax overview .......................................... 31  
      3.2.1. Literals ............................................... 31  
      3.2.2. Messages .............................................. 33  
      3.2.3. Methods .............................................. 36  
      3.2.4. Comments .............................................. 39  
   3.3. Ambience Objects .......................................... 39  
      3.3.1. Object Creation ...................................... 39  
      3.3.2. Object Evolution ..................................... 40  
      3.3.3. Object Delegation .................................... 40  
   3.4. Ambience Multimethods .................................... 41  
      3.4.1. Multimethod Specialisation ........................... 41  
      3.4.2. Multimethod Applicability ............................ 42  
      3.4.3. Multimethod Specificity ............................... 42
3.5. Ambience Messages ........................................ 44
3.5.1. Ambiguous Messages ............................... 44
3.5.2. Ambiguity Resolution Strategies ................. 46
3.5.3. Message Resends .................................... 55

4. Context-Aware Programming for Ambient Intelligence 57
4.1.1. Context Discovery ................................ 59
4.1.2. Context Management .............................. 62
4.1.3. Assumptions ...................................... 63
4.2. Ambience Contexts .................................... 63
4.2.1. Context Representation ......................... 64
4.2.2. Dynamic Context Switching ...................... 66
4.2.3. Idiosyncratic Contexts ............................ 66
4.2.4. Influence of Context on Object Behaviour ........ 67
4.3. Context Management .................................. 68
4.3.1. Framework Contexts .............................. 68
4.3.2. Context Combinations ............................ 70
4.3.3. Consistent Context Activation ................... 73
4.4. Fragility Due to Dynamic Behavioural Changes ..... 78
4.4.1. Fragility Sources ................................ 78
4.4.2. Fragility Cases .................................. 80

5. The Ambient Object System 85
5.1. Mapping Ambience to AmOS ............................ 85
5.1.1. Syntax Mapping ................................ 85
5.1.2. Standalone AmOS ................................ 86
5.2. Opening Up AmOS .................................. 89
5.2.1. Objects: Slots, Maps and Delegation .......... 89
5.2.2. Closures and Activations ....................... 92
5.2.3. Methods, Specialisation and Lookup ........... 93
5.2.4. Contexts ....................................... 95
5.2.5. Lookup Arguments ............................... 99
5.2.6. Accessor Methods ............................... 101
5.2.7. Message Resends ................................. 103
5.2.8. Targeted Resends ................................. 104
5.2.9. Bypass Resends ................................ 105
5.2.10. Sealed Slots .................................. 106
5.3. Main Protocols .................................... 107
5.3.1. Meta-Object Protocol ............................ 107
5.3.2. Context Object Protocol ......................... 109

6. Formal Semantics 113
6.1. Background ............................................ 113
6.2. Abstract Grammar .................................... 114
6.2.1. Program syntax .................................. 114
6.2.2. Evaluation contexts ......................... 117
6.3. Canonical Store .................................. 119
6.4. Reduction Rules ................................. 121
  6.4.1. Storage .................................. 121
  6.4.2. Slots .................................. 121
  6.4.3. Roles .................................. 122
  6.4.4. Execution ................................ 123
  6.4.5. Closures ................................ 124
  6.4.6. Methods ................................ 126
  6.4.7. Accessors ................................ 127
  6.4.8. Dispatch ................................ 129
6.5. Delegation Graph Linearisation .................. 132
  6.5.1. Notation and Terminology .................. 132
  6.5.2. General definitions ......................... 132
  6.5.3. Definition of C3* .......................... 133
  6.5.4. Examples ................................ 134
6.6. Relationship to the Implementation .............. 137

7. Assessment of Expressiveness of AmOS 139
  7.1. Validation Approach ............................ 139
    7.1.1. Points to Validate (the What and the Why) 139
    7.1.2. Validation Setup (the How) ................ 140
  7.2. Case Studies ................................ 141
    7.2.1. Adaptive Video Player ..................... 142
    7.2.2. Software Feature Interaction in a Mobile Phone 152
    7.2.3. Transactional Contexts ................... 165
  7.3. Wrap Up ................................... 177

8. Conclusions 181
  8.1. Contributions ................................ 181
  8.2. Related Work ................................ 185
    8.2.1. Us ................................ 185
    8.2.2. Cecil ................................ 186
    8.2.3. Prototypes with Multiple Dispatch (PMD) and Slate 187
    8.2.4. CLOS ................................ 187
    8.2.5. ContextL ................................ 188
    8.2.6. Contextual Values ......................... 189
  8.3. Limitations and Future Work ................... 189

A. Semantics Companion 197
  A.1. Term Rewriting Systems in a Nutshell ......... 197
  A.2. Complementary Reduction Rules ................ 198

B. Formal Syntax of Ambience 201
  B.1. Scanner ................................ 201
  B.2. Values ................................ 201
B.3. Messages ......................................................... 202
B.4. Methods .......................................................... 202
B.5. Auxiliary productions .......................................... 203

Bibliography ......................................................... 207
List of Figures

3.1. Applicability of a method for a given message .................................. 43
3.2. A more specific version of receive:on: .............................................. 44
3.3. Two ambiguous versions of pair:with: .............................................. 45
3.4. Smartphone delegation graph ............................................................ 46
3.5. Inheritance graph for which CLOS is not monotonic .......................... 50
3.6. Inheritance graph for which CLOS does not preserve extended
    precedence ......................................................................................... 51
3.7. Inconsistent delegation graphs .......................................................... 52
3.8. Message resend .................................................................................. 55

4.1. General context-aware system architecture ...................................... 58
4.2. Types of context information ............................................................... 59
4.3. Sample context configuration .............................................................. 65
4.4. Combined contexts and their associated behaviour ........................... 76

5.1. Cloning family maps and their copy-on-write policy .......................... 91
5.2. Delegation objects ............................................................................. 91
5.3. Prototypical activation and cloned activation with actual arguments .... 92
5.4. Method applicability for a given message ......................................... 93
5.5. Roles corresponding to the receive method ...................................... 94
5.6. Invocation of the receive method ....................................................... 98
5.7. Context graph as combination of contexts ........................................ 99
5.8. Cloned object behaviour ................................................................... 100

6.1. Program abstract syntax .................................................................... 115
6.2. Evaluation contexts .......................................................................... 117
6.3. Initial store ......................................................................................... 120
6.4. Storage manipulation reductions ..................................................... 121
6.5. Slot manipulation reductions ............................................................ 122
6.6. Role manipulation reductions ............................................................ 123
6.7. Reductions related to threads ............................................................. 123
6.8. Control flow reductions ..................................................................... 124
6.9. Reduction rules for closure literals ................................................... 124
6.10. Closure reductions ........................................................................... 126
6.11. Method reductions ........................................................................... 127
6.12. Accessor reductions ......................................................................... 128
6.13. Outer accessor reductions ............................................................... 128
6.15. Method ranking reductions. ......................... 131
6.16. Sample delegation graphs. .......................... 134

7.1. Context structure for ACME’s video player case study. .... 144
7.2. Context configuration for ACME’s mobile phone case study. .. 159
7.3. Transactional contexts. ............................... 175

8.1. Idea for supporting activation-local contexts. .............. 190

A.1. Plotkin’s $\lambda_v$ calculus. ......................... 197
A.2. Slot manipulation error reductions. .................... 199
A.3. Delegation manipulation reductions. .................... 199
A.4. Delegation manipulation error reductions. ............... 200
A.5. Delegation accessor reductions. ....................... 200
1 Introduction

This dissertation presents a novel programming model to ease the construction of dynamically adaptable applications that can exhibit context-dependent behaviour. We hypothesise that this kind of application will be the rule, rather than the exception, in the emerging field of Ambient Intelligence (AmI) [67].

1.1. Motivation

Personal computing is moving towards computers embedded into consumer electronics and household equipment in the form of devices with varying amounts of memory, processing power and diverse programmable peripherals. Some of the new devices tend to be domain-specific (e.g. music players, global positioning systems, domotic appliances, medical appliances), offering tailor-made services for specific usage scenarios.

The usefulness and quality of delivered services can be improved considerably if the devices are able to sense and act upon their surrounding environment, both at the physical and logical levels. As a simple example, modern laptops adjust their backlight keyboard and screen brightness dynamically, thanks to an ambient light sensor. At the logical level, service discovery protocols such as DNS Service Discovery (DNS-SD) have set the stage for service-oriented architectures in mobile networks, such that printers and file servers can be found on the fly. Using sensors and mobile network infrastructure, far more advanced application interactions and adaptations than the ones just mentioned can be envisaged [43]. In the vision of Ambient Intelligence in particular [99], people are assisted in their everyday activities through the proactive, opportunistic support of non-intrusive computing devices offering intuitive interaction modalities.

Current technology thus heightens the expectation that applications running on the new platforms will turn from mere isolated programs to smart software that can interact with its environment. Ideally, applications should be aware of their execution context,¹ and should adapt dynamically to such context so that they can provide a service that matches the user expectations to the best extent possible. We call this kind of applications that are smart with respect to the user needs and adaptable to their environment ambient applications.

As mobile computing progresses towards the vision of ambient applications, full dynamic software adaptation to the context becomes increasingly impor-

¹Here context is used in a broad sense: people and objects in the vicinity, environmental properties such as lighting and noise, device status such as battery charge and network signal strength, available network peers and the services they offer, and so on.
tant: the capability of a program to respond to changes that occur in its operating environment through the dynamic transformation and reconfiguration of its components and services. Ambient applications question the underlying assumption that a single application behaviour can be articulated and anticipated completely, and replace it with the view that application behaviour should change in different circumstances, sometimes even in unanticipated ways [40], due for example to behaviour emergence.

The following scenario is aimed at giving the reader a better feeling of the kind of adaptations and collaboration expected from ambient applications.

**Scenario: collaboration between a smartphone and a GPS**

The following simple, down-to-earth scenario illustrates dynamic context-aware behaviour adaptation in a typical ambient application. Thanks to domain-specific devices that can be used *in situ* and are able to communicate in a peer-to-peer fashion, potential collaborations are boosted to a point where they become synergic: the behaviour obtained from the reunion of different devices is more adapted to the context and useful to the user than the behaviour of each device in isolation. Programming languages geared towards Ambient Intelligence should be able to deal with such a scenario naturally.

The CityMaps application for smartphones contains static maps of cities, annotated with information such as street names and special spots (hospitals, hotels, public transportation stops), much like the maps one could find in a tourist book or on Google Maps. Although the CityMaps service is useful on its own, modern users expect more dynamic features. The ACME company has detected such expectation and has developed a Global Positioning System (GPS) hardware module for smartphones. Once connected, the module enhances the functionality of CityMaps such that the map section drawn on the screen is updated in real-time according to the current location of the user, detected through GPS. Additionally, the user’s avatar is drawn at the right spot of the map. The net effect of connecting the GPS module to the smartphone is that the CityMaps application can be used in a more navigational fashion. When the module is disconnected, the application reverts to its default static behaviour.

Instead of a hardware module, an alternative scenario would be that the GPS service were provided wirelessly by a network peer, for example an on-board GPS system in a car. The actual details concerning how a service is acquired (e.g. from the network or by physically plugging in a module) and released (e.g. by shutting down the network connection or removing the module) are irrelevant to the point we want to make. We consider all such scenarios equivalent. The important point is that application behaviour can be improved or degraded dynamically according to the services available in the environment.

---

²See [http://maps.google.com](http://maps.google.com).
1.2. Problem Statement

Current applications that run on modern mobile devices are still far from achieving the vision of Ambient Intelligence. Applications are hardly able to cooperate and adapt dynamically to the context. Most applications exhibit fixed functionality and fixed communication patterns; they seldom exploit service discovery or adapt their services in a context-aware fashion. Many chances of delivering improved services to users and network peers are thus missed.

The lack of cooperation and adaptability in current mobile applications is caused to a large extent by a lack of appropriate programming technology, particularly of languages that support dynamic behaviour adaptability. Using current languages, the adaptability of an application often is a design aspect derived from the software architecture. For instance, the Factory design pattern [55] allows the introduction of a certain degree of variability in the composition of software, by letting third-party code become an active part of an application. The State pattern [55] allows applications to change their behaviour at runtime by reconfiguring the collaborations among their components (objects). More involved mechanisms can be built on top of these basic techniques, most notably the plug-in architectures of many large-scale applications. Unfortunately, in all existing techniques variability points are fixed by design, and little has been achieved regarding the interplay between context-awareness and dynamic software variability. This interplay is key to the construction of applications that are smart with respect to the user needs and adaptable to the current environment.

Given that context dependencies pervade the behaviour of ambient applications, the lack of linguistic support for encoding context-dependent behaviour forces programmers to scatter these dependencies throughout application code in the form of “if” statements, or worse, to bias their software architecture towards a fixed set of adaptable points by means of design patterns. We claim that the degree of dynamic software variability required by ambient applications is so high that it becomes impossible to fully articulate in the software architecture —through the use of advanced design patterns, let alone simple if statements— all variability points that could possibly prove useful, because the class of software interaction scenarios arising from mobility and network communication is too large. A more systematic approach to expressing behavioural dependency on context is needed.

The problem not only concerns the expressiveness of programming languages and their underlying computation models described previously, but also (and

---

3The problem with encoding dependency on context through conditional statements is that the programmer needs to break open existing code and modify it to accommodate new contexts.
perhaps more importantly) their *programming models* as a whole —that is, the programming techniques and design principles used to write programs in those languages [118]. Current paradigms force programmers to think in isolation, in a way that is mostly oblivious of the context in which the software will be used. What is needed is a new paradigm that helps overcoming this limiting vision by putting programmers in the right state of mind to build dynamically adaptable applications from the ground up.

### 1.3. Thesis Statement

Real ambient applications —and ambient-minded programmers— require specific linguistic support to deal with behavioural context dependencies. This dissertation demonstrates that a simple yet powerful computation model based on classless objects and multimethods readily provides such support, leading to

- a. behaviour that can be adapted dynamically to different contexts,
- b. straightforward application logic with no added complexity due to extrinsic context adaptation concerns,
- c. simplified application logic thanks to exploitation of context-oriented programming for adaptation to intrinsic contexts or states,
- d. non-intrusive expression of adaptations (i.e. adaptation code can be introduced without modifying base application code), and
- e. context-aware applications with software architectures that are not biased towards context adaptation —rather, they can be designed in closer correspondence to their domain.

To make clear the distinction between *intrinsic* and *extrinsic* contexts mentioned in items b and c, consider again the GPS scenario. Having a GPS is extrinsic to CityMaps —it is not part of the core functionality of the application. Item b states that the code of CityMaps need not be written with adaptation to GPS services in mind, or any other possible adaptation of this kind. However, suppose CityMaps can work in *map*, *satellite* and *terrain* view modes. The visualisation behaviour of the application varies according to the current mode. Item c states that the code of CityMaps can be more easily written by leveraging the context-oriented machinery we propose, if the visualisation modes are regarded as internal sorts of contexts. These contexts are intrinsic to the definition and normal operation of CityMaps —they are an essential part of the application logic.

---

4 These examples are inspired on Google Maps, see http://maps.google.com for illustration.
1.4. Approach

There is a duality between patterns and abstractions. When design patterns and idioms become pervasive in application code, they can be absorbed as abstractions of another—perhaps more purpose-specific—language. However, in context-aware applications we are confronted with a more critical situation where design patterns cannot be so general as to suit all variability scenarios arising in ambient computing, whereas adequate programming language abstractions can allow—as we intend to show—flexible software compositions and adaptability to context.

The need for adequate programming abstractions that enable application context awareness has given rise to Context-Oriented Programming (COP) [65, 57, 68]. Our approach follows the same direction. The starting point of our research is the development of novel language abstractions and the adaptation of existing abstractions that would render context-aware, self-adaptable ambient applications easier to develop. We have applied Occam’s Razor principle by striving for a core model that is as small and uncomplicated as possible. A second principle to which we stick is homogeneity: all language abstractions and mechanisms should be ultimately expressed in terms of the minimal core. Hence, we have attempted to avoid the introduction of exceptional cases and asymmetric constructs. Yet a third principle is flexibility: a language designed for Ambient Intelligence should accommodate the dynamic nature of its execution platform and environment [36]. Static or rigid constructs would cause a sort of impedance mismatch between the computational systems found in the wild and their description (or rather, simulation [33]) in terms of objects and messages that must be encoded by programmers. Simplicity, homogeneity and flexibility are thus at the heart of our approach.

The result of our language engineering experiment is a computation model that supports highly dynamic behaviour adaptation without hard-coded, cross-cutting conditional code. Further, our approach avoids the use of design patterns (such as those mentioned in section 1.2) and dedicated software architectures. The abstractions we propose are based on a prototype-based model with multiple dispatch and subjective object behaviour. We thus obtain a flexible model in which behaviour not only depends on the message arguments, but also on the context in which the message is sent. Hence, the behaviour exhibited by objects is intrinsically bound to the current (changing) circumstances in which these objects are used.

To validate the proposed set of abstractions we developed Ambience, a new programming language geared towards Ambient Intelligence. Ambience constitutes a proof by construction of the compatibility of the abstractions we propose. The suitability of these abstractions for dynamically adaptable applications programming is demonstrated through the implementation of typical Ambient Intelligence scenarios, such as the one introduced in section 1.1.

5In general, this principle states that when there are two explanations for the same phenomenon, then the explanation which uses the smallest number of assumptions and concepts must be the right one [34].
1.5. Contributions

The contributions are presented in three categories, regarding scientific content, scientific dissemination and implementation.

1.5.1. Language Engineering

The contributions of this dissertation are situated at the intersection of three paradigmatic views on computation models, which happen to be complementary:

**Subjective Programming**

Subjectivity has been investigated to a limited extent so far, after the seminal papers by Harrison and Ossher [64] and Smith and Ungar [101] appeared. Our contribution in this regard consists in assessing the suitability of subjective language features for Ambient Intelligence. In doing so we provide insight on the pragmatics and applicability of subjectivity in object-oriented languages.

**Prototypes with Multiple Dispatch (PMD)**

Our exploration of context-oriented programming in a prototype-based language with multimethods has led us to the design of novel constructs that help homogenising the semantics and implementation of the model. We further contribute an executable, small-step operational semantics of our PMD model, expressed as a term rewriting system. Since it can be tested and the reduction process be visualised using a graphical browser [78], we have a high level of confidence that it encodes the intended meaning and has no inconsistencies or design flaws. Lastly, we improve on the syntax design of Smalltalk-like languages, adapting it so that it better fits languages with symmetric multimethods. This results in constructs that read even more naturally than any other language inspired on Smalltalk we are aware of.

**Context-Oriented Programming**

Although dedicated language abstractions for context-oriented programming have been proposed previously [29], we contribute to context-oriented programming in a number of ways. Firstly, we propose an alternative, prototype-based model, thereby showing that the advantages of context-oriented programming can be realised in a classless setting. Secondly, we propose a concrete representation of context as a graph of delegating objects and show a number of context management techniques and idioms for managing such first-class context; to the extend of our knowledge, similar techniques have not been explored before. Thirdly, we contribute new language abstractions for context-oriented programming that ease the expression of adaptive code. Fourthly, we contribute experience to the field by showing particular ways of exploiting context-dependent behaviour for adaptable applications.
1.5. Contributions

1.5.2. Publications

The following peer-reviewed publications contain the essence of the contributions just mentioned:\(^6\)


The first paper introduces context-oriented programming in Ambience, showing the techniques we developed to deal with behaviour that depends on a combination of different contexts, and discusses the issues that arise from concurrent manipulation of the context representation. In contrast to the first paper which describes the programming model we propose (abstractions, idioms, frameworks), the second paper concentrates on the underlying computation model, the Ambient Object System (AmOS), and its implementation in Common Lisp.

1.5.3. Implementation

Our implementation of the proposed computation model can be used for rapid prototyping of language constructs related to the paradigms mentioned previously. Thanks to the provided Meta Object Protocol (MOP), many experiments can be run inside the computation model, even if they modify the meaning of fundamental mechanisms such as message sending. We show an experiment of extensibility that exploits this MOP and the context-oriented machinery by implementing a transaction isolation mechanism. For cases that modify the core model in ways that the MOP cannot accommodate, the implementation can be adjusted easily thanks to a careful design and the use of Common Lisp—a higher order, dynamically typed language with a powerful macro system and good IDE support. Thanks to the use of a high-level language, the code base is relatively small and easy to understand.\(^8\) In our experience, the effort associated to the addition or modification of features has proved quite reasonable.

---

\(^6\)Earlier forms of the ideas contained in this dissertation were presented as workshop papers at the European Conference on Object-Oriented Programming in 2004, 2005 and 2006. The titles of the workshops in which we took part show the need for the kind of research we have carried out: *Object-Oriented Language Engineering for the Post-Java Era* and *Object Technology for Ambient Intelligence*.

\(^7\)An earlier version of this article appeared in the proceedings of the 1\(^{st}\) European Lisp Symposium (2008).

\(^8\)The core model counts less than 3400 lines of code to date.
The complete source code can be downloaded from the project homepage, located at http://ambience.info.ucl.ac.be. It includes the core model with accompanying unit tests, and a small context-oriented library developed for the validation cases presented in chapter 7. The source also includes the semantics in PLT Redex format [78], together with a small unit test suite that helps verifying preservation of the semantics after modifications (with a view to extending the model in the future).

1.6. Roadmap

The dissertation starts with an overview of related approaches. From this general perspective, it moves to a high-level (end-user) view of our own approach, and then progresses towards a low-level (meta-programmer or language implementer) view that shows the inner workings of the model. The final part is concerned with validation.

Chapter 2 discusses existing techniques that allow flexible run-time behaviour composition and selection. These two capabilities are the building blocks of dynamic behavioural adaptability. The discussion gives an overview of the state of the art, showing general advantages and disadvantages of previous work. Some of the approaches mentioned in the chapter have been inspirational in developing our own.

Chapter 3 presents the Ambience programming language and its underlying object model, deferring to the next chapter the part that is geared towards dynamic adaptation to context. The chapter thereby contains the description of a prototype-based language with multimethods akin to Slate [93] and Cecil [17]. An understanding of the object model described in this chapter is a prerequisite to understanding our approach to dynamic behaviour adaptation to context. The chapter also describes the improved syntax of Ambience.

Chapter 4 starts by presenting a general architecture for context-aware systems, describing the different subsystems and their responsibilities. This architecture helps us situate our work with respect to other parts of a full-fledged context-aware system. The chapter then presents our approach to context adaptation by showing the language constructs in Ambience that are specifically designed to this end.

Chapter 5 presents AmOS, our implementation of the core object model underlying Ambience. The chapter includes a thorough discussion of the object model from a language implementer’s or meta-programmer’s perspective, meaning that the most fundamental mechanisms are exposed as first-class constructs. The support of context-oriented programming is revisited from this perspective, and the chapter introduces a few new mechanisms not described in previous chapters.
Chapter 6 defines a formal semantics for the computation model implemented by AmOS (and thus underlying Ambience). The small-step operational semantics is defined as a context-sensitive term rewriting system. The main body of the chapter is devoted to presenting the reduction rules that specify the meaning of each primitive operation in the model. The chapter finishes by explaining the differences between the defined formal semantics and its realisation through AmOS.

Chapter 7 contains three case studies showing the applicability of the approach. The chapter lays out a validation matrix that closely follows the thesis claims stated in section 1.3, and shows which points are covered by which case. The case studies contain however more value than merely validating the thesis: they provide insight on the pragmatics of programming with dynamic adaptation in mind —an added value that can not be transmitted by the simpler running examples shown elsewhere in the dissertation. We believe that only by going over these examples will the reader get full grasp of the possibilities opened by the approach. Each case study illustrates different idioms and ways of exploiting contexts.

Chapter 8 concludes the dissertation by restating the contributions in a fine-grained and more technical manner, complementary to the general description already given in section 1.5. The chapter then makes a comparative discussion of related work, mentions the limitations of our approach and draws the dissertation to a close by suggesting research avenues that are open for future exploration.

Reading Aids

- To aid expeditious reading, an attention symbol on the margin demarcates the most important definitions and discussions, such as the one next to this phrase. We have attempted to place as few marks as possible throughout the dissertation.

- To visually unclutter the text and thus aid fluent reading, we place on the margin the cross-references to other sections which would normally be embedded in the running text (as parenthetical remarks). Each margin reference indicates a section pertaining to the point being discussed, such as the one next to this phrase.
2 Background

In this dissertation we set out to develop programming technology that makes context-aware dynamic behaviour adaptation easier to express than what is currently possible. In this chapter we examine several existing programming language concepts that could be exploited to achieve our objectives, discussing their advantages and disadvantages. Some of these concepts have inspired us in developing our own approach.

When confronted with the vast spectrum of possibilities to achieve dynamic behaviour adaptation, we decided to restrict our research scope, and hence the review of existing technology presented in this chapter, in three main regards. Firstly, we take a language engineering approach. For the reasons mentioned in section 1.2, this path seems more promising than exploring (for instance) configurable middleware, design patterns and plug-in mechanisms. Secondly, within the language engineering arena, we have a bias towards dynamicity in general. The possibilities of a computation model should go hand in hand with the inherent dynamicity of its target domain, namely ambient computing: peer-to-peer open networks of potentially unknown hosts that interact constantly, aiming at furnishing services whose functional and non-functional requirements may vary according to the changing context in which those services are used. Ambient computing triggers the need for complex, highly dynamic interaction among software elements. Our preference for dynamicity means that we are more interested in techniques that avoid hard-coding software interaction patterns. As a consequence, we concentrate on dynamically typed languages. Thirdly, within the dynamic language engineering bounds, we have chosen object-based programming [119] as pivotal point for our research. The family of object-based paradigms has members such as the actor-based, prototype-based and class-based paradigms. The least common denominator in the family is naturally the concept of object, which encompasses identity, encapsulated state and message passing. In the encapsulation metaphor, state is “covered by a capsule” —an interface. Two objects differing in their internal representation but having a common interface look the same from the outside, and are thus interchangeable or polymorphic. Object-based programming, in this broad sense, is well suited to the programming of dynamically adaptable applications for the following reason:
Objects help abstracting semantically coherent software units by merging state and behaviour — the two essential ingredients in computer chemistry. Objects confer to this mix a well-defined identity which allows different software entities to communicate. Further, objects are easy to think about, and thanks to encapsulation and polymorphism are prepared for highly dynamic environments where emphasis is put on unanticipated interaction.

In summary, we consider dynamism, polymorphism, encapsulation and identity to be fundamental properties for the development of applications that can adapt dynamically to changing contexts. Even though they are exhibited by many object-based languages, note that the mentioned properties do not preclude the integration of functional and logic programming abstractions.

Dynamic behaviour adaptation to context can only be achieved with techniques that permit flexible selection of behaviour at run time. However, any such technique would be rather ineffective if there were not much choice to make. Therefore, it is also important to be able to describe many behavioural pieces and be able to combine them in flexible ways, so as to obtain many different useful behaviours that can be chosen flexibly. According to this reasoning, and following the language engineering approach mentioned previously, this chapter concentrates on language abstractions that allow flexible behaviour selection and flexible behaviour composition.

2.1. Flexible Behaviour Selection

Programming technology for adaptability must support the description of more than one possible behaviour for a given situation — if there is no choice, there cannot possibly be adaptability. A second necessary property is that the selection of those described behaviours must be late bound — adaptation cannot happen a priori.  

Hence, dynamic behaviour selection is a cornerstone of adaptability.

Dynamic behaviour selection is one of the flagships of object orientation, usually referred to as late binding. In this section we review existing techniques that foster late binding, taking it beyond the barriers of singledispatch observed in most object-oriented languages.

2.1.1. Subject-Oriented Programming

The subjective school of object-oriented programming started by Harrison and Ossher [64] relaxes the emphasis on the object, and recognises more the binding concept of identity. State and behaviour are associated with objects from specific points of view or perspectives and can vary according to the current perspective. Object identity is an invariant property that ties the multiple subjective views together.

Harrison and Ossher make a distinction between the intrinsic and extrinsic properties of objects. The coordinates detected by a Global Positioning

---

1 If software “adaptation” takes place at deployment time, it is rather called configuration.
System (GPS) for instance are intrinsic to the operation of the GPS—they are its raison d’être. The market price of the device, on the contrary, is an extrinsic property that might be interesting only from the point of view of a reseller. The authors point out that the ideal classical model of object-oriented programming is inadequate to deal with the evolution of complex applications, because programmers are forced to anticipate all future uses of application objects, treating all extrinsic information as though it were intrinsic to the nature of objects. From a pure software engineering point of view, foreseeing and bundling together all possible extrinsic information into objects is questionable. System designers would need to manage ever-expanding collections of extrinsic attributes and behaviour becoming part of a same object.

Observing the limitations of classical object-oriented programming, Harrison and Ossher propose a general definition of Subject-Oriented Programming (SOP). The authors show how class hierarchies can be combined by different perspectives or subjects on the system as a whole, and describe how access can be granted or denied to instance variables and methods alike, keeping their discussion at a fairly language independent level. The authors point out the issues of new object initialisation and the richness of perspective combination strategies.

Conceptually, a subject is a collection of state and behaviour reflecting a particular gestalt, a perception of the world at large. The subject reflects a smaller, more focused perception of a complex shared model. Subjects generally describe some of the state and behaviour of many objects, rather than describing only one particular object or class. Technically, a subject corresponds to a traditional, though usually incomplete, class hierarchy, describing the interfaces and classes known to this subject. A subject does not itself contain any state. There is no global concept of class: each subject contains class descriptions of state and behaviour from that subject’s point of view.

A subject activation provides an executing instance of a subject, including the actual data manipulated by a particular subject. All code in a subject-oriented framework executes in the context of a particular subject activation. An operation call can therefore be modelled as a tuple \((a, op, p)\) where \(a\) is the subject activation making the call, \(op\) identifies the operation to be performed, \(p\) is a list of parameters. When an operation is invoked in a subject-oriented model, it might cause execution of methods in multiple subjects. Composition rules specify in detail how methods from different subjects for the same operation and class ought to be combined. There is freedom to craft and use different composition rules.

Discussion In the subject-oriented model there is no special status accorded to intrinsic properties. Intrinsic and extrinsic properties share a same status and can be implemented using similar techniques. The point is that extrinsic properties need not be mingled and be visible from perspectives in which they are irrelevant. The essential characteristic of subject-oriented programming is that different subjects can separately define and operate upon shared objects,
without any subject needing to know the details associated with those objects by other subjects. Only object identity is necessarily shared [64]. With their notion of subjects, Harrison and Ossher were effectively shaping one of the first techniques for advanced separation of concerns. The notion of separation of concerns is later on made more explicit by the proponents of Aspect-Oriented Programming (AOP) [71]. With the advent of AOP, research on subject-oriented programming took the form of multi-dimensional separation of concerns [114, 113]. With this change of direction, subject-oriented programming was not developed further so as to consider dynamic adaptation to changing perspectives (also, dynamic behaviour adaptation was of less relevance in the early nineties, without the hardware phenomena we observe today).

Although technically our approach is quite different form that of Harrison and Ossher, conceptually SOP has been inspirational to us. The mere idea of subjective object behaviour, behaviour that depends partially on the perspective from which objects are observed, is one small step away from the idea that behaviour exhibited by a system depends on the context in which that system is used. Further, the model of operations calls as tuples \((a, op, p)\), where \(a\) represents a subjective element that influences behaviour selection, suggests the way subjective behaviour can be implemented, and is generic enough that it can be used in many kinds of computation models, even classless ones.

### 2.1.2. Subjective Objects

Smith and Ungar [101] point out that the description of domain phenomena is intrinsically relative to the observer. As a consequence, there is a mismatch between the objectivity of most programming languages and the subjectivity of typical problem domains. This mismatch forces the creation of ad hoc mechanisms every time programmers come across problems that include multiple perspectives. These problems are usually tackled laboriously by implementing custom solutions, even though they could all be recast in terms of observer dependence. This duplication of work and complexity suggests that it would be profitable to support subjectivity as a fundamental principle of object-oriented computation models.

Smith and Ungar propose Us as proof-of-concept implementation built on top of Self [115] to explore subjectivity in object-oriented languages. In this subjective model, there is no single “true” state and behaviour for an object: rather, the state and behaviour of an object depend on a perspective which is reified as an object. This object can delegate to (at most) one parent object. Each individual object in the delegation chain thus formed is called a layer, and the whole layer chain is called a perspective. Hence, a reference to a layer serves also as a reference to a perspective.\(^2\)

Object identity is invariant under change of perspective. If an object exists in any perspective, it will exist in every perspective—it “exists” because the

---

\(^2\)The “layer” and “perspective” terms are used somewhat interchangeably. The latter corresponds to a black box view in which the internal delegation structure is opaque, whereas the former emphasises the existence of such structure.
2.1. Flexible Behaviour Selection

Object can be referred to from any perspective, and existing references will not vanish with changes of perspective. If two references refer to the same object in any perspective, they will do so in every perspective; if two references refer to different objects in one perspective, they must refer to different objects in every perspective.

Objects are divided into pieces, which are collections of methods and attributes. Since objects can be seen from any perspective, it follows that each object has exactly one (possibly empty) piece on each layer. The attributes within a piece in one layer can mask (override) attributes of pieces in parent layers. Cloning an object implies shallowly cloning its pieces in all layers.

Us allows methods and slots to be visible only from certain perspectives. The following transfer:To: method invokes two other methods from a pvtPersp perspective:

```plaintext
transfer: amt To: acctOrPerson = (  
  addAndRecord: amt negated.  
  acctOrPerson addAndRecord: amt.  
)⊗ pvtPersp
```

The notation `exp1 ⊗ exp2` means that the messages sent in expression `exp1` are to be evaluated from the perspective found by evaluating expression `exp2`. Layer activation using the `⊗` operator follows a stack-like discipline.

Discussion  Us is the first language to propose an abstraction for change of perspective, namely the `⊗` operator, in 1996. The few other languages that propose similar abstractions —ContextL [29], Slate [93] and our own— date from no earlier than 2005. Given the many advantages brought by subjectivity [101, 64], it is intriguing to see that the field remained dormant for more than 10 years, and is regaining attention only now.

As in Subject-Oriented Programming [64], subjective objects relax the emphasis on objects and recognise more the binding nature of identity. Variable behaviour is organised around invariable identities. Whilst behaviour is dynamic and may change according to perspective, identities provide stable reference points that allow communication among software components.

Aiming at minimal conceptual load, Us draws a symmetry between the role of the message receiver and the role of the message sender’s perspective. By extending message dispatch to include the perspective as well as the receiver, Us is effectively doing a kind of double dispatch on the perspective and message receiver.

The designers of Us are forced to separate the layer inheritance chain from the regular object inheritance graph, so that messages sent to layers when they are used as regular receiver objects are not confused with messages understood by layers when they are used as layer objects. Unfortunately, this duplicity of inheritance mechanisms on layers breaks the principle of minimality and simplicity the authors were after.

Layer combinations in Us are formed as chains of delegating layers. Behaviour
Background

is delegated from the foremost layer down to the last layer in the delegation chain. In Us there are no means of specifying behaviour that is specific to more than one layer at the same time.\(^3\) This severely restricts the kind of scenarios in which subjective behaviour can be exploited.

Despite the technical limitations of the early realisation of subjective behaviour by Smith and Ungar, the fundamental observation made by the authors remains valid, and is deeply connected to dynamic behaviour adaptability: an approach which allows the encoding of multiple behaviours for a same message, depending on the perspective from which the message is sent, is better suited to model naturally occurring systems than an objective approach. A conclusive argument in this regard is that objective behaviour is a particular case of subjective behaviour [101].

2.1.3. Context-Oriented Programming

The term *Context-Oriented Programming* has been coined in at least three different occasions for techniques aimed at supporting context-aware application development from a language engineering standpoint. We now describe these three techniques.

Gassanenko (1993 and 1998)

Gassanenko first coined the term Context-Oriented Programming (COP) referring to an approach that adds object-oriented programming concepts to Forth [56, 57]. Unfortunately, the author takes a rather technical view on COP (with a strong focus on Forth), obfuscating the concepts behind his approach. Gassanenko reports on his experience about successfully applying COP to remove conditional statements from application logic.

In this approach to COP, contexts are sets of related functions.\(^4\) Different contexts may provide different definitions for the same function. Contexts can be interchanged by other contexts that implement exactly the same interface. This way, late binding of functions is achieved. Hence, at the heart of this approach to COP there is a dynamic scoping mechanism of functions. Interestingly, dynamically scoped functions are the precursor of more recent approaches to COP, including the one proposed by Costanza et al. [30] as well as our own.

Contexts are partitioned in equivalence classes by interface. Contexts belonging to the same equivalence class share exactly the same interface, and contexts belonging to different classes do not have any function name in common. Hence, every possible message can be mapped to exactly one equivalence class. For each such class, there is a *current context* pointer telling which of the members of the class is to be used to dispatch the message. Since such pointers are global,

\(^3\) The use of single layer inheritance is probably one of the root causes.

\(^4\) Gassanenko points out that the closest match to contexts in COP are the objects of prototype-based languages such as Self [115]. Since contexts define behaviour but contain no attributes, it seems to us that contexts are akin more particularly to Self’s *traits*.
there is no way to nest context activations, although the author does foresee a possible way to implement a stack-like discipline for context activation.

Imposing separate equivalence classes without any overlapping function name whatsoever is a strong restriction to make in practise. The positive side of such strong simplification is that equivalence classes need no composition techniques. In other approaches in which interfaces can overlap, conflicts might arise, and these need to be handled manually (as in traits [95]) or automatically (as in mixins [15], ContextL layers [29] and Ambience contexts [59]).

Keays and Rakotonirainy (2003)

Keays and Rakotonirainy [68] use the term Context-Oriented Programming (COP) for a new method of programming which incorporates context as a first-class construct of a programming language. The motivation for the development of this approach is similar to ours, namely ubiquitous computing and pervasive computing. The authors describe a proof-of-concept implementation in Python and XML.

In this approach programs are skeletons with open terms that can be filled in with stubs. The context of the executing host is considered during context-filling to select the most appropriate stub for a given open term. The approach is thus oriented by syntax, with open terms being a sort of dynamic macro whose expansion is chosen according to context.

Open terms are meant to be placeholders for any syntactic entity, for example operands and operators in arithmetic expressions, receiver, selector and parameters in message sends, and even types in typecast expressions. In their implementation Keays and Rakotonirainy support only whole statements. As they stand, open terms are roughly equivalent to polymorphic call sites, allowing for the behaviour of whole statements to depend on context. Granularity is lower than in approaches in which every message is polymorphic according to context. Adaptation points must be foreseen by interspersing open terms throughout application code. Although in their general proposal of COP Keays and Rakotonirainy mention conceptual operations for context combination and reuse such as disjunction, conjunction, subtraction, negation and comparisons, none of such techniques are described.

Costanza and Hirschfeld (2005)

Costanza and Hirschfeld [29] proposed Context-Oriented Programming (COP) as a new programming approach to ease the development of context-aware applications by proposing dedicated language abstractions. The goal of COP is to avoid having to spread context-dependent behaviour throughout a program. Context-Oriented Programming brings a similar degree of dynamicity to the notion of behavioural variations that object-oriented programming brought to ad-hoc polymorphism [65].

ContextL, an extension to the Common Lisp Object System (CLOS) is one of the first programming language extensions that explicitly supports COP [29].
ContextL provides means to associate partial class and method definitions with *layers* and to activate and deactivate such layers in the control flow of a running program. Layers consist of only a name and no further properties of their own. They are the basic construct that enables grouping class and method definitions. Layered classes are thus formed, having partial definitions in different layers. When a layer is activated, the partial definitions become part of the program until this layer is deactivated. This has the effect that the behaviour of a program can be modified according to the context of use without the need to mention such context dependencies in the affected base program.

Layers can be activated in the dynamic scope of a program thanks to the following construct:

\[
\texttt{(with-active-layers (layer1 layer2 ...)} \\
\texttt{... contained code ... )}
\]

Dynamically scoped layer activation obeys a stack-like discipline, and is confined to the currently running thread.

**Discussion**

ContextL comes close to meeting the language characteristics we were after for the support of dynamic behaviour adaptation. However, we decided to stick to our development of Ambience for the following reasons.

Firstly, we wanted a prototype-based foundation instead of a class-based one. There are various accounts of the general advantages of prototype-based programming [14, 74, 73, 115, 52, 18, 85]. Furthermore, we wanted to make a design that is easily amenable to support concurrency and distribution, by keeping it in line with the computation models of the prototype-based languages E [80, 81] and AmbientTalk [36]. De Meuter [34], Dedecker [35] and Van Cutsem [117] discuss the advantages of prototypes to dissolve the illusion of distribution transparency between instances and their classes that needs to be maintained in class-based languages.

Secondly, we wanted a pure object-oriented model, in which all computation occurs in response to messages. This not only gives us the finest possible level of granularity for behavioural adaptations—all computational steps are adaptable—but more importantly, it spares the programmer from foreseeing the interactions that need to be encoded as layered (adaptable) methods instead of plain (non-adaptable) CLOS methods and functions, as must be done in the case of ContextL.

Thirdly, we wanted to have open contexts, whose structure is known to programmers and can be manipulated. This is in keeping with the concreteness and malleability of objects in prototype-based programming [74, 102]. The approach in ContextL is to provide a clear Application Programming Interface (API) for manipulation of layers, but the structure of layers (including combined layers) is opaque.

Fourthly, we wanted to have a reflective system with support of first-class environments. CLOS—and consequently, ContextL—does not support first-class environments. Having first-class, open access to environments is particularly
important when experimenting with dynamic scoping, precisely to allow experimentation through direct manipulation of such scopes. With more advanced reflection facilities in this regard, more advanced language semantics experiments become possible.

Fifthly, we did not want to include the concept of CLOS generic functions in our language design. In CLOS, methods belong to generic functions, which are central repositories for methods that comply with a predefined contract. Even though they are a good tool to establish well-defined contracts, it seems to us that generic functions confer ambient authority [81], because a whole set of methods is accessible by means of the function’s name, which is global. We wanted behaviour to belong to objects, rather than be detached from objects, in order to preserve the encapsulation metaphor —the idea that an object knows how to behave, and that such behaviour is internal to the object, as opposed to behaviour that is held by an external entity.

2.1.4. Predicate Dispatch

*Predicate dispatch* [48] is a mechanism for determining the method implementation to be invoked upon a message send. With predicate dispatch, each method implementation includes a predicate guard specifying the conditions under which the method is applicable. Applicability is given by an arbitrary predicate over the method’s formal arguments, formed from conjunctions, disjunctions, and negations of descends-from tests (method specialisers), equal-to tests (specialisers), and arbitrary boolean-valued expressions. Logical implication of predicates determines the method overriding relation. A method $m_1$ overrides another method $m_2$ if the predicate of $m_1$ logically implies the predicate of $m_2$ (this relationship is computed at compile time). Static type checking verifies that, for all possible combinations of arguments, there is always a single most-specific applicable method.

Ernst et al. [48] provide a number of examples illustrating how predicate dispatch unifies and generalises several common forms of dynamic dispatch, including traditional object-oriented dispatch, multimethod dispatch, and functional-style pattern matching. They also formally define predicate evaluation and provide a static type system that ensures that method lookup cannot fail. The validity of predicates can be tested with a conservative algorithm, which is necessary both for computing the method overriding relation and for static type checking.

**Discussion**  Predicate dispatch is among the most general behaviour selection techniques there can possibly be, because the selection criterion is left unspecified: it must be given by the user in the form of a predicate. Predicate dispatch subsumes multiple dispatch (and therefore traditional single dispatch), predicate objects and classifiers, and pattern matching. For example, under this model a traditional multimethod is simply a method whose predicate is a conjunction of “descends-from” tests of the method arguments. Thanks to
its generality, predicate dispatch could be used to support advanced forms of
dynamic behaviour selection for COP. At the same time, predicate dispatch pre-
serves several desirable properties of the approaches it subsumes, including that
methods can be declared in any order and new methods (possibly overriding
previous ones) can be added without modifying existing methods or clients.

There exist a number of systems that provide predicate dispatch [20]. They
differ in details of their design, but they all follow basically the same approach
for resolving ambiguities when comparing predicates: the set of predicates that
can be used for the purpose of method dispatch is restricted to a well-chosen
subset which is not Turing complete and can thus be statically analysed. This
leads to a viable approach, but can be limiting in some circumstances, since
users cannot easily extend predicate dispatch with their own arbitrary predi-
cates [31].

As Costanza et al. [31] point out, the generalisation of predicate dispatch
to allow arbitrary predicates has its limitations because predicate implications
cannot be decided in general. Consider a predicate prime which tests for prime
numbers, odd for odd numbers and even for even numbers. If methods are
defined for each one of these predicates, it is not possible to determine which
method is the “most specific” one, because prime numbers are not a subset of
even numbers or odd numbers, and vice versa.

In face of these limitations, Costanza et al. [31] came up recently with filtered
dispatch, an approach which sticks to inheritance for definition of specialisation
relationships between objects, but in which the arguments used for behaviour
selection can be different from the actual arguments used for behaviour invo-
cation.

This concludes our overview of language concepts that are related to be-
behavioural selection. The next section describes flexible mechanisms for be-
behaviour description and composition.

2.2. Flexible Behaviour Composition

An adaptable system must support the description of varied behaviours that
are applicable to different situations. Whereas overall system behaviour should
remain constant, detailed behaviour may be adjusted according to context [40].
For instance, a messaging application always delivers messages, but it may
deliver messages differently in different contexts. Programming technology for
adaptability must support dynamic composition of behaviour,\(^{5}\) so that detailed
behaviour can be combined with overall behaviour to achieve adaptation of
applications at run time.

Behaviour composition is one of the flagships of object orientation, and is
usually realised through various forms of inheritance [112]. In this section we

\(^{5}\) We are interested only on open systems, in which adaptations can be designed indepen-
dently, and be installed on software that is already deployed, rather than preconfigured
systems in which all adaptation scenarios have been foreseen and fixed in advance.
review the few forms of inheritance that are interesting from a dynamic behaviour adaptation perspective, as well as other composition techniques aimed at replacing or complementing inheritance. The versatility of behavioural adaptation—the ability to cope with different kinds of contexts—is in direct correspondence with the flexibility of available software composition mechanisms.

2.2. Flexible Behaviour Composition

2.2.1. Dynamic Inheritance

Most prototype-based languages feature a delegation mechanism\(^6\) [74], known also as object-based inheritance. The sharing role that inheritance plays in a class-based language is taken over by delegation in prototype-based languages. The idea is that one object can identify another object to be its parent through a parent link in such a way that every message that cannot be handled by the object itself is automatically delegated to the parent object.

Most languages with delegation support dynamic inheritance [116, 22, 112].\(^7\) In class-based systems, the references between children and parents are usually maintained by the system and are inaccessible to the programmer. In contrast, in prototype-based systems these references (the parent links mentioned previously) are typically reified as parent slots, which can be manipulated freely as if they were plain slots. It is thus possible to change parent objects on the fly, merely by assigning new values to parent slots.

Many delegation-based inheritance schemes have been devised in prototype-based languages. We briefly mention the most notable examples next.

Multiple inheritance

Self [115] features multiple inheritance by allowing an object to have more than one parent slot. Dynamic inheritance has been used in Self to implement mutable objects with several distinct behavior modes, such as binary trees with empty and non-empty states [116]. Cecil [18] and Slate [93] are inspired on the multiple inheritance scheme of Self.

The aforementioned languages serve also as example of different resolution strategies for the ambiguities introduced by multiple inheritance. The designers of Self proposed a prioritised inheritance mechanism [22]. Cecil rules out ambiguities by detecting them at compile-time. This is possible in Cecil thanks to static inheritance. Slate linearises the delegation graph by means of a depth-first method lookup mechanism.\(^8\)

\(^6\)A notable exception being the Kevo language [111, 110]. Kevo objects are conceptually stand-alone and typically have no shared properties with each other. Instead of supporting shared behaviour through delegation, Kevo provides a mechanism called propagation allowing the application of operations such as method addition to all objects of the same cloning family.

\(^7\)A notable exception being the Cecil language [18]. Cecil sacrifices dynamicity to enable compile-time optimisations. To compensate for this loss it supports predicate objects, discussed in section 2.2.2.

\(^8\)Io also features multiple dynamic delegation-based inheritance and depth-first lookup, see http://www.iolanguage.com. Both Slate and Io are modern languages in which medium-scale programs have been written. There still seems to be a place for this kind of inheri-
Differential inheritance
NewtonScript [103] introduced differential inheritance, also known as comb-inheritance, because the hierarchy looks like a comb. The design is aimed at reducing the memory footprint of programs.\(^9\) Objects in NewtonScript consist of two inheritance chains. One chain is the chain of data objects linked with parent links in RAM. Every part of this chain has a proto link and these proto-parts also form a chain (by their respective parent pointers) that resides in ROM. Hence, the parent is the “data parent” and the proto is the “behavioural parent” [34]. Messages are delegated (recursively) to the proto parent, and if not understood, they are delegated (again recursively) to the parent of the object.

Single inheritance
JavaScript [46] features single inheritance. JavaScript objects have one assignable, predefined prototype slot. Single inheritance trades expressiveness (rich object composition) for simplicity (impossibility of ambiguous messages).

Discussion  Dynamic inheritance permits flexible software composition. Object protocols can change at run time simply by modifying the object’s inheritance graph. From an external, black box perspective, objects exhibit dynamic behaviour: the set of supported messages may vary over time, as well as the behaviour exhibited for each supported message.

Dynamic behavioural changes come in handy for adaptation to different states or conditions. The behaviour of objects typically varies considerably depending on these logical states. For instance, the behaviour of the pop operation of a stack depends essentially on whether the stack contains items or is empty; in the latter case pop must not return any value but rather raise an error.

One common way of capturing different behavioural modes is to include a flag instance variable defining the behaviour mode, and testing the flag at the beginning of each method that depends on the behaviour mode. This obscures the code for each behaviour mode, merging all behaviour modes into shared methods that are sprinkled with conditional statements. Furthermore, it is hard to add new behaviour modes without modifying existing code, and it is difficult to understand a particular mode since its code is intermixed with code for other behavior modes. More advanced techniques, such as the State design pattern [55] are a symptom of to the rigidity of static inheritance. Conditional statements and design patterns have a negative impact on the maintainability of code [112].

Behaviour modes are naturally implemented in classless languages by using dynamic inheritance to choose from a small set of parents. This style of programming does not compromise the structure of the system; on the contrary,

\(^9\)The design was heavily influenced by the concerns of the target platform: palmtop computers developed by Apple in 1993, running with less than 640 Kib of RAM [34].
it can make the structure and organisation of the system clearer by separating out the various modes of behaviour [116]. Thanks to dynamic inheritance, method definitions can be more concise and clear, because the methods corresponding to one mode do not have to know how the other modes behave. In the stack example mentioned previously, a stack object can delegate to one of three prototypes depending on its state: empty-stack, full-stack or stack; the delegation link is switched dynamically among the three prototypes according to this state. The pop and top methods of empty-stack throw an error, as well as the push method of full-stack. The prototypical stack encodes regular behaviour—not empty, not full. The empty-stack and full-stack prototypes delegate all non-overridden behaviour to stack.

In contrast, the close coupling between a class and its representation prevent class-based languages from being extended naturally to handle behaviour modes. Dynamic inheritance is typically impossible in class-based systems, mainly due to the fact that classes describe not only the behaviour but also the structure of objects. If the superclass of a class were suddenly changed at run time, the structure of the instance would likely be incompatible with the structure specified by the new superclass [112]. As a consequence, most class-based languages feature static inheritance.

Static inheritance imposes a degree of structural rigidity on the system that hinders dynamic evolution. It does not allow the kind of dynamic system reorganisation that prototype-based architectures permit. Quoting Foote [52],

> it is essential that the structure of the system be able to evolve in such a way that it matches that of the problem itself. (Form must continue to follow function.)

From this perspective, dynamic inheritance seems an adequate abstraction to support adaptable system behaviour.

Unfortunately, powerful inheritance schemes such as dynamic inheritance and multiple inheritance are difficult to use effectively. In the case of dynamic inheritance, if the properties of a new parent do not comply with those needed by the descendant, run-time binding errors will result. In the case of multiple inheritance, automatically resolution of ambiguities between multiply-inherited conflicting behaviour can confuse users with unexpected but “correct” behaviour. The power of inheritance must be balanced against its complexity and its potential to confound intuition [22].

### 2.2.2. Predicate Objects

Predicate objects [18] (or predicate classes in [19]) are a relatively language independent idea that can be seen as a form of structured dynamic inheritance. Predicate objects support an automatic form of classification into specialised subclasses based on the run-time state of instances. Since this state can be mutable, the classification of an object can change over time.\(^{10}\) This mechanism

\(^{10}\)This notion of dynamic classification is similar to the subjective class hierarchies of subject-oriented programming [64].
Background

enables inheritance and classification to be applied even when modelling time-varying properties of an object. For example, a rectangle can be automatically classified as the predicate subobject square whenever the rectangle satisfies the predicate that its length equals its width, even if the length and width of the rectangle are mutable. Chambers [19] provides several other examples of the usefulness of predicate objects.

Predicate objects are like normal objects except that they have an associated predicate expression. The semantics of a predicate object is that if an object inherits from the parents of the predicate object and also the predicate expression is true when evaluated on the child object, then the child is considered to also inherit from the predicate object in addition to its explicitly-declared parents. Since methods can be associated with predicate objects, and since predicate expressions can test the value or state of a candidate object, predicate objects allow a form of state-based dynamic classification of objects.

Semantically, predicate expressions are evaluated lazily as part of method lookup, rather than eagerly as the state of an object changes. Only when the value of some predicate expression is needed to determine the outcome of method lookup is the predicate evaluated. Since the state of an object can change over time, the results of predicate expressions evaluated on the object can change. If this happens, the system will automatically reclassify the object, recomputing its implicit inheritance links. Predicate expressions are expected to be pure functions, so that the exact time of evaluation of predicate expressions is unimportant.

Discussion  Object-oriented languages support one kind of dynamic binding of messages to methods, where the method to run can depend on the run-time class or type of the message receiver (for singly-dispatched languages) or for some subset of the message arguments (for multiply-dispatched languages). Some object-oriented languages (including prototype-based ones), such as Slate [93], CLOS [13], Cecil [17] and Dylan [8] can dispatch on the identity of an argument, but cannot easily dispatch on a more general condition of an argument. In the absence of predicate objects, a method whose behaviour depended on the state of an argument object would include an if statement to identify and branch to the appropriate case. Predicate objects support associating state and behaviour with possibly time-varying behaviour modes of an object. Dynamic inheritance can do the same, but predicate objects eliminate the clutter of these tests and clearly separate the code for each case. Predicate objects are thus more declarative and modular.

On the downside, the realisation of predicate objects in Cecil is less powerful than dynamic inheritance. The set of potential predicate descendants of an object is statically determined at link-time, whereas in languages with dynamic

\footnote{Where a program using predicate objects would have an assignable field and a group of predicate objects whose predicates test the value of the field, a program using dynamic inheritance would have an assignable parent slot and a group of parent objects that could be swapped in and out of the parent slot.}
inheritance any object, including objects generated at run time, can be used as delegates.

2.2.3. Traits

A trait [95, 41] is a set of methods that implement the behaviour it provides. A trait requires the set of methods that it uses but does not implement. In the original model, traits do not have state of their own, and can access state only by means of accessor methods. The stateless nature of traits brings about a number of limitations that have been addressed recently by introducing stateful traits [10]. Traits can be composed in arbitrary order. The composite entity has complete control over the composition, and can resolve conflicts explicitly.

Traits bear a superficial resemblance to mixins [82, 15], with several important differences. Several traits can be applied to a class in a single operation, whereas mixins must be applied incrementally. Trait composition is unordered, thus avoiding problems due to linearisation of mixins. Traits contain only methods, so state conflicts are avoided, but method conflicts may exist. A class is specified by composing a superclass with a set of traits and some glue methods. Glue methods are defined in the class and they connect the traits together—they implement required trait methods (possibly by accessing state), they adapt provided trait methods, and they resolve method conflicts.

Trait composition respects the following three rules:

- Methods defined in a class itself take precedence over methods provided by a trait.
- A non-overridden method in a trait has the same semantics as if it were implemented directly in the class (flattening property).
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

A conflict arises on the combination of two or more traits that provide different methods with the same signature. Conflicts are resolved by implementing a glue method at the level of the class, which overrides the conflicting methods, or by excluding a method from all but one trait. In addition traits allow method aliasing; this makes it possible for the programmer to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden.

Discussion  Traits help overcoming problems of single, multiple, and mixin-based inheritance [41], by not featuring an inheritance mechanism all together—rather, addition, exclusion and aliasing operators are defined for trait composition. Traits support the reuse of coherent groups of methods by otherwise independent classes.

Traits are conceived as a “manual” composition mechanism, meaning in particular that the programmer is in charge of dealing with arising conflicts either by implementing a glue method, or by excluding a method from all but
one trait. In assessing the use of traits as a model for dynamic composition geared towards Ambient Intelligence, manual interventions by programmers are out of the question — software that is already deployed and running on the device cannot stop working. Hence, one of the main flagships of traits, namely manual resolution of conflicts, is not an option in our setting. In contrast to traits, delegation is designed to support *dynamic* component adaptation [41]. This lead us to favour delegation as composition mechanism.

Traits could still be used if a dynamic trait composition mechanism were devised. This requires developing sound mechanisms to automatically resolve composition conflicts arising at run time. Automatic resolution of conflicts takes us back to the point we were with mixins and delegation-based inheritance, in which ambiguities are either solved automatically (although not always desirably) by inheritance linearisation mechanisms such as those of Flavors [82], CLOS [13] and Dylan [8], or ambiguities are signalled as is the case in Cecil [17]. Whether dynamic trait composition could be superior to delegation-based composition, is a research question that we might explore in the future.

### 2.2.4. Dynamic Aspect Weaving

Aspect-Oriented Programming (AOP) [71] allows the programmer to encapsulate concerns that cross-cut modularisation boundaries (e.g. classes) in a construct called an *aspect*. Aspects are designed to supplement the basic composition mechanisms provided by the host language. There are three main *binding times* for aspects: compile time, load time and run time. Dynamic aspect weaving occurs at run time.

Dynamic aspect weaving can be thought of as a tool for dynamic behaviour adaptation, since it allows for base application logic to change at run time (for instance, to be adapted to non-functional concerns such as security or low power computation). Dynamic aspects can be woven and unwoven according to context.

#### Dynamic aspects in PROSE

Popovici et al. [88, 89] propose a dynamic AOP technique for run-time adaptability of mobile systems. Their initial motivation is practically the same as ours, namely the need for dynamic behaviour adaptation to context raised by ubiquitous computing. Globally, their approach is of the same kind also, namely support for run-time software composition, although the means to achieve this goal are quite different. The authors developed PROSE, a platform based on Java which addresses dynamic AOP.

PROSE supports aspect objects, which contain one or several *crosscut objects*. A crosscut object defines an advice and describes the joinpoints where the advice should be executed. Joinpoints are detected by hooking PROSE to

---

12 There are kinds of conflicts that are not well handled by traits, a reason for introducing *freezable traits* [42] in languages that support public/private method visibility.
2.2. Flexible Behaviour Composition

the JVM Debug Interface (JVMDI).\(^{13}\) A rich set of events supported by the debugging interface can be intercepted, such as field accesses, catch and throw of exceptions, class loads and unloads, and breakpoints (we presume that support of method invocations goes without saying). The interception of these events allows for detection of joinpoints at run time and execution of corresponding advice. Considering the number of events that can be intercepted, the debugger interface is almost as powerful as behavioural reflection mechanisms \([89]\).\(^{14}\)

Discussion In a pure object-oriented model, every possible form of computation is performed by executing methods. This includes field accesses, exception handling, and class loading and unloading. Breakpoints and binding of formal to actual arguments are special cases that cannot be handled solely by intercepting method execution. Letting these special cases aside, we prefer to restrict our assessment of dynamic aspect weaving to the interception of method invocations, and assume that the rest follows from this basic mechanism.\(^{14}\)

Once reduced to the essentials, the advantages of AOP can be obtained by simpler means, namely dynamic scoping of functions \([26]\). Costanza and Hirschfeld \([29]\) integrated this idea into the CLOS object-oriented model, developing a form of dynamic scoping of methods. Whereas AOP has been developed as an external mechanism that can be laid on top of existing computation models (e.g. Java \([71]\), Smalltalk, Python), dynamic scoping must be integrated as part of the base computation model. Hence, for base computation models that cannot be changed, dynamic AOP can be a suitable option to obtain behaviour adaptation to context — at the expense of more conceptual and technical complexity. Computation models with intrinsic support for dynamic method scoping can support dynamic behaviour adaptation in a simpler way, as demonstrated by Costanza and Hirschfeld \([29]\).
3 The Ambience Programming Language

This chapter introduces Ambience, the programming language we have developed as our testbed to research language features geared towards ambient applications [59]. Ambience is a dynamically typed, prototype-based language with delegation-based multiple inheritance. Since every first-class program entity is an object, and all interaction among objects takes place through message passing, the model is purely object-based. As a frame of reference, all these features are shared by the Self language [115]. Additionally, Ambience features multimethods [54] and subjective dispatch [93].

Ambience aims at being a multiparadigm model that does not sacrifice simplicity and homogeneity for expressiveness and flexibility. This chapter seeks to demonstrate that from an end-user perspective. The chapter starts by describing the core concepts of the computation model in section 3.1. It then describes the syntax of Ambience in section 3.2. A short description of objects and delegation is given in section 3.3 before diving into messages and multimethods in section 3.4.

3.1. Core Concepts

This section starts by highlighting the main concepts behind Ambience. These concepts form the cornerstones of the object model, on which all the rest is based.

Objects Every first-class entity in Ambience is an object — that is, the model is purely object-based. The observable properties of objects are their identity, acquaintances and behaviour. Whereas identity is an immutable (defining) characteristic, acquaintances and behaviour can vary over time. The latter two thus constitute the state of an object.

Some objects in the system act as representative examples of domain entities, and are therefore called prototypes. However, prototypes do not have a special status in the language other than being meaningful exemplars [74].

Cloning New objects can be created by cloning existing ones. Cloned objects have a distinct, unique identity, but are initially equivalent to the cloned object in all other regards.
**Messages** Interaction among objects happens through message passing. A message is a request for interaction among the participants involved in the message. To this effect, each message has a *selector* object that identifies the desired interaction, and an argument list of objects that will take part in it. Messages are *symmetric*: there is no distinguished receiver for any given message. Rather, all participants are considered receivers.

**Delegation** Behaviour can be delegated from one object to another by placing a delegation link between them. Since objects can have multiple delegations, a directed graph of delegation links can be formed. Messages that are not understood by an object can be handled by one of the delegates in the delegation graph. When we refer to *inheritance* in this dissertation we mean such delegation-based inheritance. Cyclic delegations are supported, as explained in section 3.5.2 on page 53.

**Methods** Methods describe prototypical interactions among objects. Every method has a selector that identifies the particular interaction it implements, and a list of prototypical arguments that take part in the interaction. The method is said to be *specialised* on those particular arguments, and each prototypical argument is called an *argument specialiser*.

Rather than belonging to a single class as in Java or to a single generic function as in the Common Lisp Object System (CLOS), Ambience methods belong simultaneously to all their specialisers. In other words, method ownership is shared, both at a conceptual and technical level. Methods are thus *symmetric*, just like messages are.

**Method applicability** For any given message, a method is *applicable* if the selector and arguments of the message match the selector and prototypical arguments of the method. The selectors match if they have the same object identity. The arguments match if each message argument delegates in zero or more steps to the method specialiser in the same position. This is illustrated in section 3.4.2 on page 42.

**Method specificity** Due to multiple inheritance, and multiple dispatch, more than one method might be applicable for any given message. A notion of *specificity* is introduced to solve ambiguities, which defines a strict, total order relationship among applicable methods. Ambience therefore features *asymmetric dispatch* [25]. This topic is discussed thoroughly in section 3.5 on page 44.

These concepts are all there is to the basic computation model of Ambience. The least trivial part is message disambiguation. This topic is discussed in section 3.5.1 on page 44.

---

1. Given that the only relevant property of a selector is its identity, any object can be used as selector, although most often symbols are used.
2. Zero steps meaning that the actual and prototypical argument are exactly the same object.
3.2. Syntax overview

This section describes the syntax of Ambience intuitively,\(^3\) explaining the rationale behind its design. The syntax has been designed with readability in mind, drawing inspiration from Smalltalk.\(^4\) Ambience expressions can be read almost like plain English, for example:

```ambience
define: #smart-phone as: mobile-phone clone.
```

This expression contains three message sends and a literal symbol. Different expressions are separated by full stops (.)

Expressions are the basic syntactic constituent of Ambience. They are subdivided in three main categories: literals, messages and methods, explained in the following sections.

3.2.1. Literals

Literals are objects that can be created ex nihilo [85], simply by writing their textual representation in the program text. Such is the case of numbers, strings, symbols, code blocks and sequences. The syntactic form of these literal objects is explained next.

**Numbers**

Number literals are divided into the following categories:

- **Naturals** a string of 1 or more digits. Example: 42
- **Integers** a natural preceded by a plus or a minus sign. Examples: -42, +42.
- **Floats** an integer with a decimal point. Examples: -4.2, -.42

**Strings**

Strings are arbitrary sequences of characters delimited by single quotes. To make a single quote part of a string, it must be repeated. Examples:

```ambience
'this is a string'
' Ambience’s strings can contain single quotes and multiple lines'
```

**Symbols**

Symbols are objects that are associated to a name. As in Smalltalk, different occurrences of a symbol literal with the same name will always yield the exact same symbol object. Symbols are denoted by a sharp sign (#) followed by the name of the symbol:

\(^3\)For a formal definition of the syntax, refer to section B on page 201.

\(^4\)The syntax design is also heavily influenced by Slate [93] and Self [115]. The use of dashes instead of CamelCase to separate words comes from Lisp [98, 105].
Symbol names can contain all but a few special characters that would confuse the parser, like dots, commas and spaces. A completely arbitrary name that has any possible character can be given as a string:

`
' with spaces '
'with/p,u.n;c.t u:a-tion'
``

The single quotes that delimit the string are not part of the symbol name.

### Code blocks

Code blocks (or blocks for short) are anonymous closures. The literal notation is with square brackets surrounding the body to be executed upon invocation:

```
[ print: 'inside code block body' ]
```

Code blocks can have any number of arguments. Arguments are separated by commas, and a vertical bar separates the arguments from the body:

```
[ x, y | x + y ]
```

The value yielded by the last evaluated expression is returned as result of a code block invocation. In the previous example it will be the result of \( x + y \).

Last, code blocks can have any number of local variables, to be declared after the arguments:

```
[ x, y | m | m: x + y. m ]
```

The following example shows a block with no arguments and two local variables:

```
[ | m, n | m: 1. n: 2. m + n ]
```

Getter and setter methods are automatically defined for every block parameter and local variable. For instance, in the above examples methods \( x \) and \( m \) are implicitly defined as getters, and methods \( x: \) and \( m: \) as setters. Unlike Smalltalk, there is no way to access variables directly.

### Sequences

Sequences are ordered collections of objects. The literal notation of sequences is with curly braces and commas separating the elements:

```
{ a_1, a_2, \ldots, a_n }
```

For example:

---

5 A closure is a function that carries a reference to its enclosing lexical environment [90]. The body of a closure always is executed in its lexical environment, enriched with the actual/formal argument bindings.
{ 1, #a, [ x | x + 1 ], 'panoramix' }

There is no constraint on the type of objects contained in a sequence.

3.2.2. Messages

Message sending is the most fundamental operation in Ambience. Messages can be nullary, unary, semi-binary, binary or keyword, as explained in this section. The reason for having such an heterogeneous syntax for messages is readability, as in Smalltalk [58].

**Nullary messages**

Messages with no explicit arguments are called *nullary*. They are the simplest kind of message, with the following general form:

```
selector
```

Although a nullary message has no apparent arguments, the current activation\(^6\) is always passed implicitly, as is the case for all messages. This hidden parameter represents the current execution context of the sender. The following are examples of common nullary messages:

- `true`
- `false`
- `object`
- `code-block`
- `current-context`

Nullary messages are typically used to get information from the current context, in particular prototypical objects such as `true`, `object`, `string` and `code-block`.

Another common use of nullary messages is to access the parameters and temporary variables of code blocks:

```
[ x, y | x + y ]
```

Here, the `x` and `y` messages in the block body are nullary. These messages will result in the invocation of the getter methods that are automatically generated for code block parameters.

The only constraint on the selector of a nullary message is that it must contain at least one alphabetic character, or an underscore. Names such as `a`, `2a`, `a2` are all valid nullary selectors. Although uncommon in practise, mixing alphanumeric and operator characters in selectors is allowed: `a2+`, `_2`, `-a-`, `a+b`. This last example requires some explanation. If the intention were to sum `a` and `b` rather than to specify the 3-character selector `a+b`, then white space would need to be used to separate the operator from its operands: `a + b`.

---

\(^6\)Activations are discussed in section 5.2.2.
Unary messages

Unary messages are like nullary messages except that they have one explicit argument, in addition to the implicit argument that is always passed. The general form is as follows:

\[ \text{argument selector} \]

For example, the selector \texttt{arity} can be sent to a code block in order to obtain the number of arguments accepted by the block:

\[ [ x, y | x + y ] \text{ arity} \Rightarrow 2 \]

The following example combines a nullary message with a unary message:

\texttt{mobile-phone clone}

Here the \texttt{clone} unary message is sent to the result of evaluating \texttt{mobile-phone} as nullary message. A clone of the prototypical mobile phone object is obtained.

Binary messages

Binary messages have two explicit arguments (in addition to the implicit one every message has), and use infix notation. The general form is the following:

\[ \text{argument}_1 \text{ selector} \text{ argument}_2 \]

The selector must be composed of one or more of the following characters:

\[ + ! @ $ % & * = ? / \sim - < > : - _ \]

As in Smalltalk [58], the selector of a binary message cannot contain alphanumeric characters. If it does, it is interpreted by the parser as a unary message, with the left argument taken as the only argument of the message.

Many common arithmetic, bitwise and boolean operators are binary messages sent to their operands:

\[ 40 + 2 \Rightarrow 42 \]
\[ 4 << 1 \Rightarrow 8 \]
\[ \text{true} \lor\lor \text{false} \Rightarrow \text{false} \]
\[ \text{true} \lor\lor \text{false} \Rightarrow \text{true} \]

Like Smalltalk, but unlike most languages that support infix operator notation, binary messages in Ambience have no precedence differences based on their selector. Binary messages are always evaluated from left to right regardless of the particular message being sent. Thus, whereas in many languages the expression \( 1 + 2 \times 3 \) is interpreted as \( 1 + (2 \times 3) \), in Ambience it is rather \( (1 + 2) \times 3 \), since \( \times \) has the same priority than \( + \) has.

Semi-binary messages

A special kind of binary message with only one argument is permitted: the left-hand side argument can be omitted. The most common occurrence of semi-binary messages is the return operator:

\[ ^\times \]
This kind of message does not classify as unary because the selector is placed to the left of the argument, whereas in unary messages the selector is placed to the right. Furthermore, the selector is composed of operator characters exclusively, a distinguishing trait of binary selectors. Semi-binary messages are thus halfway between unary messages and binary messages. Unlike Smalltalk, in Ambience operators such as the return shown above are normal messages.

**Keyword messages**

Keyword messages have 1 or more explicit arguments — besides the usual implicit argument. Their distinguishing characteristic is that arguments are separated by *keywords*. Each keyword is a nullary selector ending with a colon. The general form is the following:

```
arg₁ keyword₁: arg₂ keyword₂: arg₃ ...
```

The message selector is formed by concatenating the keywords. For example, in the following message the selector is “*set:to:*”:

```
person set: #age to: 30
```

A second form of keyword message is possible, in which the message starts by a keyword rather than an argument:

```
keyword₁: arg₁ keyword₂: arg₂ ...
```

An example of this type of message is “*define:as:*”, used to define a slot in the current environment:

```
define: #pi as: 3.1416
define: #smart-phone as: mobile-phone clone.
```

Smalltalk does not support this kind of message since it requires all messages to be sent to an explicit receiver object, which syntactically must be placed before the first keyword of the message.

**Message precedence**

When an Ambience expression contains several messages, the order of evaluation is defined by giving precedence to each message type as follows:

1. Nullary messages.
2. Unary messages.
4. Binary messages.
5. Keyword messages.

All messages have left associativity, except for keyword messages which have no associativity — if they did, only messages with one keyword would be possible. Parentheses must be used to compose keyword messages:
define: #name as: ([ x | x reverse ] mapped-on: strings)

3.2.3. Methods

Method definitions consist of a method header and a method body. The header resembles the corresponding message sending syntax. Therefore method definitions, like messages, are also divided in nullary, unary, semi-binary, binary and keyword categories. The only difference is that argument specialisers can be specified, as will be shown further on. Two notes are in order:

1. An argument representing the current context is always prepended implicitly to the list of explicit arguments given in the method definition. This implicit argument is specialised on the context that is current when the method is defined. This is key to our approach and is elaborated further in section 4.2 and chapter 5.

2. Argument specialisers can be omitted. Omitting a specialiser is equivalent to specifying object as specialiser. Since everything in Ambience is an object, specialisation on the object prototype does not actually constrain the kind of arguments that will be accepted by the method.

Nullary methods

Nullary method definitions specify methods that can be invoked by means of nullary messages. Nullary method definitions have the following general form:

```
selector
[ body ]
```

Nullary methods have only one implicit argument, which is specialised on the context of definition of the method.

For example, a `time` method can be defined that returns the current system time upon invocation:

```
time
[ system clock time ]
```

Although an invocation of this method through a nullary message `time` looks like an attribute access, actually the nullary method will be invoked and respond to the message. Whether the effect of a nullary message is the invocation of a “user-defined” nullary method, or rather of a system-generated “getter” nullary method, is transparent to the user.

Unary methods

Unary method definitions specify methods that can be invoked by means of unary messages. The general form is as follows:

```
argument (specialiser) selector
[ body ]
```
3.2. Syntax overview

The following example shows the definition of the **value** method, used to evaluate a code block that receives no arguments:

```plaintext
block (code-block) value
[ invoke: block arguments: { } ].
```

The **value** method simply calls the **invoke:arguments:** method to invoke the block, passing an empty argument list. Once the **value** method has been defined, it is possible to evaluate a code block this way:

```plaintext
[ 1 + 2 ] value => 3
```

**Binary methods**

Binary method definitions specify methods that can be invoked by means of binary messages. The general form is as follows:

```plaintext
argument₁ (specialiser₁) selector argument₂ (specialiser₂)
[ body ]
```

The selector must be comprised of binary-selector characters only. As said previously, the specialiser can be omitted if the argument is specialised on **object** (the prototype of an object).

The following example shows the definition of two binary methods that together implement the behaviour of the boolean conjunction operator:

```plaintext
a (false) /\ b [ false ]. a /\ b [ b ].
```

Notice the omitted specialisers. The non-specialised arguments will accept any passed object as value. In this boolean system, anything which is not **false** is considered as true. Whenever `/\` is sent with **false** as first argument, the first implementation shown above will be more specific than the second, yielding **false** as result. If the first argument is not **false**, then the first method is not applicable, whereas the second method is; the latter returns its second argument.

**Semi-binary methods**  Like for semi-binary messages, binary methods with only one argument are permitted: the left-hand side argument can be omitted. A specialisation of the return operator for numbers can be defined as:

```plaintext
^ value (number)
[ print: value. resend. ]
```

The method prints the value to be returned and then invokes the overridden definition of the return operator by means of a **resend** call. The overridden definition will actually return the value to the caller. The **resend** method is like a **super** call in Java.
Keyword methods

Keyword method definitions specify methods that can be invoked by means of keyword messages. The general form is as follows:

\[
\text{arg}_1 (\text{spec}_1) \text{ keyword}_1: \text{arg}_2 (\text{spec}_2) \text{ keyword}_2: \text{arg}_3 (\text{spec}_3) \ldots \\
\text{[ body ]}
\]

The \text{value:} method, used to evaluate a code block of one argument (analogous to its homonym in Smalltalk [58]), illustrates this type of keyword method definition:

\[
\text{block (code-block) value: argument} \\
\text{[ invoke: block arguments: \{ argument \} ].}
\]

The method relies on the \text{invoke:arguments:} method that evaluates a block for an arbitrary list of arguments. A list with the sole element \text{argument} is passed.

Methods that start by a keyword rather than an argument are also possible:

\[
\text{keyword}_1: \text{arg}_1 (\text{spec}_1) \text{ keyword}_2: \text{arg}_2 (\text{spec}_2) \ldots \\
\text{[ body ]}
\]

The \text{define:as:} method illustrates this kind of keyword method definition:

\[
\text{define: name (symbol) as: value} \\
\text{[ define: name as: value in: current-context ].}
\]

The \text{define:as:} method relies on the more general \text{define:as:in:} keyword method.

Anonymous method arguments

The names of arguments that are not used in a method body can be omitted in the method definition header. If the name is omitted, the argument specialiser must be specified. The following example illustrates anonymous arguments:

\[
\text{if: (false) then: (code-block) else: block (code-block)} \\
\text{[ block value ]}.
\]

\[
\text{if: (object) then: block (code-block) else: (code-block)} \\
\text{[ block value ]}.
\]

In both method definitions, only one argument has a name: the \text{block} argument. By the way, the two definitions above suffice to define a classical \text{if:then:else:} conditional, assuming that every object that is not \text{false} is regarded as true. If the second \text{if:then:else:} method were specialised on \text{true} rather than on \text{object}, a two-valued (\text{true, false}) boolean system like the ones of Java and Scheme would be obtained.
3.2.4. Comments

Comments are surrounded by double quotes. They can span multiple lines:

"example
  of a
  comment"

Any character is permissible except double quotes (").

This concludes the explanation of Ambience’s syntax. The next sections explain the associated semantics.

3.3. Ambience Objects

Objects are a core feature of our computation model. In Ambience, every first-class entity —that is, every value that can possibly be involved in a computation— is an object. This makes Ambience a pure object-oriented language, in the best tradition of Smalltalk [58].

Like Self [115], Ambience does not have a built-in concept of class. Objects are self-sufficient entities that can function properly without a class: object state and behaviour are reunited in the same language abstraction. If needed, the semantics of classes can be implemented in Ambience —actually, on any classless object model that features delegation [74].

The roles that would otherwise be played by classes, such as object creation and sharing of behaviour, are substituted by alternative object-based mechanisms, as explained next.

3.3.1. Object Creation

Ambience features two object creation mechanisms.

**Cloning** New objects can be created by cloning existing ones. Cloning is performed by copying shallowly the slots of the original object into the new object.

Cloning is implemented by the clone primitive (a unary method). This primitive can be specialised if a more involved cloning semantics is needed. In particular, clone can be overridden for objects that should copy their parts in a deep fashion:

```plaintext
dolly (sheep) clone
[ | clone |
  clone: dolly clone.
  clone head: dolly head clone.
  clone legs: dolly legs clone.
  clone ]
```

Here, sheep are cloned together with their head and legs.
Ex nihilo  Ambience also features ex nihilo object creation [85] —that is, producing new objects from scratch, rather than depending on previously existing ones. By writing down the literal representation in the program text, new objects are created, as in the following examples:

5 'string' #symbol { 42, 1984 } [ x, y | x + y ]

In Ambience, literals are the only objects that can be created ex nihilo. All other objects are created by cloning.

3.3.2. Object Evolution

Objects can evolve after they have been created by addition and removal of slots at run time. The dynamic modification of object structure is a distinguishing characteristic of object-based languages. Ambience offers the following primitive methods:

object add-slot: name valued: value
  Creates a slot with the given name and value in object.

object add-anonymous-slot: value
  Creates a new unnamed slot initialised with the given value.

object remove-slot: name
  Removes from object the slot with the given name.

object remove-slot: index (natural)
  Removes from object the slot at the given index position. This method can be used to remove anonymous slots.

For most objects it makes sense to add named slots, for instance the “gender” and “age” of a person. However, some objects such as arrays have (usually many) anonymous slots. In this case slot names are unnecessary because no particular meaning is attached to each slot. Anonymous slots can be accessed by index (with a slot-at: call).

Not only the structure, but also the behaviour of objects can change over time. Objects in Ambience are thus said to be open. Open objects support the addition of new slots and methods after having been constructed, even after deployment on clients. They are analogous to open classes in class-based languages such as Smalltalk [58] and MultiJava [24]. Open objects fit very naturally in languages based on symmetric multimethods —methods whose ownership is shared, rather than retained by only one object or class.

3.3.3. Object Delegation

The term delegation was originally introduced by Lieberman [74] and implemented in prototype-based languages like Self [115].\(^7\) Delegation consists in

\(^7\)“Delegation” is often used in the literature in relation to design patterns [55]. This other concept of “delegation” is about invoking a collaborator’s method using plain message
automatically forwarding the messages that an object does not implement to other objects, called delegates. The lookup mechanism involved in this process is explained in section 3.4.

The delegates of an objects are kept in an array associated to that object. Since this array is a normal object, delegates can be added, modified and removed dynamically simply by manipulating the array with the primitives shown in section 3.3.2. Ambience thus features dynamic inheritance, as delegation links can be freely manipulated at run time.

3.4. Ambience Multimethods

Ambience borrows the Prototypes with Multiple Dispatch (PMD) computation model from Slate [93] and is also inspired by the similar object system of Cecil [17]. Multiple dispatch departs from the idea that messages are passed to a single distinct receiver. A more expressive form of message passing is obtained where all arguments participate in method lookup at run time. A method that takes advantage of such a multiple dispatch mechanism is called a multimethod.

3.4.1. Multimethod Specialisation

The definition of a multimethod specifies the kind of arguments for which the method is designed to work. To this end, each formal argument declaration is annotated with an argument specialiser. As an example consider two objects, one representing a prototypical mobile phone (mobile-phone) and another representing a prototypical phone call (phone-call). A method that handles incoming calls can be defined this way:


The method selector is receive:on:, the formal argument names are call and phone, and the phone-call and mobile-phone prototypes are the argument specialisers. The method implementation is written as a code block between square brackets. The multimethod as a whole is said to be specialised on phone-call and mobile-phone. It will advertise the call (for example, have the phone emit a ringtone) and enqueue the call in the incoming calls queue.

In Ambience, argument specialisers are plain objects, in contrast with the multimethods of class-based languages such as CLOS [13] and MultiJava [25], which use classes as argument specialisers. Hence, a multimethod can be specialised on any combination of particular objects at hand. For example, the
receive: on: method can be specialised on calls from user Bob arriving on Alice’s phone:

```small
receive : call (bobs-phone-call) on: phone (alice mobile-phone) [ "Specialised behaviour for calls from Bob to Alice." ... ]
```

This way, idiosyncratic behaviour is defined for the particular combinations of arguments that delegate to the specialisers of this method. Notice that argument specialisers between parentheses can be arbitrary expressions; in this case the second argument is specialised on the result of evaluating the expression `alice mobile-phone`.

### 3.4.2. Multimethod Applicability

Given a message with a selector and list of actual arguments, a multimethod is said to be **applicable** for that specific message if the following conditions are met:

1. the message selector matches the method selector, and
2. each method argument specialiser can be found in the delegation graph of the corresponding message argument; in other words, there must be a delegation path of any length (including 0) from each actual argument to the corresponding argument specialiser (see figure 3.1).

As an example, consider the `receive: on:` method shown previously, and suppose the following message is sent with a particular phone call from user Alice to the mobile phone of user Bob:

```small
receive: alices-call on: bobs-phone.
```

This message will only be applicable to the `receive: on:` method if objects `alices-call` and `bobs-phone` delegate to `phone-call` and `mobile-phone` respectively, which are the two argument specialisers of the method. Figure 3.1 illustrates the idea. Even though in the depicted situation there is a direct delegation link between the actual arguments and the argument specialisers, the method would still be applicable in case of indirect delegations (arbitrary delegation paths).

### 3.4.3. Multimethod Specificity

Given the criteria described in section 3.4.2, it is possible for multiple methods to match a call. Once the set of applicable methods for a message has been determined, one of the methods must be chosen for execution: preferably, the method that is the most “adequate” to handle the given message under the current circumstances, if such a method exists.\(^9\)

---

\(^9\)The notion of adequacy is domain specific and partially depends on run-time factors that cannot be fully anticipated at development time. The problem of adequacy of chosen
A very common approach to defining adequacy is to introduce the notion of specificity among methods. In Ambience in particular, methods which have shorter paths leading from actual arguments to argument specialisers are considered more specific than methods for which paths are longer. The intuitive justification is that the closer an object is by delegation form the specialiser, the more similar (semantically and behaviourally) the object will be to the specialiser.

Applicable methods can thus be partially ordered by the notion of specificity. If there happens to be one method that is more specific than all the others, it can be chosen for execution. If on the contrary there are two or more methods that are more specific than all the others, but are not comparable amongst themselves, the message is ambiguous with respect to currently defined system behaviour. The discussion of ambiguity resolution strategies is deferred to section 3.5.2.

As an illustration of possible differences in method specificity with respect to a message, consider an overridden version of receive:on: (originally presented in section 3.4.1) that is specialised on emergency calls:

  push: call in: phone incoming-calls ]

As seen in the method body, this implementation pushes the call in front of the incoming call queue so that it is processed immediately, rather than enqueueing the call at the end. Any received call that delegates to emergency-phone-call will trigger execution of this receive:on: method which will be considered more specific than the original version. Figure 3.2 depicts a situation with a patient-call object that delegates to emergency-phone-call, and a doctor's phone which is a normal mobile phone. For the particular combination of patient-call and drs-phone arguments, the latter version of the receive:on: method is more specific, since the path between patient-call

behaviour with respect to the run-time context is crucial to our approach and will be discussed in depth in chapter 4.
and emergency-phone-call is of length 1, while that between patient-call and phone-call is of length 2. Hence, the more specific version of the method (with the shortest paths) will be chosen and the emergency call will be treated with priority.

3.5. Ambience Messages

The first defining characteristic of pure object-based systems, as mentioned in section 3.3, is that every first-class entity is an object. The second defining characteristic is that all object interaction takes place through message passing—meaning that every possible computation is initiated by sending a message.

3.5.1. Ambiguous Messages

Given a message, it can be that two applicable methods are incomparable, that is, none of the two is more specific than the other, yet the two of them are applicable. In this case the message is said to be ambiguous with respect to the set of applicable methods currently defined in the system. There are two sources of ambiguity in Ambience, arising from two independent language features: multiple inheritance and multiple dispatch. This section discusses both types of ambiguity and possible techniques for resolving ambiguous messages.

Ambiguity raising from multiple dispatch

Any language supporting multiple dispatch must deal with ambiguity, even if the language has single inheritance. The following example illustrates a typical case in Ambience. Consider the following method that is part of the Bluetooth subsystem of a constructor’s mobile phone:
### 3.5. Ambience Messages

Given the delegation graph shown in figure 3.3, the following message is ambiguous:

**pair: carols-phone with: bobs-phone.**

It can be seen as the pairing of a Bluetooth phone and a generic Bluetooth device (in that order), since Bluetooth phones are in particular Bluetooth devices, thus invoking the first version of the method; but it can also be seen as the pairing of a generic Bluetooth device and a Bluetooth phone (in that order), which would result in the invocation of the second version. There is no *a priori* way of telling which version is more appropriate. In this specific example we know that both choices are equally adequate, but this involves semantic knowledge about the behaviour of the methods (namely, that Bluetooth pairing is commutative) which in the general case is impossible to deduce automatically, by means of code analysis for example. In the general case, behaviour can be different depending on the chosen method.

**Ambiguity raising from multiple inheritance**

Any language supporting multiple inheritance must deal with ambiguity, even if it is singly dispatched. This well-known source of ambiguity has been studied to a great extent in languages such as C++, Common Lisp and Dylan. The problem is that, for an object with two or more unrelated delegates, it is not clear which of them is more specific [8] —that is, they are incomparable. As a typical example, consider the delegation graph shown in figure 3.4, and suppose

![Diagram](image-url)
The Ambience Programming Language

Figure 3.4.: Smartphone delegation graph; the media player object is unrelated to the mobile phone object.

that the following unary methods are defined in the system:

```
3.2.3
device (acme-media-player) supported-features
[ "Return feature collection for ACME’s media player."
  { #play-mp3, #play-m4a, #play-mpg } ]
```

```
device (acme-mobile-phone) supported-features
[ "Return feature collection for ACME’s mobile phone."
  { #receive-call, #make-call } ]
```

These methods return a collection of identifiers telling the capabilities of the device. The `supported-features` message applied on `acme-smartphone` is ambiguous:

```
acme-smartphone supported-features
```

Should `acme-smartphone` return its features seen as a media player, or as a mobile phone? Neither of the two methods specialised on `acme-media-player` and `acme-mobile-phone` is more specific than the other.

### 3.5.2. Ambiguity Resolution Strategies

In face of ambiguities (due to multiple inheritance or multiple dispatch) three basic strategies can be taken:

**Explicit resolution** Define a refined method that covers the ambiguous case. This solution requires foreseeing possible ambiguities at design time.

**Implicit resolution** Extend the dispatch mechanism with additional rules that ensure a total order among applicable methods at run time. This way, there will always be a most-specific method that can be chosen.

**Exception handling** Raise a run-time exception when more than one method is applicable.

These options are compatible (i.e. not exclusive). For any ambiguity that has not been foreseen at design time and therefore has no explicit resolution, or that has been left up to automatic resolution purposely by the designers, an
exception can be raised,\textsuperscript{10} and have as default exception handler the implicit resolver. All three options are discussed further in the following.

Explicit resolution by method refinement

Both types of ambiguity, arising from multiple inheritance and multiple dispatch, can be solved by defining a method that is more specific than all existing ambiguous methods;\textsuperscript{11} the way this method handles the ambiguous situation is domain specific. The following two examples illustrate the technique of explicit resolution both for ambiguities caused by multiple dispatch and by multiple inheritance.

Resolution of multiple dispatch ambiguities

Ambiguity caused by multiple dispatch can be solved manually by defining a symmetrically specialised version of the method that overrides all asymmetrically specialised versions. For example, the solution to the ambiguity of the \texttt{pair:with:} methods shown in section 3.5.1 is to define a new method that has \texttt{acme-bluetooth-phone} as specialiser for both argument positions:

\begin{verbatim}
pair: a (acme-bluetooth-phone) with: b (acme-bluetooth-phone) [ "Pairing behaviour between two Bluetooth-enabled phones from ACME." ... ]
\end{verbatim}

Resolution of multiple inheritance ambiguities

Ambiguity caused by multiple inheritance can be solved manually by defining a method specialised on the object that inherits multiply from two or more sources defining the same method. In the case of the \texttt{supported-features} methods described in section 3.5.1, the solution is to define a method specialised on \texttt{acme-smartphone} that simply aggregates the features of the media player and mobile phone embedded in the smartphone, and returns the aggregation of features:

\begin{verbatim}
device (acme-smartphone) supported-features [ "Return aggregation of features from media player and mobile phone." 
  acme-media-player supported-features 
  append: acme-mobile-phone supported-features ]
\end{verbatim}

The solution is domain specific, meaning that for supported software features it suffices to aggregate the result of the overridden methods. For other domains, this solution could not make sense at all.

\textsuperscript{10}More precisely, a condition can be signalled, such that the stack is not unwound and therefore the current execution state is preserved.

\textsuperscript{11}An alternative to defining a new refined method would be to let the programmer explicitly choose one of the existing alternatives. This is the idea behind traits, discussed in section 2.2.3. To the extent of our knowledge, no traits-like mechanism has been proposed for a language with multimethods yet.
When can explicit resolution be used? In many cases ambiguities are obvious and can be spotted easily by the developer, specially if they are local to the application. For non-trivial cases, only good software engineering methodologies can help foreseeing the ambiguities that require manual intervention (involving domain knowledge) at development time.

Assistance for detection of ambiguities at development time is harder in a highly dynamic language such as Ambience than in languages with static inheritance or closed objects. In Ambience, the system can give warnings at development time about delegation graphs that cannot be linearised, or methods that might be ambiguous. However, these checks cannot possibly be conclusive. Since Ambience features dynamic inheritance and open objects, ambiguities due to multiple inheritance and multiple dispatch can be introduced (and removed) arbitrarily at run time. Hence, all warnings Ambience could possibly give at development time are indicative.

Implicit resolution using tiebreaker rules

In Ambience there can never be ambiguous messages. Ambiguity is ruled out by defining a total order among applicable methods. The total order comes from a set of dispatch rules due to Barrett et al. [8], that have been carefully crafted to increase the chances that the implicit choice of behaviour closely follows the intuition of the programmer. As a consequence, foreseen ambiguities will more often be left up to implicit resolution, and unforeseen ambiguities will more likely —but not surely— be solved in a coherent way, respecting the programmer’s intentions.

Solving ambiguities due to multiple dispatch

The kind of ambiguities explained in section 3.5.1 are solved by giving left-to-right precedence to arguments. Recall the otherwise ambiguous message suggested in that section:

\[ \text{pair: carols\text{-}phone with: bobs\text{-}phone.} \]

Having left-to-right precedence implies that the method specialised on Bluetooth phones on the first position is more specific than the method which has such specialiser on the second position (these methods are shown in section 3.5.1 and illustrated in figure 3.3).

The rationale behind left-to-right precedence is that earlier arguments are in some sense “more important” to the user than the more auxiliary arguments that usually come on the 3\textsuperscript{rd}, 4\textsuperscript{th} and latter positions of a message. The extreme case is seen in singly dispatched languages, where the leftmost argument is always the most important, having complete priority over all other arguments. Even if only the receiver affects method lookup at run time, the tendency to place semantically important arguments first is also observed in singly dispatched languages such as Java and C++. Multiply-dispatched languages such as Ambience, CLOS and Slate exploit this natural tendency\(^{12}\) and thus use left

\(^{12}\text{Probably more natural for people who write from left to right in their mother language.} \)
3.5. Ambience Messages

to right precedence during dispatch.

**Solving ambiguities due to multiple inheritance** A mechanism to solve contention among the different arguments of a message has just been explained. Even so, ambiguities can still arise for any individual argument —independently of the others— due to multiple inheritance (recall section 3.5.1). This kind of ambiguity can be solved by linearising delegation graphs during dispatch —that is, by topologically sorting the delegation graph of each argument. A complete (linear) order among delegates is thus obtained, thereby rendering every delegate of the argument either more specific or less specific than any other given delegate.

Ambience uses an adaptation of a particular topological ordering called C3, originally proposed by the designers of Dylan [8]. This ordering exhibits a number of characteristics that language designers have deemed desirable for linearisations. Two of these characteristics —monotonicity and Extended Precedence Graph compatibility— were advocated by Ducournau et al. [44], whereas the third one —observing local precedence order— is due to CLOS. The name “C3” is derived from compliance with these three properties. There is yet a fourth property —acceptability— to which C3 complies. These properties are explained next.

**Acceptability**

An acceptable linearisation is one in which only the topology of an object’s delegation graph determines the linearisation [44]. Acceptability is aimed at excluding some uninteresting linearisations, an extreme example being a linearisation based on a lexicographical ordering of delegation slot names. For such linearisation, the modification of a delegation slot name could produce behavioural changes in the object that owns the slot, and all other objects that delegate to the affected one. Letting aside such extreme cases, we see no a priori reason to preclude the influence of local information on the linearisation process: the state of objects being linearised could be taken into account in some way. This could give rise to a mechanism similar to predicate objects [18]. However, in this dissertation we stick to the definition of acceptability by Ducournau et al. [44].

**Monotonicity**

A monotonic linearisation is one in which every property inherited by an object comes from one of its direct delegates —that is, an inherited property cannot skip over all direct delegates and be inherited from some indirect delegate instead.

As trivial as this property might seem, it is not always observed in linearisations such as that of CLOS. Consider the delegation graph shown in figure 3.5. The linearisations of objects 2 and 4 in CLOS and C3 are equal, namely:

\[
\text{CLOS}(2) = \text{C3}(2) = (2, 3, 5, 6, 7) \\
\text{CLOS}(4) = \text{C3}(4) = (4, 5, 7)
\]
Figure 3.5.: Inheritance graph for which CLOS is not monotonic.

One would expect that the linearisation of object 1 (which delegates to objects 2 and 4) respects the order of the two linearisations just shown. Surprisingly, the linearisation given by CLOS is:

\[ \text{CLOS}(1) = (1 \ 2 \ 3 \ 6 \ 4 \ 5 \ 7) \]

The behaviour inherited from object 5 by the direct delegates 2 and 4 will not necessarily be exhibited by object 1, because object 6 might override it with different behaviour. The linearisation is therefore not monotonic.\(^\text{13}\)

The intuition by looking at the diagram in figure 3.5 is that object 5 should have precedence over object 6 in the linearisation of object 1, because of left to right precedence in multiple delegation, and because this would respect the linearisation of object 2 when seen in isolation. The C3 linearisation behaves as expected:

\[ \text{C3}(1) = (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7) \]

Monotonicity means that the linearisation of an object’s delegation graph must be an extension without reordering of the linearisations of all of its delegates. With a monotonic linearisation, the behaviour of an object is easier to understand when multiple inheritance is used, largely because such behaviour can be explained in terms of the object’s direct delegates [8]. Monotonicity plays a critical role by ensuring that the delegation mechanism behaves “naturally” relative to incremental specialisation of behaviour.

**Preservation of local precedence order**

The order of direct delegations is called the *local precedence order* of the

\[^{13}\text{Curiously, if object 3 is omitted from the graph in figure 3.5, so that 2 delegates directly to 5, then the relative order of 5 and 6 will be consistent with CLOS(2): 5 will be followed by 6. It is unintuitive that the removal of the intermediate node 3, which could very well be an implementation detail of object 2, results in such reorganisation of the linearisation.}\]
3.5. Ambience Messages

Figure 3.6.: Inheritance graph for which CLOS does not preserve extended precedence.

object. A linearisation that observes local precedence order is constrained to respect the order of direct delegates. In the previous example, suppose that the order of arrows in figure 3.5 when read from left to right depicts the order delegates; then object 1 delegates in first place to object 2 and in second place to object 4, and consequently, the former must precede the latter in the linearisation of object 1. The property is maintained in a deep fashion, thanks to monotonicity: in any valid linearisation of object 1, object 3 will precede object 6, respecting the local precedence order of object 2.

Extended precedence graph consistency

Consider the delegation graph shown in figure 3.6. The linearisations of objects 2 and 3 in CLOS and C3 are:

- CLOS(2) = C3(2) = (2 4 5 7)
- CLOS(3) = C3(3) = (3 4 6 7)

If monotonicity and local precedence order are respected, these two lists can be merged in two different ways to produce a linearisation of object 1; the first is given by CLOS, whereas the second is given by C3:

- CLOS(1) = (1 2 3 4 6 5 7)
- C3(1) = (1 2 3 4 5 6 7)

Since objects 5 and 6 are not directly comparable (imagine 1 was not part of the graph), their order can be swapped in the linearisation without breaking monotonicity and compliance with local precedence order. However, intuitively object 5 should have precedence over object 6, because 5 is reached through the first delegate of 1, whereas 6 is reached through the second delegate of 1 —the extended precedence order should be respected.

As has been shown, an order between objects 5 and 6 can be established by

\[\text{In CLOS the local precedence order corresponds to the order in which superclasses of a class are listed.}\]
Dealing with inconsistent delegation graphs

It is possible for a delegation graph to be inconsistent under a given linearisation mechanism. This means that the linearisation is over-constrained and thus does not exist for the given delegation topology [8]. In particular, C3 leaves room for incompatibilities.

Consider the graph depicted in figure 3.7(a), and suppose that the left-to-right order of outgoing arrows corresponds to the local precedence order of delegations. The order for object 2 is thus (4, 5), whereas for object 3 it is (5, 4). Hence, the linearisations of these objects are as follows:

\[ C3(2) = (2 4 5 6) \]
\[ C3(3) = (3 5 4 6) \]

Given the order inversion of 4 and 5 in the linearisations of its two direct delegates, object 1 cannot be linearised by C3, because monotonicity cannot possibly be respected.

Cycles are another source of inconsistency. A sample case is illustrated in figure 3.7(b). In general, a directed graph has a topological ordering if and only if it is acyclic. Hence, cycles not only prevent C3, but actually any compatible linearisation whatsoever.

For maximum flexibility, Ambience poses no constraints on the topology of
delegation graphs. To solve inconsistencies such as the ones shown in figure 3.7, we use an adapted version of C3 to topologically sort the delegation graph of an object. Our adaptation of C3, which we call C3*, resolves inconsistencies by observing the local precedence order of objects. In the example, the linearisation process of object 1 starts by determining the linearisations of its direct delegates 2 and 3 (shown previously). The two linearisations are subsequently merged to form the linearisation of object 1. In case of conflict (as in our running example), Ambience gives priority to the linearisation order of delegates occurring first in the local precedence order. Hence, the linearisation order of object 2 prevails over that of object 3. The final result will thus be:

\[
C3^*(1) = (1 \ 2 \ 3 \ 4 \ 5 \ 6)
\]

The local precedence order of object 2 has been respected, but to solve the inconsistency, the order of object 3 has not. The rationale is that 2 has more precedence than 3 has, since the former is an earlier delegate of 1 than the latter.

To solve inconsistencies due to cycles, Ambience implements a “first seen has precedence” policy—that is, it takes into account only the first occurrence of a delegate during the linearisation process and ignores any further occurrences arising from cycles. To linearise the sample cyclic graph shown in figure 3.7(b), Ambience starts by object 1, marking it as “seen”; it then goes on recursively linearising its delegates, namely object 2, then object 3 and finally object 4 (all of which are also marked as seen). At each step, Ambience discards the direct delegates that it has already seen. Hence, the linearisation of 4 will not contain 2—even if it is a direct delegate—because by the time a linearisation of 4 is attempted, 2 will have already been traversed. The intermediate and final linearisations are therefore:

\[
\begin{align*}
C3^*(4) & = (4) & \text{(after having visited 1, 2 and 3)} \\
C3^*(3) & = (3 \ 4) & \text{(after having visited 1 and 2)} \\
C3^*(2) & = (2 \ 3 \ 4) & \text{(after having visited 1)} \\
C3^*(1) & = (1 \ 2 \ 3 \ 4) & \text{(final answer)}
\end{align*}
\]

The linearisation of a cycle depends on the object through which the cycle is entered. The linearisation of object 5 in figure 3.7(b) will be:

\[
C3^*(5) = (5 \ 4 \ 2 \ 3)
\]

In general, one of the following linearisations will be embedded in the linearisation of any graph containing the cycle (as is the case for objects 1 and 5):

\[
\begin{align*}
C3^*(2) & = (2 \ 3 \ 4) \\
C3^*(3) & = (3 \ 4 \ 2) \\
C3^*(4) & = (4 \ 2 \ 3)
\end{align*}
\]

All linearisations of a cycle are pairwise incompatible (contradictory).

As a concluding remark, note that C3* behaves exactly like C3 for consistent delegation graphs. What we have done is define an automatic resolution strategy for inconsistent graphs. Such strategy is necessary in Ambience because ambiguities cannot always be detected at development time. Due to
dynamic inheritance, delegation graphs can change arbitrarily at run time, and chances for ambiguous cases are higher than in systems with static inheritance. We feel that the tiebreaker rules we propose follow the programmer’s intuition about which objects should be considered more specific than others in face of inconsistencies, much as the other rules proposed by the designers of C3 are designed to be intuitive for consistent cases.

**Exception handling**

The final mechanism that can help solving ambiguities is signalling a condition and eventually handling the situation in the best way possible for the current context. In Ambience, inconsistencies in the delegation graph of objects are signalled, giving the programmer an opportunity to deal with them, or otherwise let them be resolved automatically. When an inconsistency is detected, the run time will signal a *warning* condition.\(^{16}\) Unlike exceptions (i.e. *error* conditions), such warning conditions can be handled without unwinding the stack, thereby preserving the current execution state and allowing for normal resumption of execution at the point where the condition was signalled. For most cases, resolution strategies will be domain-specific.

Whereas ambiguities arising from multiple inheritance are signalled as described previously, multiple dispatch ambiguities are not. The implicit resolution strategy that gives left to right precedence to arguments is always used. Ambience does not signal a condition for efficiency reasons. Supporting such signals would imply determining whether there is more than one applicable method for the message being dispatched. This would impede an important optimisation in the current implementation of Ambience.\(^{17}\) By improving our implementation (namely, by putting a caching mechanisms in place), it is likely that signals for this kind of ambiguity could be supported without diminishing the benefits of said optimisation.

**Final remark on ambiguity resolution strategies**

The thorough discussion given throughout this section might give the impression that ambiguities are not easy to deal with in Ambience. Note however that a similar discussion applies to CLOS, a successful object model that has been used widely in different domains. In our experience with Ambience, the ambiguity resolution strategies described in this section have proved to be sufficient. What happens in practise is that applications do not contain ambiguities by design, and introduction of ambiguities at run time is avoided by manipulat-

---

\(^{16}\)This is done through the condition system of the underlying implementation in Common Lisp [98].

\(^{17}\)The dispatch mechanism can sometimes detect that the very first applicable method found is the most specific in the total ordering. It will therefore shortcut the complete method lookup process and return that method immediately. The effectiveness of this optimisation is considerable.
3.5. Ambience Messages

If needed, the most-specific method chosen for a given message can invoke the overridden behaviour of the next most-specific method by means of a so-called resend message. Invocation of overridden behaviour can continue in this way from more specific versions to less specific ones. A resend invocation in the least-specific method (the end of the applicable method chain) results in an error since there is no overridden behaviour that can be invoked.

As an example, consider the implementations of the receive:on: methods shown previously. These methods send the advertise:on: message to advertise the received call. The system can define two handlers for this message, depending on whether the phone is a plain one or a more sophisticated smartphone:

advertise: call (phone-call) on: phone (mobile-phone) [ play: phone ringtone on: phone speaker during: 15 seconds ]

advertise: call (phone-call) on: phone (smartphone) [ show: phone ringtone-animation on: phone display during: 15 seconds.
resend ]

Figure 3.8.: Message resend.

ing delegation links in simple and controlled ways. If one day the automatic strategies fall short of our expectations, the many benefits brought by multiple inheritance and multiple dispatch will make it worth to investigate more advanced resolution strategies.

3.5.3. Message Resends

If needed, the most-specific method chosen for a given message can invoke the overridden behaviour of the next most-specific method by means of a so-called resend message. Invocation of overridden behaviour can continue in this way from more specific versions to less specific ones. A resend invocation in the least-specific method (the end of the applicable method chain) results in an error since there is no overridden behaviour that can be invoked.

As an example, consider the implementations of the receive:on: methods shown previously. These methods send the advertise:on: message to advertise the received call. The system can define two handlers for this message, depending on whether the phone is a plain one or a more sophisticated smartphone:

advertise: call (phone-call) on: phone (mobile-phone) [ play: phone ringtone on: phone speaker during: 15 seconds ]

advertise: call (phone-call) on: phone (smartphone) [ show: phone ringtone-animation on: phone display during: 15 seconds.
resend ].

18 A drastic case of control could be the idiomatic use of single static inheritance and single polymorphism to emulate Java-like semantics, or no inheritance and no polymorphism as in C, if such extreme level of control becomes necessary. C++ style—multiple static inheritance and single polymorphism— can be used as well. Ambience, at least, gives the choice, which can be adjusted according to the problem at hand.

19 In particular, these features are at the heart of our approach to dynamic adaptation to context, as explained in chapter 4.

20 The term “resend” comes originally from Self [115].
The smartphone version starts showing an animation on the screen, and then invokes the default behaviour for plain mobile phones by means of a `resend` call. Since the plain behaviour plays a ringtone, the net effect is that the smartphone shows an animation and plays a ringtone when it receives a call.

As shown in figure 3.8, the `smartphone` prototype delegates to `mobile-phone`. If such delegation link were not present, the version of `advertise:on:` specialised on `smartphone` would not be overriding the one of `mobile-phone`; in this case the two methods would be independent, and the `resend` call would fail.
4 Context-Aware Programming for Ambient Intelligence

More than a decade ago, the combination of wireless networks and miniaturisation started buzzing people’s minds with the possibility of interconnecting heterogeneous computers and embedding them in the human habitat, so that they would assist human activities by interacting in the most intricate ways, yet as smoothly as possible to avoid disrupting those activities. Numerous applications of these systems have been envisioned through the years by the proponents of ubiquitous computing, calm technology [121, 120], pervasive computing [94], the disappearing computer [107], ambient intelligence [99, 67, 43] —the term we adopt in this dissertation— and more. The variety of terms in use are associated with particular institutions or perspectives, each with its own emphases and nuances, but all initiatives share similar goals.

The steady convergence towards this kind of ambient systems has brought new functional and technical challenges that were a non-issue upon the time desktop and server systems dominated the computing platform spectrum. One of the prime advantages of ambient systems is their availability in situ, right at the moment and place in which their computing services are most useful. Thanks to real-time availability of information coming from their physical and logical environment, ambient systems have the potential to adapt swiftly to changing running conditions. We lift this potential to a requirement: ambient systems should be aware of their execution context and should adapt dynamically and autonomously to such context so that they can provide a service that fulfils the user needs to the best extent possible.

At the software level, the platform shift from fixed systems such as desktops and servers to ambient systems such as Personal Digital Assistants (PDAs) and embedded devices equipped with sensors must be accompanied by a shift in software architectures from relatively fixed structures to interoperable and adaptable services. Full dynamic software adaptation to context thus becomes increasingly important in ambient systems: the capability of these systems to respond to changes that occur in their operating environment through the dynamic transformation and reconfiguration of their components and services. Context-aware dynamic software variability is therefore key to the construction of applications that are smart with respect to the user needs and adaptable to the current environment. We say that such applications are ambient. Ambient applications question the underlying assumption that a single application behaviour can be articulated and anticipated completely, and replace it with the view that application behaviour should be causally connected to its context and
so flexible as to gracefully accommodate the most varied circumstances [40].

4.1. General Context-Aware System Architecture

Ambient systems should be in particular context aware. Figure 4.1 shows the general architecture we envision for context-aware systems. Our computation model has been designed with this architecture in mind. The parts of the architecture pertaining specifically to our work are enclosed in a dashed box and set in bold type. Even though this dissertation concentrates on those parts, it is important to explain the global framework for which our approach has been conceived. This shows our particular mindset and overall scheme of things for engineering new language constructs. In this section we give an overview of the general framework and explain the responsibilities of each subsystem, so that in particular the scope of our work is clearly delimited and the interactions with the other parts of the framework are well understood.

Figure 4.1 serves as an index for the forthcoming discussion. From left to right, the context discovery subsystem is explained in section 4.1.1; the context management subsystem is described in section 4.1.2; the dashed box containing part of the context management subsystem, context representation, base
4.1. General Context-Aware System Architecture

![Diagram of context categories]

Figure 4.2.: Types of context information.

application logic and interactions among these is treated in the remainder of the chapter (and dissertation).

4.1.1. Context Discovery

The first main part of our proposed architecture for context-aware systems is the context acquisition or context discovery subsystem, which is responsible for the extraction, aggregation and deduction of contextual information [66, 92]. Such contextual information should be an accurate depiction of the surrounding environment at any moment in time. Besides the surroundings, contextual information also encompasses the internal environment of the device. Applications can adapt their behaviour according to this joint computational snapshot of the external world and internal state.

We distinguish two other kinds of contextual information, physical and logical, which are orthogonal to the internal/external distinction. Context information can be about internal physical properties of the device, or external physical properties of the environment. Analogously, the logic environment has internal and external components.

A last subdivision pertains the logical context only: we distinguish between semantic context and service context. Again, the semantic and service contexts can be either internal or external. The different context categories are illustrated in figure 4.2 and explained next.

Physical Context Discovery

The physical context is made up of information pertaining to the physical surroundings of the device. The kind of physical data a context-aware system is likely to obtain and process is exemplified by the following lists of typical sensor categories for ambient computing, inspired on Beigl et al. [9]. Some of these sensors are common in many modern consumer devices.

Internal physical information  Physical information can be gathered by internal sensors that measure the properties of components making up the device.
Such internal information is generally not shared knowledge and cannot be known by other peers unless communicated explicitly.

Signal strength – Electromagnetic signal strength of WiFi, Bluetooth, ZigBee and GPS radio transmitters.

Battery charge – Remaining charge in embedded power supplies.

Temperature – Temperature of internal components the device.

Humidity – Humidity inside the device.

Force – Force applied to various parts of the device including averaging measurements such as weight and atmospheric pressure.

Time – Relative time or wall time as measured by the device’s operating system or built-in clock.

**External physical information** This kind of information can be obtained from sensors embedded in the device or from distributed sensors\(^1\) that measure properties of the surrounding environment. The information is generally shared knowledge.

Temperature – Environment temperature.

Biometrics – Heartbeat and breathing rhythm, blood pressure.

Proximity – Detection of near(ing) objects and users.

Movement – Velocity, acceleration, rotation and vibration.

Location – Spatial position and orientation; altitude from ground level.

Humidity – Environment humidity.

Sound – Noise level, frequency spectrum.

Light – Level and change over time for various wavelengths like daylight or infrared light.

Time – Network time from Network Time Protocol (NTP) servers.

Raw measurements of time, temperature, battery charge or noise level might be directly usable by applications. In such cases physical context discovery is sufficient and no additional processing of the input data must be performed. Applications requiring higher-level information can rely on a semantic context discovery subsystem.

\(^1\)We think particularly of **wireless sensor networks** \([32]\) — networks of inexpensive, low-power sensing devices that can be deployed throughout a physical space, providing dense sensing close to physical phenomena.
4.1. General Context-Aware System Architecture

Semantic Context Discovery

For cases in which plain physical data is not meaningful to an application, a knowledge discovery module can process the data and assign it a meaning that is closer to the application’s required input. As depicted in figure 4.1, knowledge discovery is part of the context discovery system, and is responsible for acquiring the semantic context of the device. During knowledge discovery the raw measurements delivered by sensors are scanned, searching for patterns that can be considered knowledge about those measurements. As a simple example, out of raw spatial coordinates it could be deduced that an object is “being used” or “standing still”. Further, it can be discovered that a person is “running” by analysing her location, heartbeat rhythm and current velocity. In similar ways it can be discovered that a person is “working”, “talking”, “driving”, and so on. This information is at a higher semantic level than the plain data from which the knowledge was obtained.

Many knowledge discovery techniques can be applied to the stream of raw measurement data produced by the different sensors, although for most practical purposes —given the constraint that such techniques have to be applied in real time— simple methods that require little processing are most likely to be used in ambient systems. These fast methods detect simple situations such as “being in a silent environment”, “receiving strong sunlight”, “running with low battery charge”, and so forth. Devices with little computing power can use this simple context information and relay complex context acquisition needs (if any) to network peers with greater power. If needed, fat peers (e.g. fixed servers) could boast complex knowledge discovery architectures to provide higher-level information.\(^2\)

Service Context Discovery

Current wireless interconnection technology allows devices to roam across different networks. If fixed infrastructure is not available, the devices can form their own cooperative infrastructure without the aid of centralised support services.\(^3\) Hosts can freely enter and leave the open networks they encounter and engage in communication with other peers —whether fixed or mobile— that are part of the same network.

Open networks constitute highly interactive spaces with a changing offer and demand of services from each device connected to the network. The services offered by network peers make part of the logical environment of a device. We shall call this part of the logical environment the service context. The service context is composed of the remote services currently available in the network.\(^4\)

\(^2\)A wealth of research exists in the domains of pre-processing, feature extraction, feature selection and classification of gathered data.

\(^3\)Such decentralised infrastructure is known as a Mobile Ad-hoc Network (MANET) [6], and is sustained thanks to protocols such as IPv6 Stateless Address Autoconfiguration (RFC2462) and Multicast DNS (mDNS).

\(^4\)Existing service discovery protocols such as DNS Service Discovery (DNS-SD) can be used to detect service context changes.
New services are advertised and existing services are withdrawn as peers freely enter and leave the open networks they encounter. Applications that are able to interoperate with those services can adapt their behaviour opportunistically to offer improved functionality, and degrade gracefully when the services loose quality or vanish.

Some offered services can be about context discovery themselves. We call these context discovery services. For instance, a peer might offer a temperature measurement service from which other peers can know the current ambient temperature, rendering their physical context representation more complete; another device can offer semantic information about the activities people are carrying out in the room, thus helping other devices that do not feature semantic context discovery themselves. Nevertheless, not every possible service is necessarily a context discovery service: for example, a printing service does not tell anything about the client device and its surrounding environment (except perhaps for the knowledge that there is a printer somewhere nearby); the same is true of an on-line payment service.

4.1.2. Context Management

The second main component of the architecture depicted in figure 4.1 is the context manager, which is in charge of handling all incoming context information from the context discovery system, and manipulating the system’s context representation accordingly so as to obtain well-adapted behaviour. The context manager implements user and application adaptation policies [38], specifying the behaviour that should be exhibited under specific circumstances. Further, the context manager maintains behavioural consistency by avoiding sudden, erratic or contradictory behavioural changes.

The architecture we assume for the context manager is inspired on proposals by Mostinckx et al. [84] and Desmet et al. [38]. The context manager incorporates an inference engine that holds a set of context activation rules. They describe the conditions for which specific context changes should take place. These rules trigger the context switches that need to be performed to obtain behaviour that is well-suited to the current context and implements user and application adaptation policies. For example, the inference module of the context manager may include the following rules and associated actions:

- Noisy environment mode: if environment noise is over 80dB, activate the noisy context.
- Quiet place: if the current location is within a church, hospital or library, activate the quiet context.
- Low battery charge: if battery charge is under 10%, activate the low-power context.
- Calm situation: if the time is between 11pm and 7am, activate the calm context.
4.2. Ambience Contexts

The consistency manager module deals with situations in which rules are conflicting. For instance, during a philharmonic concert organised in a church, the first two rules could be applicable, triggering the activation of contradictory contexts.

The examples just given are but a glimpse of the kind of reasoning that a sophisticated context manager could perform by means of its inference engine, and of the kind of situations a consistency manager may need to solve. In general, rules define when and what adaptations should take place. The how is given by behaviour that is specifically tailored to each situation (noisy, low-power, and so on). These three axes of context adaptation —the when, the what and the how— are fundamental to any context-aware system, and all require the development of appropriate technology that can provide sound answers to the main issues they rise. This dissertation concentrates on the how. Given the outcome of the context manager in terms of decisions —that is, what contexts need to be activated and deactivated at each particular moment— we provide solutions on how to encode behaviour that is well suited to particular contexts of interest. We do not deal with discovery and inference of context information, nor do we address management of inconsistencies such as the one exemplified previously. Note however that we do provide solutions for one part of the context management module, namely the context aggregation box shown in figure 4.1.

4.1.3. Assumptions

In adopting the general system architecture described previously, we assume the existence of a context discovery engine and part of the context management engine. Ongoing research on context discovery techniques [39, 91, 23, 87] and inference engines for context-aware systems [84, 38] has already provided some answers and established the feasibility of a number of techniques for implementing those subsystems. Our assumptions are backed up by such research. The main motivation for our work is the lack of corresponding answers on the third important part of the architecture, namely appropriate programming technology for implementing context-aware application logic.

Having explained our general mindset and main assumptions about context-aware systems in detail, we devote the remainder of this chapter to explaining our proposed solution for writing dynamically adaptable and context-aware application code.

4.2. Ambience Contexts

Ambience is a language that facilitates writing context-aware base application logic. The Smalltalk-like syntax and Prototypes with Multiple Dispatch (PMD) semantics of Ambience are the subject of chapter 3, but an explanation of the context-oriented part of the language has been deferred to this chapter, so that a coherent view of context-aware programming in Ambience is given, starting
with the general system architecture explained in section 4.1, and continuing with the proposed language abstractions and context management techniques in the remainder of the chapter. This section explains our proposed context representation and the way it influences application behaviour. The section thus corresponds to the “context” and “application logic” boxes of figure 4.1 on page 58.

Our approach to context-aware behaviour adaptation can be regarded as an instance of context-oriented programming [65] and subjective programming [101]. The main idea is that object behaviour, exhibited in response to a message send, does not only depend on the message arguments, but also on the context from which the message is sent. That is, the point of view of the caller affects behaviour selection [64]. Hence, the behaviour that is exhibited by objects is intrinsically bound to the current (changing) circumstances in which they are used.

The dependency on the caller’s perspective is realised in Ambience by means of subjective dispatch, a mechanism originally found in the Slate programming language [93], which in turn draws inspiration from the notion of subjective objects by Smith and Ungar [101]. Technically, the point of view of the caller is reified as a plain object. Whereas related approaches call such object a subject or layer [29, 101, 93], we call it a context. The nature of context objects is explained next.

### 4.2.1. Context Representation

In Ambience, contexts are first-class. Contexts reify the physical and logical properties of the environment in which the system is running. These properties may be about the user, the machine, the surroundings or in general any information which is computationally accessible [65], be it acquired through sensor input, network communication, generated internally, or otherwise.

For every relevant situation there is an associated context object that represents such situation computationally. For instance, being inside a car can be associated to a car context; whether it is currently day or night can be represented by morning, afternoon and evening contexts; running with low battery charge can correspond to the low-power context already mentioned in section 4.1.2 on page 62. The current activities or state of the user can also be reified if needed by contexts such as working, programming, sleeping, and so on. In general, context objects are a reification of the perceived state of affairs both inside and outside the device, at the physical and logical levels.

A context object, as any normal object, can delegate part of its behaviour to other objects. These delegate objects can be seen as representing more general “supercontexts” of the original context (correspondingly, the more specific context is a “subcontext”). General context objects can delegate further to

---

5The general state of things; the combination of circumstances at a given time as perceived by the context discovery and context management subsystems explained in sections 4.1.1 and 4.1.2.
coarser-grained contexts as needed. Figure 4.3 shows a sample context configuration. There is a special current-context object that represents the current situation as a whole. This representation is subdivided by way of delegation in a number of domain-specific subcontexts. These contexts that are currently reachable in the delegation graph starting from the current context are said to be active. The current context thus serves as a handle to all currently active subcontexts. Note that by definition the current context is always active. Furthermore, it is the most specific context that can possibly be active at any given time. The reciprocal of the active status is of course inactive: any context that is not linked to the current context delegation graph.

In the situation depicted in figure 4.3, the device is currently being used in a meeting room to make a presentation in a quiet place, a situation that is represented computationally by having active meeting-room, presentation and quiet contexts. This way, relevant information on environment acoustics, user activities and kind of location is represented in the system. The dotted arrows in figure 4.3 show possible alternative subcontexts that can be activated or deactivated according to changes detected in the environment. The manipulation of delegation links at run-time gives rise to what we call dynamic context switching.

Before proceeding to explain dynamic context switching, note that figure 4.3 shows only the behavioural part of the context —that is, the delegation links. That does not mean that context objects are limited to storing delegation information exclusively. The context objects comprised in the graph can also contain plain slots with references to arbitrary contextual information needed by applications. Such slots account for the data-oriented part of the context. When we mention in the following the “context graph”, we refer to the delegation subgraph of the whole object reference graph.
4.2.2. Dynamic Context Switching

The dynamic nature of the context representation is crucial to our approach. The context graph topology is an instantaneous representation of the current perceived internal and external state of things, which might change on the fly as changes in the environment are detected. Continuing with the previous example, if people start talking during the presentation and an embedded microphone detects a high level of noise, the delegation link in figure 4.3 can be changed from quiet to noisy. Similarly, the delegation link to meeting-room can be switched to a.229-2nd-floor when the user finishes the presentation and returns to her office. Dynamic inheritance is thus exploited to adapt the context graph such that it reflects the current environment as timely and accurately as possible: context changes in the domain system are reflected in the computational system by delegation slot changes. We call each of these changes a context switch. A seemingly simple or unitary action such as moving from one room to another can give rise to many context switches in the context graph, each one reflecting a change at a different domain-specific level, such as variations in illumination, room acoustics, available services, and user activities.

As explained in section 4.1.2, the context manager is in charge of activating and deactivating contexts as needed. In our example, the generic meeting-room context is to be activated when a device enters such a space. Behaviour that is specific to meeting rooms will thus be applicable. Thanks to the delegation link from meeting-room to room, more general behaviour that is meant for rooms (whether they are meeting rooms, waiting rooms, offices, etc.) will be activated as well. Hence, through delegation, many context activations (and deactivations) can be caused by only one context switch.

4.2.3. Idiosyncratic Contexts

One of a kind contexts can be defined naturally. In the example of figure 4.3, the a.299-2nd-floor context represents one particular office. Whenever a device enters room a.299, context a.299-2nd-floor will be activated, giving rise to the idiosyncratic behaviour of that particular environment. By virtue of delegation, all behaviour that is typical of offices and more generally of rooms will be operational also. The natural support of singletons and their idiosyncratic behaviour is a known advantage of prototype-based programming [85], which we leverage for context-oriented programming. This kind of contexts arises for instance as a result of client configuration and customisation. If default behaviour deployed in a device is unsuitable to the user, for instance generic office behaviour, then her customisations can be installed in singletons, giving rise to contexts such as a.299-2nd-floor in the example.6 Since customisations

---

6Customisations could be specified via user settings in the configuration panel of a Graphical User Interface (GUI), or scripts written in a configuration Domain-Specific Language (DSL), for instance.
4.2. Ambience Contexts

in-context: quiet do: 
[ receive: call (phone-call) on: phone (mobile-phone) 
  [ activate: phone vibrator during: 10 seconds. 
    enqueue: call in: phone incoming-calls ] ]

Snippet 4.1: receive:on: adapted to quiet environments.

can be handled naturally within the model, we need not rely on additional mechanisms for their support.

4.2.4. Influence of Context on Object Behaviour

Ambience builds on the observation by Smith and Ungar [101] that any language with multiple dispatch can easily support subjective object behaviour. In Ambience, the current context is passed implicitly as the first argument of every message. Correspondingly, an extra formal argument is added implicitly to every method definition, using the current context as implicit argument specialiser. Methods are thus specialised on their context of definition. Therefore —following the multiple dispatch semantics explained in section 3.4— a method is applicable only when its context of definition is active. This implicit argument and the interplay with the multiple dispatch semantics constitute the core of subjective dispatch. Note that the underlying Prototypes with Multiple Dispatch model needs no modification in order to support subjective dispatch, other than the addition of an implicit context argument to messages and methods. The model thus keeps its original simplicity.

As an example of the subjective dispatch mechanism just described, consider again the method receive:on: introduced in section 3.4.1 on page 41. The method could be defined to behave differently depending on the acoustics of the current environment, by specialising it on quiet and noisy contexts. The first version, which avoids making noise, can be used in places such as libraries and situations such as meetings; it is shown in snippet 4.1. The in-context:do: call switches the current context to quiet and evaluates the passed code block within that context. Since the code block contains the receive:on: method definition, the defined method will be specialised on the current context, which will be the quiet context. The second version of receive:on:, shown in snippet 4.2, can be used in noisy places. This version of the method will be specialised on the noisy context. If at any given time current-context happens to delegate to quiet (i.e. if quiet is active), then the first version will be applicable. However, if the delegation is switched from quiet to noisy, then the second version of the method will come in force. If neither quiet nor noisy are active, the default version of the method will be used. As Salzman and Aldrich [93] point out,

 prototypes naturally support composition of subjects by delegation,

---

7Actually, it is the current method activation record that is passed, but the activation record delegates to the current-context object. This is explained in detail in chapter 5.
Context-Aware Programming for Ambient Intelligence

Snippet 4.2: \texttt{receive:on:} method adapted to noisy environments.

allowing for a sort of dynamic scoping of methods by merely linking contexts together with dynamic extent.

4.3. Context Management

The foundations of our approach to dynamic context-aware behaviour adaptation have been laid in the previous section. We model the context as an object graph that guides behaviour selection. Managing this object graph correctly is vital for behaviour coherence. This section describes the techniques we have built on top of the subjective approach to context adaptation to ease context management and maintain behaviour coherence. This section thus corresponds to the “context management” box, and more specifically to the “context aggregation” and “consistency manager” subsystems, shown in figure 4.1 on page 58.

Let us revisit the scenario from section 1.1 on page 2 to illustrate the main practical issues we have found in context management and our proposed solutions. The scenario is used as a running example throughout this section, and it serves also as a more advanced example of the approach to context-oriented programming introduced in previous sections. Recall that the CityMaps application from the scenario is about showing maps to the user. To draw the maps on the screen, the application contains the methods shown in snippet 4.3. The \texttt{city-maps} context represents the availability of the CityMaps application —it is activated when the application is launched, and deactivated on application shutdown or crash.

4.3.1. Framework Contexts

The methods defined in \texttt{city-maps} context are specific to CityMaps. Methods that are meant to be reused by many applications running in varied situations can be defined in what we call \texttt{framework contexts}, discussed next.

In the scenario, the availability of a Global Positioning System (GPS) service in the environment renders the CityMaps application more navigational. The availability of a GPS should change the behaviour of CityMaps such that the current geographical location is taken into account when the map is displayed. We thus need extension code that accounts for situations where CityMaps and a GPS are both active simultaneously. These applications can interoperate if they have a common ground that allows them to talk to each other, even if they have been developed independently. Shared vocabularies and behavioural
in-context: city-maps do:
[
  draw: map (map-section) on: display (canvas)
  [ "Draws map elements from background to foreground"
    draw: map background on: display.
    draw: map buildings on: display.
    draw: map streets on: display.
    draw: map highlights on: display.
    draw: map labels on: display ].

  draw: elements (collection) on: display (canvas)
  [ elements do: [ element | draw: element on: display ] ].

  draw: street (avenue) on: display (canvas)
  [ print: 'drawing avenue' ].

  draw: street (highway) on: display (canvas)
  [ print: 'drawing highway' ].

...]

Snippet 4.3: Map-drawing behaviour implemented in the draw:on: method.

in-context: gps do:
[
  define: #gps-locator as: object clone.

  locator (gps-locator) coordinates
  [ locator longitude paired-with: locator latitude ].
]

Snippet 4.4: Generic GPS framework code.

contracts are needed. In Ambience, framework contexts provide such common
ground.

The independently developed CityMaps application and GPS software from
ACME will be able to interoperate thanks to the standard GPS framework shown
in snippet 4.4. The coordinates method returns a pair (latitude, longitude)
with the current geographical location. This code is oblivious to the particular
provider of the longitude and latitude methods. These methods could be
provided by any vendor of GPS services, including implementations that relay
this physical context acquisition need\(^8\) to network peers.

In the scenario, the GPS service is provided by ACME’s hardware module for
smartphones. This vendor provides the methods shown in snippet 4.5 to query
the hardware and obtain geographical data. The latitude and longitude

\(^8\)Recall section 4.1.1 on the different kinds of context discovery we define.
define: #acme-gps as: object clone.
acme-gps add-delegation: gps.

in-context: acme-gps do:
[
  locator longitude
  [ "Returns a fake longitude for the sake of the example."
    print: 'reading longitude from hardware'.
    (random: 360) - 180 ].

  locator latitude
  [ "Returns a fake latitude for the sake of the example."
    print: 'reading latitude from hardware'.
    (random: 180) - 90 ].
]

Snippet 4.5: Methods to query ACME’s GPS module.

methods are specific to situations in which the GPS service from ACME is active (i.e. when the GPS module is attached to the smartphone). The vendor-specific acme-gps context delegates to the more general gps framework context. Hence, the generic coordinates method is inherited. When ACME’s hardware module is connected to the smartphone, its detection will trigger the activation of the acme-gps context. As a consequence of delegation, the gps context will also become active. Note that the generic gps context should never be activated on its own. Doing so would render the generic coordinates method applicable, which would invoke undefined latitude and longitude methods. As illustrated by this example, framework contexts must be used as delegates of more specific contexts in the current context graph. Only frameworks providing suitable default behaviour for their hook methods should be used directly.

4.3.2. Context Combinations

In the scenario, the availability of a GPS service renders the CityMaps application more navigational. We have shown the independent code of CityMaps on the one hand and of a generic GPS framework and ACME’s customisation of that framework on the other hand. The three parts have been conceived separately. Now we need glue code that prescribes their interaction. Such interaction is not specific to the CityMaps application alone (i.e. to the city-maps context), nor is it to a GPS alone (i.e. to the gps or acme-gps contexts). A combined context is needed, in which to define the cooperation.

Snippet 4.6 shows the needed extension code that accounts for situations where both the CityMaps application and a GPS are active simultaneously in the smartphone. This is expressed by passing a list of contexts {city-maps, gps} to in-context:do: instead of only one context. When the in-context:do: method receives the list of contexts, it creates a new context object that deleg-
4.3. Context Management

izes both to city-maps and to gps. The new context represents the combination of the two original contexts, and is therefore called a context combination. The code block containing the definition of the draw:on: method will be evaluated in this newly combined context. Therefore, this version of draw:on: will be specific to that particular combination—that is, it will be applicable only to situations in which both city-maps and gps are active.

The new draw:on: method changes the behaviour of CityMaps such that the current geographical location is taken into account when the map is displayed. The coordinates message sent to locator on line 4 reads the current geographical location from ACME’s hardware module. Once the map section has been relocated according to the GPS coordinates just read, the resend message on line 5 invokes the original version of draw:on: (like a super call in Java), which implements the default map-drawing behaviour shown previously. The last two lines draw the graphical representation of the user at her current location.

The net effect of the adaptation code in snippet 4.6 is that the location shown in the map will correspond to the current geographical location, and the user will be represented graphically at the centre and on top of streets, buildings, and other map elements. CityMaps will thus have become navigational, as was intended in the scenario.

Uniqueness of context combinations

At all times, there can be at most one context object representing the combination of a given set of component subcontexts. For instance, the combination of \{city-maps, gps\} always results in the same combined context object that delegates to city-maps and to gps. If this were not the case, that is, if a new context object delegating to city-maps and to gps were created each time it were needed, then the methods specialised on the first instance of the context combination would not be visible (applicable) on the second or any new subsequent instances that would be created, despite the fact that conceptually they represent the same combination. Conceptually there is only one \{city-maps, gps\} combination, and computationally this must also be the case. Furthermore, the order in which contexts are combined is irrelevant.

In summary, there is a one-to-one correspondence between sets of component
subcontexts \{c_1, \ldots, c_n\} and the context object \( c \) representing the combination. On a practical level, this uniqueness property implies that created combination objects need to be memoised by Ambience’s context management system,\(^9\) so that these very objects can be retrieved when the same combination is needed. The identity and behaviour of context combinations is preserved this way.

**Delegation among combined contexts**

Besides the uniqueness of context combinations, Ambience’s context manager must also take care of maintaining delegation relationships among combined contexts. Combined contexts that are more specific than other existing combinations must delegate to those less specific (or equivalently, more general) combinations. In the previous example, the \{sunlight, city-maps, noisy, acme-gps\} combination context should have a delegation link to the \{sunlight, city-maps, noisy\} combination context, since the former corresponds to a superset of the latter. The delegation link makes sense conceptually because supercombinations, as we call them, are more specific (contain more information about the environment) than subcombinations. The least-specific combinations are those of only one subcontext. In these cases, a new object that represents the combination is not created; rather, the sole subcontext is taken as representation of the combination. For example, the combination of the set \{city-maps\} is the context \( \text{city-maps} \) itself. If two combinations are not comparable (neither is more specific), then no delegation link is established between them.

In determining the specificity of a combination (whether it is a supercombination or a subcombination of another one) it is not sufficient to examine its delegation links shallowly. Suppose that the current combination is \{sunlight, city-maps, noisy, acme-gps\} —as said before, this could be the case if the CityMaps application with the GPS module is being used on the street. This combination should delegate to the combination of \{city-maps, gps\}; if it did not, the specialised \texttt{draw:on:} method in snippet 4.6 would not take effect, and the user would see no difference at all with respect to plain CityMaps behaviour. The delegation link is thus needed. However, the set \{sunlight, city-maps, noisy, acme-gps\} is not a superset of \{city-maps, gps\}, since the former lacks element \( \text{gps} \). Nonetheless, by going one step further in the delegation hierarchy, we observe that \( \text{acme-gps} \) delegates to the missing \( \text{gps} \). Hence, the first combination actually is more specific than the second, implying that a delegation link should be established between them. We conclude that the notion of specificity of context combinations cannot be based solely on the immediate components that make up the combination: the delegations of each subcontext must be taken into account.

The example just explained gives the intuition behind the following definition of specificity, which dictates the delegation relationships among context combinations. The \textit{specificity} order among combinations is induced by the proper

\(^9\)More specifically, by the context aggregation subsystem, recall figure 4.1.
subset relation $\subset$ between combination linearisations. Given two combinations $c_1$, $c_2$ we define $c_1 < c_2 \iff C^3_*(c_1) \subset C^3_*(c_2)$.\(^{10}\) When $c_1 < c_2$, $c_1$ is said to be less specific than $c_2$. Since the specificity order is induced by $\subset$, it is strict and partial. As mentioned previously, delegation links are placed only between comparable combinations—incomparable combinations are not related through delegation. More precisely, the link goes from $c_2$ to $c_1$, from more specific contexts to less specific ones. Even though specificity is a transitive relation (as any order relation is), the delegations among combinations that the run-time system places are never transitive. If $c_1 < c_2$ and $c_2 < c_3$, it is the case that $c_1 < c_3$, but there will be no direct delegation link between $c_3$ and $c_1$. Such link would leap over the intermediate links from $c_3$ to $c_2$ and $c_2$ to $c_1$, which in the best case would be redundant, and in the worse case could result in the behaviour of $c_2$ not being expressed by $c_3$. Even the best case is undesirable.

**Implicit combination of contexts**

The invocations of `in-context:do:` that pass a literal list such as `{ city-maps, gps }` in snippet 4.6 are not the only points at which contexts are combined. Actually, this kind of explicit combination—the only kind shown so far—is scarce. Most context combinations are performed implicitly by the system and on the fly, as environment changes are detected. Whenever the context manager deems the activation of a context necessary, the context is combined with the active contexts that make up the current combination. For example, if the current combination is `{ sunlight, city-maps, noisy }` (e.g. the user is using the CityMaps application on the street) and `acme-gps` is activated (e.g. the user plugs in the GPS hardware module), then the system will create a new combination `{ sunlight, city-maps, noisy, acme-gps }`, and such combination will become the current context.

4.3.3. Consistent Context Activation

Sections 4.3.1 and 4.3.2 discuss the techniques we use to support context aggregation. This is one of the main responsibilities of the context management subsystem depicted in figure 4.1. Another important responsibility (also depicted in the figure) is consistency management, which is about avoiding the activation and deactivation of contexts at inappropriate times. Arbitrary context changes could break the running system or give rise to unexpected behaviour. Next we discuss two techniques aimed at maintaining consistency.

---

\(^{10}\)The C3* linearisation is defined in section 6.5 on page 132. C3* yields sequences instead of sets, but these sequences can be seen as sets in the definition of specificity, and thus be compared with usual set operators. C3* never returns a sequence with repeated elements, but even if it did, it would not matter that these repeated elements “collapse” into one single element when the sequence is seen as unordered set—we need not work with multisets.
Concurrent context manipulation support

Context switches take place dynamically, as changes are detected in the surrounding environment. As a consequence, context switches occur concurrently, at the same time applications run on the device. Not all points in execution are safe to perform those context switches without affecting the behavioural consistency of the system. Consider again the draw:on: method specialised on the \{city-maps, gps\} combination from snippet 4.6 on page 71. Suppose that at a given moment line 5 is being executed, that is, the default map elements are being drawn by the original version of draw:on: shown in snippet 4.3 on page 69. At this point the user decides to remove the GPS module from the smartphone.\(^{11}\) The acme-gps context will thus be deactivated, and as a consequence, also the delegate gps context will become inactive. The smartphone is henceforth unaware of GPS-related functionality. When control returns from the resend method and reaches line 6, the coordinates message sent to locator will not be understood, since the coordinates method is specialised on the gps context that is no longer active. The problem, stated generally, is that a context has been switched off in the middle of execution of a method that depends on that context to work properly, thus invalidating the remainder of the computation of the method.

Brittle code that depends on unreliable resources such as network connections and removable peripherals could be surrounded by try/catch blocks. However, this solution would result in tangled, less readable code. Our solution is of another nature. Instead of passing directly from having a context to absence of the context, we go through a series of intermediate contexts that allow us to gracefully degrade the system. Each stage can have specialised methods that deal with the situation and exhibit context-adapted behaviour. For our running example, we define one intermediate degraded-acme-gps context. The evolution of the context can be depicted as a chain of available services:

\[
\cdots \rightarrow \text{acme-gps} \rightarrow \text{degraded-acme-gps} \rightarrow \emptyset
\]

The disconnection of the GPS module will result in the deactivation of acme-gps and in the activation of degraded-acme-gps. Note that the latter must also comply with the GPS framework contract explained in section 4.3.1 on page 69, so that the ongoing execution of methods that depend on the GPS service is not disrupted when degraded-acme-gps is switched on. The code implementing degraded functionality is shown in snippet 4.7.

After a predefined timeout of (for instance) 10 seconds, if the connection with the GPS service has not been regained, the degraded-acme-gps context can be deactivated. However, the previous methods are still insufficient for tackling our problem. When the system is running in degraded-GPS mode, the draw:on: method specialised on the \{city-maps, gps\} combination is still applicable, as explained next. The current combination contains degraded-acme-gps and

\(^{11}\)In a similar scenario in which the GPS service were provided by a network peer, removing the hardware module would be analogous to losing the connection with the peer.
define: #degraded-acme-gps as: object clone.
degraded-acme-gps add-delegation: gps.

in-context: degraded-acme-gps do:
[ locator longitude
  [ print: 'returning extrapolated longitude' ].

  locator latitude
  [ print: 'returning extrapolated latitude' ].
]

Snippet 4.7: Degraded GPS functionality

in-context: { city-maps, degraded-gps } do:
[ draw: map (map-section) on: display (canvas)
  [ resend.
    draw: disconnection-icon on: display notification-area ]
]

Snippet 4.8: Degraded version of draw:on:

city-maps, and the former delegates to gps, so together these constitute a supercombination of {city-maps, gps}. In other words, there will be a delegation link that leads to the {city-maps, gps} combination. Since the draw:on: method is still applicable in degraded mode, it can be that the decision of removing degraded-acme-gps from the context is made at the wrong moment, precisely in the middle of the execution of draw:on:. This will raise exactly the same problem described previously for the deactivation of acme-gps. The solution is to define a degraded version of draw:on: that does not rely on GPS-specific functionality (i.e. that refrains from invoking methods from gps). The new version of draw:on:, shown in snippet 4.8, first invokes the plain map-drawing behaviour and then draws a notification icon telling the user about the loss of GPS signal. We use a new degraded-gps context instead of degraded-acme-gps, so that the degraded draw:on: method can be used with any vendor. The degraded-gps context extends the GPS framework. Hence, all vendors will be aware that degraded functionality needs to be defined.

A birds-eye view of the context configuration described so far is given in figure 4.4. Delegation relationships among context objects are represented by arrows, and the pseudo-code of methods specialised on those contexts is associated with dotted lines. The net effect is that, instead of abruptly passing from the navigational CityMaps to the static CityMaps application, which would be disconcerting for the user, the degraded versions of the draw:on:, latitude and longitude methods will maintain the behavioural coherence of the application, ensuring a smooth context transition. After a given timeout, the
draw: map-section on: canvas
[ read GPS latitude and longitude.
  resend (invoke overridden behaviour).
  draw user persona on display ]

city-maps

{ city-maps, gps }

gps
degraded-gps
degraded-acme-gps

acme-gps

draw: map-section on: canvas
[ draw map background.
  draw map streets.
  draw map labels ]

{ city-maps, degraded-gps }

locator latitude
[ read latitude from hardware ]

locator longitude
[ read longitude from hardware ]

locator latitude
[ return extrapolated latitude ]

locator longitude
[ return extrapolated longitude ]

Figure 4.4.: Combined contexts and their associated behaviour.
4.3. Context Management

degraded-acme-gps context will be switched off. The notification icon will no longer be shown and the map will be static again. The GPS service will have been completely —and gracefully— removed from the system.

The technique just explained sketches a possible way of maintaining coherence in face of abrupt context switches (in this case, interruption of services). It is the programmer’s responsibility to foresee such degraded contexts, since degradation cannot be handled automatically —it is domain specific as illustrated by the example. A supporting programming and design methodology is needed whereby programmers can more easily spot possible degradation cases. We believe such methodological support can be coupled to Context-Oriented Domain Analysis (CODA) [37]. This line of future work is described in section 8.3 on page 195.

Context activation coherence

Another important part of context management consistency is the coherent activation and deactivation of contexts. We distinguish the following cases:

**Requested activation** A requested or explicit activation (resp. deactivation) is one that occurs as a consequence of explicitly calling the activate: method (resp. deactivate:). In the running example, an explicit activation request activate: acme-gps is made when the GPS hardware module is plugged in.

An explicit activation adds the given context to the current context combination. The extended combination becomes the current context. Since this combination delegates to the subcontext, the subcontext becomes part of the current context graph.

The explicit activation of a context that is already part of the current combination will not change the current combination, but the context management system will record this additional activation by incrementing an activation counter. Upon a deactivation request, the counter is decremented, and if it reaches zero, the deactivated subcontext will be effectively removed from the current combination. The rationale behind this mechanism is that contexts may be activated by different context management agents (that track different kinds of information from the context discovery subsystem), and a context is effectively no longer needed only when all agents have deactivated it.

**Induced activation** Context activations and deactivations observe delegation relationships. When a context is activated explicitly, its delegates will be implicitly activated as well, because they will also become part of the current context graph. We say that the activation of the delegates is induced, as a consequence of a requested activation. In the example, the activation of acme-gps induces the activation of its delegate gps.

The context management system must detect all induced activations to properly maintain context activation counters. When a context is expli-
citly activated, the counters of its implicitly activated (delegate) subcon-
texts are also incremented. In this case, the increment marks the use 
of the subcontext by one of its supercontexts (instead of the explicit use 
by an external agent that called activate:). A reciprocal mechanism is 
used for deactivation.

With the context activation mechanism just described, context combinations 
are akin to multisets. When a context is activated, all its subcontexts, and 
the context itself, are added to the multiset. When it is deactivated, the same 
elements are removed from the combination —although some of them might 
still belong to the combination if more than one occurrence was present. If 
a context happens to be the last occurrence in the multiset to be removed, 
then the context combination will no longer exhibit the behaviour from that 
particular context which is now absent. This context management technique 
allows the support of nested and interleaved context activations, exemplified in 
section 7.2.2 on page 157.

4.4. Fragility Due to Dynamic Behavioural Changes

It is generally accepted that dynamism brings fragility. This is the main flag-
ship used to advocate static typing. Whether there is an inescapable trade-off 
between dynamism and robustness, as seems to be the case of the traditional 
trade-off between time and space efficiency in algorithms, is still to be determ-
ined.

Upon undertaking our research in Ambience, we have deliberately privileged 
dynamism in the design choices of our language. We wanted to explore how 
much potential for adapting application behaviour dynamically according to 
changing contexts could be brought to the language, while still retaining the 
possibility of writing elegant and easy to understand programs. Nevertheless, 
this high level of dynamism can be detrimental to robustness. Therefore, in this 
section we review some of Ambience’s sources of fragility in detail and sketch 
possible ways of dealing with them. Since we have not incorporated any of 
these possible solutions in our model yet, they are speculative and constitute 
lines for future research. However, in discussing them we shed some light on 
the potential advantages and shortcomings of our approach, and we argue that 
faced problems are likely to be overcome and constitute no ultimate barrier to 
the applicability of our proposed computation model in large scale software.

4.4.1. Fragility Sources

Ambience not only is dynamically typed, but also includes at least four other 
degrees of dynamism, which we group in the following two major categories.

---

12A multiset is a set-like object in which order is ignored, but multiplicity is explicitly signi-
ﬁcant. Therefore, multisets \{1,2,3\} and \{2,1,3\} are equivalent, but \{1,1,2,3\} and \{1,2,3\} 
differ.
This section gives an overview of these dynamic properties and the next section discusses the ways they might threaten software robustness.

**Dynamic protocol modification**

This category encompasses *dynamic inheritance* and *dynamic method modification* (i.e. open objects). The question arises as to which execution points are safe for dynamic protocol modification, such that behaviour remains consistent and follows end-user expectations.

Dynamic inheritance is the capability of modifying delegation links at run time. The topology of an object’s inheritance graph is therefore dynamic. Many authors have acknowledged the modelling power and elegance of dynamic inheritance but at the same time have called for ways to harness this power and to make it more amenable to “disciplined use” [72, 7]. In a sense, technology based on dynamic inheritance is lagging behind technology based on static inheritance when it comes to aspects of discipline. Conversely, static inheritance approaches lag behind in the areas of system dynamics and late composition [109].

Dynamic method modification is the capability to add, redefine and remove methods from the system at run time. With dynamic method modification, the protocol of an object may be extended, reduced, or its implementation be modified —including the case of a newly introduced method that overrides an inherited method.

**Dynamic behaviour selection**

This category encompasses dynamic dispatch (multiple dispatch) and dynamic method scoping (context-dependent behaviour). The main concern raised by dynamic behaviour selection is predictability and determinism in the selection process —namely, knowing the conditions or invariants under which specific behaviours will be chosen, so that applications can be guaranteed to work reliably.

With multiple dispatch, the decision on the applicability of methods is influenced by the run-time value of passed message arguments. In Ambience, the kind of arguments of a message cannot be predicted statically, due to dynamic typing and to dynamic inheritance. This makes it difficult to establish guarantees at development time on the methods that will be chosen for a given message.

Dynamic method scoping consists in letting the calling context determine the methods that are visible to the caller. In Ambience, the current context is a

---

13 Slot modifications can be observed only through the accompanying modification of their accessor methods. Hence, we consider *dynamic slot modification* as a particular case of *dynamic method modification*.

14 Dynamic method modification enables *dynamic code upgrading*, which is most interesting for highly available systems [5]. The ability to change small parts of application logic, in a controlled manner, is useful for systems that should not be brought down for maintenance. This is the kind of systems for which Ambience has been conceived.
kind of dynamic scope\textsuperscript{15} which can change on the fly. Hence, the sets of methods that are visible to the caller change constantly according to context.\textsuperscript{16} If effected naively, context changes are essentially unpredictable: they depend on user mobility, weather conditions, etc. Hence, the establishment of behavioural guarantees requires managing context changes carefully.

4.4.2. Fragility Cases

The properties described in section 4.4.1 enable \textit{dynamic behavioural changes} and \textit{dynamic object reconfiguration}, which are key features for dynamic adaptation to context. At the same time, these dynamic properties are potential sources of fragility if not managed properly.\textsuperscript{17} The use of Ambience's dynamic features “as is” can be hindered by what we call \textit{role tangling}, \textit{inopportune behavioural changes} and \textit{inappropriate method ordering}.

Role tangling

Changes in the inheritance graph of objects lead to changes in the specificity of methods implemented by those objects. Some specificity changes are the very goal of switching a given delegation link: the programmer intends to give precedence to the behaviour of an object over another, according to different run-time needs.\textsuperscript{18} However, it could be that the programmer switches a delegation link to put forward (i.e. render more specific) some of the methods of an object, but does not necessarily want the rest of the behaviour of that object to also be brought forward and be expressed by the delegating object. Hence, as a result of a delegation switch, part of the methods of the delegate might interact with other behaviour in unintended ways. If the methods that became visible unintentionally have the same contract than the methods they override, the unintended behaviour will be consistent, although it will not necessarily be desirable or well suited from an end-user perspective. If the newly visible methods implement a different contract, the system will behave erroneously and probably break.

The root cause of the problem described previously is that objects are “all or nothing” units of reuse, in which two or more \textit{roles} \textsuperscript{[4]} are usually tangled. As an example, a \texttt{circle} object might mix geometric information (e.g. its radius) with graphic information (e.g. its colour). The programmer might want to put forward the geometric part, but not the graphic part of the object—this is impossible, since the object couples both parts. A programmer cannot have only a subset of the methods of an object become more specific, unless

\textsuperscript{15}This is explained clearly in section 5.2.4 on page 95.
\textsuperscript{16}In this sense, context-dependent behaviour selection belongs to the previous category too, since it leads to \textit{dynamic protocol modification}.
\textsuperscript{17}The same happens with dynamic types: they bring flexibility, but can augment software fragility if used carelessly.
\textsuperscript{18}A simple example of this is given in section 2.2.1 on page 23, about a stack that chooses among three delegates depending on its state: \texttt{empty-stack}, (regular) \texttt{stack} and \texttt{full-stack}.
the object is split in two or more separate pieces, such that only one of those pieces can become a delegate. Splitting the object raises issues about object identity: each object piece now has a separate identity. Also, having many small objects that have only a few methods raises the conceptual complexity of programs. Bardou and Dony [7] introduced split objects to support separation of behaviour yet preserve object identity. Split objects aim at keeping a one-to-one correspondence between domain entities and their object representation, yet support multi-faceted objects, since in reality the corresponding entities can also be seen from different perspectives, or accomplish different roles. The canonical example is a person object, which can be seen as employee, sportsman, citizen, etc. Again here, the programmer might want to bring forward the behaviour as sportsman, but not as employee. Split objects help overcoming this problem.

From the previous reasoning about split objects it could appear that context-oriented programming (subjective object behaviour) can provide similar answers to the problems of behaviour granularity and role tangling. Context-oriented programming allows objects to remain in one piece, yet exhibit different behaviour according to the context in which they are used. Nevertheless, the semantics is not the same. In split objects, the role goes with the reference. Two different references might point to the same object (identity), but to different roles of that same object. In context-oriented programming, all references to an object give access to the same behaviour; this behaviour can vary according to context, but still all references will see the same behavioural adaptations. Hence, it would not be redundant if a language supported split objects and subjective objects, and such mix could solve role tangling, if used appropriately.

Another solution to role tangling is encapsulation policies [97, 96]. An encapsulation policy expresses how a client can access the methods of an object: which methods can be called or overridden by the client. The designer can associate an arbitrary number of encapsulation policies to an object, or these policies could conceivably be generated after design (e.g. at deployment time or even at run time) if needed. As with split objects, the encapsulation goes with the reference: two different references can point to the same object, yet give access to different subprotocols of that object. Split objects seem more powerful in that they not only allow to expose different protocols of a same object, but also they allow to exhibit a same protocol which is implemented differently (by two different pieces of the object); the latter is impossible with encapsulation policies. However, encapsulation policies permit preventing a client from overriding certain methods, which could be exploited to avoid the problem mentioned previously of unexpected method overriding.

**Inopportune behavioural changes**

A dynamic behavioural change in an object (e.g. due to a delegation link switch, or to the definition of a new method) might be effected in the middle of execution of a method that relies on the original behaviour of the object to work properly. Since that behaviour is no longer in force, the remaining
computation of the method is invalidated.\footnote{As said before, if the new behaviour implements the same contracts than the previous one, the system might not break, but still there is the possibility that exhibited behaviour is not well suited according to end-user needs or expectations.} We call these behavioural changes \textit{inopportune}: they come at the wrong time. This kind of fragility has been exemplified once already, in section 4.3.3 on page 74.

Inopportune behavioural changes have been investigated for the GILGUL language~\cite{gilgul2000, gilgul2001}. Similar to what is done in GILGUL, a possible solution to avoid inopportune behavioural changes would be to block the thread that is attempting to replace a delegate with another one, if a method belonging to the delegate happens to be currently executing.\footnote{The method might belong to the delegate itself, or may be inherited from delegates deeper in the delegation graph.} When a method starts executing, the delegation graphs of the method owners\footnote{Recall from section 3.1 on page 30 that in our model method ownership is shared among the method’s specialisers, rather than being held by only one object or class as in traditional object-oriented languages.} would be “frozen” so that the method can complete execution safely. They are frozen by blocking the threads that attempt concurrent modification of the graphs’ topologies. The previous strategy would lead to a deadlock if modifications of a graph are attempted in the same thread that has frozen the graph. GILGUL throws an exception in such cases. Another possibility would be to defer the delegation switch (e.g. accumulate the switch in a queue of pending operations) until it can be performed safely.

Yet another possibility to manage inopportune behavioural changes was the one discussed in section 4.3.3: to allow the immediate change, even if it happens in the middle of execution of a method, but making sure that the new behaviour respects the same behavioural contracts\footnote{The contracts are specified as part of framework contexts, explained in section 4.3.1 on page 68.} and is well suited according to end-user expectations. In said example, the \texttt{degraded-acme-gps} can immediately take over the computation that \texttt{acme-gps} left over, because it provides a smooth behavioural transition by interpolating the coordinates. This solution is desirable in situations that require immediate, real-time system response, but is domain-specific and requires careful planning —it is still the programmer’s responsibility to maintain behavioural coherence.

\textbf{Inappropriate method ordering}

Although inheritance is well-suited for incremental construction of more specific objects starting from existing less specific ones, it is not necessarily appropriate for composing reusable building blocks \cite{inheritance2002}. Object composition is done by means of (multiple) inheritance, but it is difficult to ensure that the linearisation produced by Ambience will always order objects in the most appropriate way. The measure of “appropriateness” can be domain specific, or worse, it can depend on the preferences of each particular user.

A possible way of overcoming the shortcomings of automatic linearisation mechanisms would be to have a more \textit{declarative} or \textit{intentional} approach to
describing method applicability, such as predicate dispatch (discussed in section 2.1.4 on page 19), or predicate objects (discussed in section 2.2.2 on page 23). Filtered dispatch [31] could also help encoding the programmer’s intention more accurately than plain inheritance relationships do. Even so, resolution of ambiguous messages would still be an issue with these more declarative approaches, because ambiguity is not always due to a lack of expressiveness in encoding behaviour applicability—in many situations, even end users would hesitate as to what is the most appropriate behaviour that the system should exhibit. The same resolution strategies described in section 3.5.2 on page 46 could be used to handle arising ambiguities.

As mentioned previously, we have chosen for dynamism in our design of Ambience. Even though our experience has been positive so far, we still need to make a full assessment of the fragility problems described in this section by experimenting with large scale scenarios. In face of the fragility issues raised by dynamism, we could decide to make Ambience more static. However, we can also decide to keep the language’s very dynamic nature, and develop new supporting tools to help the programmer build dynamic programs that are as robust as possible. These tools can be advanced language abstractions, composition techniques, development environments, methodologies and guidelines. Future research along these lines is suggested in section 8.3 on page 189.
5 The Ambient Object System

Whereas our model has been presented in chapters 3 and 4 using a Smalltalk-like surface syntax, its core has been written in Common Lisp. We call this core the Ambient Object System (AmOS). In essence, AmOS is a prototype-based object layer built on top of Common Lisp, featuring multimethods and subjective dispatch [93]. AmOS does not extend Common Lisp’s Object System (CLOS) [13], in particular because AmOS does not have a notion of class [85].

In complement to the previous chapter where we illustrated the main features of our Ambience language and how they support run-time behaviour adaptation of applications to changing contexts, in this chapter we open up the inner workings of the underlying object system, thereby exposing in detail the kernel on which Ambience is based. Our object model is thus revisited from a meta-programmer’s perspective. The chapter introduces a few novel constructs not shown before.

5.1. Mapping Ambience to AmOS

Before diving into the core abstractions of our model, this section shows the way Ambience syntax is mapped to AmOS syntax, and how AmOS can be used independently from Ambience. The remainder of this chapter, and also chapter 7, use standalone AmOS.

5.1.1. Syntax Mapping

Internally, Ambience translates the expressions it reads to Lisp syntax. We retake a running example from chapters 3 and 4 to show the mapping from Ambience to AmOS. Recall the receive:on: method to handle incoming calls from section 3.4.1 on page 41:

receive: call (phone-call) on: phone (mobile-phone)
[ advertise: call on: phone.
   enqueue: call in: phone incoming-calls ].

Ambience’s parser translates the code to Lisp form as follows:

(defmethod receive:on:
   ((call (phone-call)) (phone (mobile-phone)))
   (advertise:on: (call) (phone))
   (enqueue:in: (call) (incoming-calls (phone))))

It does so by means of a Generalised LR (GLR) parser implemented in GNU Bison. The grammar of Ambience is shown in section B on page 201.
This example shows the translation of a keyword method definition, and a few keyword, unary and nullary messages used in the body of the method. The other kinds of methods and messages described in sections 3.2.2 and 3.2.3 are translated in an analogous way. Note that method specialisers are obtained from evaluation of messages. Arbitrary expressions can be used if needed, beyond the simple (phone-call) and (mobile-phone) nullary messages.

To render the syntax less verbose, AmOS supports the expression of argument and prototype-accessing messages as if they were plain variable references:

```lisp
(defmethod receive:on: ((call phone-call) (phone mobile-phone))
  (advertise:on: call phone)
  (enqueue:in: call (incoming-calls phone)))
```

This takes the syntax closer to normal Lisp style, even though under the surface messages are being sent. This is discussed further in section 5.2.6.

### 5.1.2. Standalone AmOS

We have decoupled AmOS from Ambience so that AmOS can be used separately as a standalone object system for Common Lisp. Hence, instead of only including the minimal functionality needed to support Ambience, AmOS features user-end syntax (in the form of macros such as `defmethod`) that makes it easy to define methods, send messages and work with prototypes and contexts in Common Lisp.

To illustrate standalone AmOS usage, we show one of the running examples from previous chapters on how the behaviour of a mobile phone can be programmed and made adaptable to the context. This understanding of AmOS is necessary for the discussions in this chapter and chapter 7. The example concentrates on functionality related to receiving and advertising calls on mobile phones, with the following requirements.

- Urgent calls are treated with priority over normal calls.
- Incoming calls can be advertised by playing a ringtone or by activating a built-in vibrator.
- The choice between ringtone and vibrator depends on the current environment: the ringtone is used by default, whereas the vibrator should be used in silent places like museums, libraries and situations such as meetings.
- Calls received while the user is sitting inside a car should mute the car’s radio and be advertised on the car’s speakers.

In our example, we first create a `@telephony` context, representing a prototypical situation in which a telephony service is available. Inside a mobile phone such service always is:

2Aimed at improving understandability, the `defcontext` construct is syntactic sugar for
5.1. Mapping Ambience to AmOS

(defcontext @telephony)

By convention, prototype names are prefixed with the @ symbol. The prototypical @telephony context thus created is a plain object, without any special status in comparison to other objects in the system.

Next we proceed to define objects and behaviour that are specific to the telephony context. For the sake of the example, a phone object simply contains a list of incoming calls and a speaker on which to advertise those calls:

```lisp
(with-context @telephony

  (defproto @phone (clone @object))
  (add-slot @phone 'incoming-calls (list))
  (add-slot @phone 'speaker 'phone-speaker)
  (defproto @mobile-phone (extend @phone)))
```

For simplicity, we use a Lisp symbol to identify the speaker, but in a fully developed application, the speaker would be a more complex object with suitable behaviour. On line 5 the result of extend is an empty object that delegates to the object being extended. As a result, all behaviour that is not understood directly by @mobile-phone will be handed over to @phone. Still in telephony context, we define a phone call as an object that can be received on any phone:

```lisp
(with-context @telephony

  (defproto @phone-call (clone @object))
  (defmethod receive ( (call @phone-call) (phone @phone))
    (advertise call phone)
    (add-incoming call phone))
  (defmethod advertise ( (call @phone-call) (phone @phone))
    (format t "Playing ringtone through ~a" (speaker phone)))
  (defmethod add-incoming ( (call @phone-call) (phone @phone))
    (enqueue call (incoming-calls phone))))
```

The receive multimethod is specialised on both @phone-call and @phone. It encodes the prototypical behaviour for receiving calls on a phone: the call is advertised and added to the list of incoming calls. The advertise method encodes the prototypical way of announcing a call to the user, i.e. by playing a ringtone. The add-incoming method encodes the prototypical way of treating an incoming call, i.e. by enqueuing it to the phone’s list of incoming calls.

Behaviour that is better suited for urgent calls can be defined by overloading add-incoming as follows:

```lisp
(defslot @telephony (clone @context)). This adds a slot named @telephony to the current context object whose value is a clone of the prototypical context object.
```

In essence, the (with-context context body) construct is syntactic sugar for (activate context) body (deactivate context); context activation is explained in section 4.2.1. However, the expansion is slightly more complicated than this, since it also uses Common Lisp’s unwind-protect construct to make sure that context is deactivated even if control flow exists prematurely from body because of (e.g.) an exception.

The defproto construct is a synonym of defslot for addition of a slot with given name and value to the current context. We prefer the use of defproto over defslot because it encodes explicitly the programmer’s intention.
This version of add-incoming, specially conceived for urgent calls, puts the call in the front of the incoming call queue instead of at the end. Overloaded multimethods permit defining behaviour that is better suited to certain kinds of objects.

Functionality that is specific to car context can be defined as follows:

```
(defcontext @car)

(with-context @car
  (defproto @radio (clone @object))
  (defmethod mute ((device @radio))
    (format t "Muting radio-"))
)
```

In the context of cars with a radio on board, the behaviour of the advertise method can be specialised so that the car’s speaker is used instead of the phone’s built-in speaker, after the car’s radio has been muted:

```
(with-context (@telephony @car)
  (add-slot @phone 'speaker 'car-speaker)
  (defmethod advertise ((call @phone-call) (phone @phone))
    (mute @radio)
    (resend))
)
```

This version of the advertise method is specific to the combination of the @car and @telephony contexts. Whereas the @telephony context is inherent to the phone and is always active, the @car context is activated or deactivated dynamically when the user enters or leaves a car. The behaviour just defined will be exhibited only when the phone is in car context. Note on line 2 that a context-specific slot is added to @phone, for which a speaker accessor method will be defined in the (@telephony @car) context combination. When this combination is inactive, the original speaker accessor method (and thus original slot value) will be used.

All code shown so far is written at development time and deployed into the phone. During normal use, actual mobile phones and phone calls are created by cloning the respective prototypes, and behaviour is triggered by invoking multimethods like receive:

```
(let ((bobs-phone (clone @mobile-phone))
       (alices-call (clone @urgent-call))

    (receive alices-call bobs-phone))
the default output will be:
Playing ringtone through PHONE-SPEAKER
```

whereas in car context the output of the same expression will be:

```
5Context combinations are explained in section 4.3.2.
```
5.2. Opening Up AmOS

The core concepts explained in section 3.1 on page 29 are not only meant for end programmers. The mechanisms used in the lowest levels are no different from those exposed to users, namely objects, cloning, messages, delegation, and multimethods. The inner workings of these important entities are not mere implementation details; rather, they are key to understanding thoroughly the context adaptation mechanisms put in place by AmOS. Furthermore, by exposing the inner workings we show the simplicity, homogeneity and flexibility of the model at its core, and explain the rationale behind the different language design choices.

The mechanisms described in this section underlie the ones explained in chapters 3 and 4 for Ambience. In those chapters, the discussion is given from an end-user perspective. In this section, we describe the fundamentals of core model abstractions from a language implementer’s perspective, or rather, a meta-programmer’s perspective. The section shows that AmOS is an open system [69, 53] and that open support of contexts and other constructs adds no complexity to the model. AmOS is not only open, but also accessible, meaning that meta-programmers can quickly get acquainted with the fundamental mechanisms.

5.2.1. Objects: Slots, Maps and Delegation

References to an object’s acquaintances are held in so-called slots. In AmOS, every object is basically a slot array—that is, a collection of references to other objects. Hence, in AmOS the distinction between objects and object arrays is (purposely) blurred. The only observable difference between a “pure array” and other kinds of objects is that arrays are generic containers with anonymous slots that usually have no special semantics, whereas other objects attach a name and a meaning to some or all of their slots. It is perfectly possible to have an object that is half a pure array and half not, if for example it has a header

Footnote: With computational reflection [75], the distinction between implementer and meta programmer is blurred.
of named slots, followed by a tail of anonymous slots. Arrays are to AmOS what lists are to most Lisp dialects: a basic data structure used to represent all entities in the computation model.

Slot Names

In AmOS, slots are not associated with a name *per se*. Rather, slots are associated with their corresponding accessor methods. The name of the reader method is considered the name of the slot. Given that it is possible to define any number of accessors for a slot, slots can have multiple names, or be anonymous—if no corresponding accessor method is defined (in this case, the slot can only be accessed by index, in an array-like fashion). Furthermore, accessor methods can depend on the context. This means that a same slot can have different names in different contexts; similarly, different slots of an object can have the same name—the slot that is actually read with that name depends on the current context.

Cloning Family Maps

In AmOS, new objects are created by cloning existing ones. Cloning means replicating the state of the original object into the new object. Since replication of all information would be costly in terms of time and particularly space, efficiency is regained in AmOS thanks to *cloning family maps*, a concept originally proposed by the designers of Self [21]. Cloning family maps are constant pools of information that is common to all members of a cloning family. This way, there is only one physical copy of all shared information, thereby achieving the same storage efficiency of class-based languages. All members in the family have a reference to their map. The map is a normal object containing among others the methods, delegations and constant slots of the members in the family. The only piece of information that does not have shared storage is the array of mutable slots, which is kept individually per object.

Unlike Self [21], a cloning family map does not consist of one atomic object, it is a whole object graph containing the shared information. What we call the *map object* is the handle or entry point to the whole graph. Figure 5.1(a) shows a family map shared by three clones of the same family; all three have the same delegations and roles (the purpose of these structures is unimportant here). Family maps are maintained with a copy-on-write policy. An attempt to change a piece of information that is stored somewhere in the map graph of an object (e.g. a delegation link) gives rise to a chained cloning operation: AmOS replaces the path of objects starting from the main map object (map 1 in figure 5.1(a)) and leading to the map object whose slot is going to be written (delegations 1 in figure 5.1(a)), by a cloned path—that is, a path made of clones of the original path members. Figure 5.1(b) shows the replaced path.

---

7This corresponds to a *semi-array* or *annotated array*: think of an IP network packet with a header followed by the payload, or a PNG image with a header followed by raw pixel data.
5.2. Opening Up AmOS

(a) Family map shared by three clones. (b) Split family map, after having modified a delegation slot of object b.

Figure 5.1.: Cloning family maps and their copy-on-write policy.

(a) Conceptual delegations and their graphical depiction (b) Actual objects and object references involved

Figure 5.2.: Delegation objects.

made up of map 2 and delegations 2. Other objects in the map, such as roles 1, remain shared between the two family maps. The modification of an object’s slot with shared storage always gives rise to a family split, because the affected member becomes the sole member of a new cloning family, as illustrated in figure 5.1(b). The remaining cloning family members stay in the original family, all still referring to the original, unmodified family map.

Delegation Objects

Unlike Self [115], there is no distinction between plain slots and delegation slots. Rather, delegates are kept in a separate delegation object. Figure 5.2 shows the correspondence between delegations at a conceptual level, and the underlying object structure involving delegation objects and plain object references. The order of an object’s delegations corresponds to the order in which the delegations are stored in the corresponding delegation object.

Having no distinction between different slots types brings derived advantages,
The Ambient Object System

prototypical arguments

parent

current activation

clone

Figure 5.3.: Prototypical activation and cloned activation with actual arguments. Solid arrows represent object references, the hollow arrows represent delegations.

for instance that there needs not be separate operations for plain slot addition and delegation slot addition in the language kernel. On the other hand, special delegation accessors\(^8\) methods become necessary to access the slots located in the delegation object.

5.2.2. Closures and Activations

The most basic executable entity in AmOS is the closure. It has \textbf{lambda}-like syntax and semantics, as the following example illustrates:

\[
(& (x y) (+ x y)) \rightarrow \text{closure}
\]

Every closure has an associated \textit{activation record} —hereafter simply called \textit{activation}— which holds the dynamic information that is associated with its invocation. Activations are the environments in which closure code is executed.\(^9\) Like in Self \([21]\), activations are first-class objects.

It is possible to specify prototypical argument values to be held in the activation of a closure. They are placed next to each argument name:

\[
(& ((x 1) (y 2)) (+ x y)) \rightarrow \text{closure}
\]

This closure is illustrated in figure 5.3. As can be seen, the prototypical activation delegates to an \textit{arguments} object, which holds one slot per closure argument. Upon invocation, the closure activation is cloned and the prototypical arguments are substituted by the actual arguments. The closure’s code is then executed in this freshly created environment and is thus fully reentrant. Figure 5.3 shows the fresh activation resulting from the following invocation:

\[
(\text{invoke} (& ((x 1) (y 2)) (+ x y)) (\text{list} 3 4)) \rightarrow 7
\]

\(^8\)See section 5.2.6.

\(^9\)Activations are the object-based version of what is usually known as stack frames in other models.
Each activation delegates to a parent object, also illustrated in figure 5.3. Messages not understood by the current activation or by its arguments object are delegated to the parent. The parent corresponds to the enclosing lexical scope of the closure, so that outer definitions can be seen inside the closure’s environment. For the particular case of the top-level activation, which has no enclosing lexical environment, the parent is the so-called current context. This context link is crucial to our approach and is explained further in section 5.2.4; it is also discussed in section 4.2 on page 63.

As has been shown in this section, the semantics of closures involves nothing more than objects, cloning and delegation. The next section explains methods and their dispatch infrastructure.

5.2.3. Methods, Specialisation and Lookup

Methods are obtained by enriching closures with a dispatch mechanism. Since methods are extended forms of closures, the execution semantics described in section 5.2.2 applies unmodified to methods. In the case of methods, the prototypical arguments are considered to be argument specialisers. The code of the method is designed to work for those specialisers in particular, and for any extension (through delegation) thereof. Reconsider for instance the receive method introduced in section 5.1.2:

```lisp
(defmethod receive ((call @phone-call) (phone @phone))
  (advertise call phone)
  (add-incoming call phone))
```

The receive method is basically a named closure with prototypical arguments @phone-call and @phone, which are used as specialisers. Figure 5.4 depicts applicability of this method for the arguments alices-call and bobs-phone introduced in section 5.1.2.

---

10The order of delegations is important here. The arguments have more precedence than the parent by having an earlier position in the delegation object of the activation (recall...
Roles

The link between a method and its specialisers is established through roles, originally proposed in the Prototypes with Multiple Dispatch (PMD) model [93]. Any object that is used as method specialiser plays a role in the interaction prescribed by the method. As illustrated in figure 5.5, the argument specialisers @phone-call and @phone play a role in the receive interaction, at the first and second positions respectively. The illustrated roles are triplets \((s, i, m)\) of the selector \(s\) identifying the interaction, the position \(i\) at which the object plays the role, and the method \(m\) implementing the behaviour.

Figure 5.5 also shows the conceptual difference among the different kinds of objects. Objects in the plain layer correspond to concrete domain entities that are being manipulated at the moment; objects in the prototypes layer are prototypes (usually meant for cloning, rather than direct manipulation); finally, the core computation model is available through a series of meta objects describing base objects, their roles, methods, and so on.

Method specialisation is useful in defining behaviour for special kinds of objects and dealing with particular cases without hard-coding conditional statements. Section 5.2.4 explains the way we further exploit specialisation and multiple dispatch to define context-specific behaviour, and the way such behaviour can be adapted dynamically as needed.

section 5.2.1 on page 91). Figure 5.3 does not depict this order.
Method lookup and rank vectors

Figure 5.5 illustrates the intuitive idea behind method lookup. Basically, a method is applicable if it is reachable for every argument position, starting from the actual arguments (bottom part of the figure) and following delegate objects (upwards in the figure). We now discuss a few notions regarding method lookup that were left out from section 3.5 on page 44.

Method overloading brings about the problem of choosing the method version that is best suited to the given arguments. Specificity among applicable methods is defined by rank vectors [93]. Each rank vector entry contains the delegation distance between the message argument and corresponding method specialiser. For instance, the rank vector of the method illustrated in figure 5.5 for the message with arguments alices-call and bobs-phone is (2, 2), since the path in the delegation graph that goes from message argument to method specialiser is of length 2 for both arguments. As another example, the version of the add-incoming method specialised on urgent calls (see section 5.1.2) has rank vector (1, 2), since alices-call is one hop away (delegation-wise) from @urgent-call, and bobs-phone is two hops away from @phone. A rank vector with only zeroes is a “perfect match”, corresponding to the case where the message arguments are the very method specialisers.

In calculating rank vectors and determining their specificity there are two sources of ambiguity, already introduced in section 3.5.2 on page 46. We review them in their relationship to rank vectors:

1. Multiple inheritance renders the notion of “delegation distance” ambiguous: starting from an actual argument, two objects found by following different delegation paths can be at the same distance from said argument. We use an adapted version of the C3 linearisation algorithm [8] to topologically sort the delegation graph of each actual argument and thus obtain a well-defined notion of distance. This solves ambiguities arising from multiple inheritance.

2. Once rank vectors have been determined for all applicable methods, there must be a way of ordering them according to some notion of specificity so that one particular method can be chosen. Ambiguities arising from multiple dispatch — for example, considering whether the rank vector (1, 2) is more specific than (2, 1) — are precluded by imposing left to right argument precedence (i.e. a lexicographic ordering) as in CLOS: (1, 2) is thus considered more specific than (2, 1). As a consequence, methods with a better match in earlier argument positions are considered more specific than other applicable methods. This solves ambiguities arising from multiple dispatch.

5.2.4. Contexts

This section explains the context-oriented mechanisms of AmOS. Contexts are naturally supported within the execution model of closures and methods (ex-
plained in sections 5.2.2 and section 5.2.3).

**Main Cornerstone: Dynamic Scoping of Behaviour**

Run-time behaviour adaptation is supported in AmOS by introducing a kind of dynamic scoping mechanism for methods. Generally speaking, a *dynamic scope* is a namespace that is used to find the value of identifiers at run time, just like lexical scopes are. What makes dynamic scopes special is that the language provides the tools to change at run time the scope with which dynamically scoped identifiers are resolved. In contrast, the scopes of lexical identifiers are fixed, following the program text structure. Nevertheless, arbitrary dynamic scope changes are usually disallowed, and structured means of changing scopes are provided instead —namely, constructs that execute a user-defined function in an extension of the current dynamic scope, and guarantee that the original, non-extended scope is restored when the function finishes, whether it does so normally or abruptly due to an error. This way, the effect of the extended dynamic scope is confined to the execution of the given function. Since the lifespan of dynamic scopes is usually coupled to the execution of functions, dynamic scopes are said to follow a *stack-like discipline* that parallels the nested executions of functions [106].

One of the main reasons why dynamic scoping is useful is that it allows the caller’s state to influence the behaviour exhibited by the callee in a deep fashion (i.e. across nested method calls). Such influence is not intertwined in the form of arguments that must be passed from one function or method to the next. Clearly, having such *pass-through arguments* is quite inconvenient, as the arguments crosscut all methods and messages that need to be influenced [26], and all possible influences that might prove useful must be foreseen and hard-coded in the Application Programming Interface (API). Dynamic scoping can help alleviating these problems.

Some languages such as Common Lisp offer different namespaces for variables and for functions, whereas other languages such as Scheme use only a variable namespace. Dynamic scopes are yet another kind of namespace, and again can be divided in two, dynamic scopes for variables and dynamic scopes for functions, in languages that draw such distinction. Given that the concept of variable is not intrinsic to AmOS, we need be concerned only with dynamic scoping of methods in our discussion, but not of variables. Furthermore, AmOS

---

11 Namespaces contain *bindings* between *identifiers* and their associated *values*.

12 The user-defined function can be anonymous. For instance, a *(dynamic-let bindings body)* construct can create a parameterless anonymous function with the given *body*, and execute that function in a dynamic scope that extends the current one with the given *bindings*.

13 There are different ways to implement dynamic scoping [90]. One of them consists in passing dynamic scopes as hidden arguments of the language evaluator. All pass-through information is ultimately threaded in the execution of programs thanks to these hidden arguments, which can be consulted deep down in nested procedure calls.

14 As explained in section 5.2.6 on page 101, what looks superficially as variable accesses are actually accessor method invocations.
does not make a distinction between lexical and dynamic scoping. There is only one kind of namespace, *dynamic method scopes*. Lexical scoping is obtained as a particular case.\(^\text{15}\)

**Dynamic Scoping in an Object-Based World**

AmOS identifies dynamic scoping—a concept coming mainly from the functional programming world—with subjective behaviour—a concept coming from the object-oriented world [101]. Subjective behaviour is roughly equivalent to dynamic scoping: it is behaviour that depends on the caller’s point of view or state.

The power of dynamic scoping, or similarly, of subjective behaviour, can be brought to the object-oriented world fairly easily under certain conditions. Any language with multiple dispatch can support subjective behaviour [101], merely by passing with every message an implicit argument that represents the current point of view or state of the caller. This implicit argument participates in the dispatch process as any other argument does. As a result, chosen behaviour will depend on this implicit subjective element [93].

In AmOS, the *current activation* of the executing closure or method\(^\text{16}\) is passed implicitly as first argument of every message. This way, behaviour selection will depend on the current execution environment of the sender. This simple exploitation of multiple dispatch results in a kind of dynamic scoping mechanism that is surprisingly convenient, as illustrated in chapter 7.

**Context as a Graph of Delegating Objects**

As explained in the previous section, the current closure’s activation is passed implicitly as first argument of every message. Hence, for any given message, applicable methods are first looked up in the current activation, and by following the lexical parent link, they are looked up further in enclosing lexical scopes, until the top-level activation is reached. Rather than stopping at this point by having an empty object be the parent of the top-level activation, we assign a plain object which we consider the *current context*. The current context can delegate further to other context objects as needed. Figure 5.6 shows a sample configuration of activations and context objects corresponding to the invocation of the *receive* method. Activation parent links correspond to enclosing lexical scopes and are therefore kept constant, in correspondence to the program text structure. Delegation links starting from the current context object and beyond are dynamically managed and may change at run time. Following normal delegation semantics, messages that are not understood by the static activation chain will be delegated to the current context. The objects that are reachable by delegation starting from the current context constitute the *current context graph* or simply *context graph* (shown in the dashed box of figure 5.6).

---

\(^\text{15}\)Generally, static constructs are always a particular case of dynamic constructs: if a dynamic construct does not change at run time, it is actually static.

\(^\text{16}\)Recall figure 5.3 in section 5.2.2.
The Ambient Object System

Figure 5.6.: Invocation of the receive method. The lexical scope delegation chain is static, whereas the context graph topology is managed dynamically.

The context graph can be seen as a reification of the physical and logical environment in which the system is currently running. Each individual context object represents one part of such environment, and is generally domain-specific. In figure 5.6 for instance, the @telephony context object has a number of prototypes and method definitions that are about telephony. Another example is the @acoustics context, which contains functionality related to the noise level of the surrounding environment. This example is used next.

**Dynamic Adaptation through Context Manipulation**

The behaviour of the system can be different if it is being used in a library, factory, on the street, and so on. In particular, the behaviour of the advertise method used by receive can be adapted to the current acoustic level. As explained in section 5.1.2, the default implementation plays a ringtone through the phone speaker. However, behaviour that is better adapted to a silent environment can be defined as follows:

```lisp
(with-context (@telephony @silent)
  (defmethod advertise ((call @phone-call) (phone @phone))
    (format t "Activating phone vibrator-%")
)
```

This second version of advertise activates the phone vibrator, without producing sound. Note that this method is specialised on two context objects at the same time, namely @telephony and @silent, rather than only @telephony as the default version. This is an example of a context combination. Context combinations are context objects of their own, representing the combination as a whole. Behaviour that is specific to the particular combination can be defined as illustrated previously; other behaviour not specific to the combination is delegated to the constituent subcontexts, thanks to suitable delegation.

---

17 Context combinations are introduced in section 4.3.2 on page 70 for Ambience.
Figure 5.7.: Context graph as combination of contexts.

links as illustrated in figure 5.7. In the figure, contexts to the left-hand side are context combinations (denoted by a plus + sign) of two or more contexts, whereas contexts on the right-hand side are “unary” combinations of only one constituent context. Context combinations are managed by the system, which is in charge in particular of creating them on demand, and placing delegation links from more specific combinations to less specific ones.

When the @silent context is activated (for example, if the system detects that a library has been entered), it will be combined with the currently active contexts. Delegation links among combined contexts are automatically maintained by the system, so that more specific combinations delegate to less specific ones. The current context object constitutes the most specific combination, whereas basic (non-combined) context objects such as @telephony and @silent are the least specific.

5.2.5. Lookup Arguments

AmOS introduces a novel construct called lookup arguments. Whereas actual arguments are the objects explicitly passed by the user, lookup arguments are the objects in which the method lookup mechanism found the currently executing method. More specifically, lookup arguments are the objects owning the roles that led the dispatch process to choose the current method.

Lookup arguments often coincide with argument specialisers, since the specialisers are precisely the objects in which roles are installed when a method is defined. For example, figure 5.5 on page 94 illustrates a case where the actual arguments are alices-call and bobs-phone, but the objects implementing the requested behaviour (owning the roles) are the delegates @phone-call and @phone. These latter two are passed as lookup arguments of the invocation. In this example, lookup arguments and method specialisers coincide. However, in face of cloning, lookup arguments and method specialisers differ, because
The Ambient Object System

the methods of a cloned object still have as specialiser the original object, not the clone. Consider for example figure 5.8, in which a receive message is sent to two arguments that respectively delegate to @phone-call and a clone of @phone. The specialisers of the receive method are still @phone-call and @phone, irrespective of any clones that are made of these specialisers. The clone of @phone will be able to play the receive role in second position, just as the original @phone does. This is illustrated by index “2” in the connection between the objects and the receive method. Hence, the method is applicable and will be invoked. In this case the specialisers do not coincide with the lookup arguments, because the latter will be @phone-call and the cloned @phone, which are the two objects where the receive behaviour has been found.

To access lookup arguments in the body of methods, they can be specified in third position:

```
(defmethod a ((b c d))
  d)
```

In method a, argument b has prototypical value c, and associated lookup argument d; the method just returns its lookup argument. Actually, each component of an argument specification has an associated keyword. Hence in the full (uncontracted) syntax, method a is written as:

```
(defmethod a ((:name b :value c :lookup d))
  d)
```

The order of keywords is unimportant. Thanks to keywords, it is possible to specify only a :lookup argument (without :name), or to have an anonymous argument (that uses only the :value keyword). To avoid verbosity, we use the contracted form of argument specifications whenever possible, as has been done so far.

Lookup arguments can be seen as a sort of “background” information passed by the run-time system on where the current method has been found. This includes the implicit argument passed to every method: there will be a

---

18 Common Lisp keywords are symbols starting with a colon.
19 Having this mental image, at some point we considered calling them shadow arguments.
corresponding *implicit lookup argument* telling the object where the method has
been found for argument position 0. Since methods are implicitly specialised
on their context of definition, this means that the implicit lookup argument will
most often be this very context. If a method is invoked with a context clone,
the implicit specialiser of the method will stay the same (the original context),
whereas the passed lookup argument will be the cloned context. The implicit
lookup argument turns out to be very useful in the implementation of many
fundamental constructs such as accessor methods, resend methods, and sealed
slots, as explained next.

### 5.2.6. Accessor Methods

Sticking to a *pure* object-based semantics, in AmOS there is no such thing as
a variable access. Every action in AmOS is done through message passing. In
particular, argument accesses are actually method invocations (as in Self [115]),
even though at the surface they look like plain variable accesses. Let’s revisit
the receive method once more:

```sh
(defmethod receive ((call @phone-call) (phone @phone))
  (advertise call phone)
  (add-incoming call phone))
```

In the body of the method, the occurrences of the symbols call and phone,
which are the parameter names, are replaced by message sends.\(^{20}\) The symbols
@phone-call and @phone are also messages under the surface.\(^{21}\) The code just
shown is equivalent to the following:

```sh
(defmethod receive ((call (@phone-call)) (phone (@phone)))
  (advertise (call) (phone))
  (add-incoming (call) (phone)))
```

In this code it is apparent that everything is done through message passing.
The choice between the first (implicit) syntax and second (explicit) syntax for
accessing arguments is left to users.

Invocations like (call) and (phone) are message sends with no explicit
arguments, although the activation of the sender (which in this case is the
activation of receive) is passed implicitly as first argument. The arguments of
receive are accessible through this activation. Nevertheless, in some situations
the sender’s activation is not the right object to look for the slots that need to
be read or written. Consider for instance the following closure nested inside a
method:

```sh
(defmethod adder (x)
  (& (y) (+ x y)))
```

Method adder returns a closure that “remembers” x and will add it to whatever
number y it is given as argument. As before, the occurrence of symbol x in

\(^{20}\) Thanks to Common Lisp’s symbol-macrolet facility.
\(^{21}\) They are defined with Common Lisp’s define-symbol-macro facility.
the closure’s body is actually a message send. The accessor method invoked in response to this message receives the sender’s activation as first argument, which at the point where x occurs happens to be the closure’s activation. However, the slot that needs to be read belongs to the method’s activation, not the closure’s. The implicit argument passed to the accessor is therefore useless in this case. For this reason, accessor methods read and write the slots belonging to their environment of definition, rather than the environment of invocation. This environment of definition is passed as implicit lookup argument. In the example, the accessor for argument x is defined on (i.e. added to) the activation of the adder method, and will read from and write to the clones of the activation that are made each time the method is invoked.

Accessor methods have no special status (they are plain methods) and use no special semantics, besides lookup arguments, to access the slots of their host objects. Lookup arguments make it possible to have at most one copy of every accessor method used in the system.

**Accessor kinds**

We call the accessors discussed so far **inner accessors**, because they are used to access the slots of an object “form the inside” —that is, when the object is being used as an evaluation environment, as activations for example are normally used. Accesses to method and closure arguments in activations are not the only uses of inner accessors. Prototypes, stored in context objects, are also accessed by means of inner accessors. For example, the @phone-call and @phone messages invoke inner accessors that read slots from @telephony context.

Besides inner accessors, AmOS features other kinds of accessors; **outer accessors** read slots form the “outside” of an object, as the accessor age in the following expression does: (age person). In this example, the outer accessor receives an explicit argument person from which it is supposed to read a slot (or write it, if setf is used). In contrast to inner accessors, outer accessors work directly on the given explicit argument rather than any of their lookup arguments. One more important kind of accessors is **delegation accessors**, which work on the delegation object of their argument.

Support for mixed types of accessors is needed and built into AmOS. There are for instance inner delegation accessors and outer delegation accessors. An example of slot that is accessed via an inner delegation accessor is the parent link of closures (illustrated on figure 5.3 on page 92). A similar (very important) example is the current context: the current context is a delegation slot that is accessed “from the inside” thanks to a current-context inner delegation accessor.

---

22 Recall section 5.2.5.
23 This context is introduced in section 5.1.2 on page 86.
24 Delegation objects are discussed in section 5.2.1 on page 91.
25 As explained in section 5.2.4 on page 97.
5.2. Message Resends

Explaining message resends is interesting for two reasons. Firstly, they constitute one more example of the use of lookup arguments. Secondly, resend methods are specialised on an unusual kind of context: the method activation context. We thus show that the implementation of such fundamental construct (resend) reuses the context-oriented machinery of AmOS.

Method activations are first-class objects that support extended functionality with respect to simple closure activations. In particular, a method activation understands the (resend) message, which has no explicit arguments, but has the current activation as implicit argument (as all messages do). However, as in the case of accessor methods, this implicit argument is not necessarily the right place to look for the information resend needs. Consider this use of resend inside a closure:

```
(defmethod receive ((call @phone-call) (phone @phone))
  (& () (resend)))
```

This version of receive returns a closure of arity 0, which will resend the message when invoked. When the resend message is sent inside the closure, it is looked up in the current activation, which is the closure’s. Since closures do not understand resend, method lookup continues “outwards” in the chain of lexical scopes (recall figure 5.3). The next activation found is that of receive. This activation does understand the resend message because it is a method activation; the resend method is thus invoked in response to the message, passing it the closure’s activation as implicit argument, and the method’s activation —where the resend method definition has been found— as lookup argument. The implementation of resend is actually interested in the latter, the method activation. Having access to the method activation, the implementation of resend can find out the necessary information to resend the message; such information is not available in a closure activation.

Thanks to the support of first-class activations, context-oriented programming and lookup arguments, resend can be defined easily as shown in snippet 5.1. Note that resend is defined in method activation context. More precisely, it is defined in the prototypical @method-activation, which belongs to the prototypical @method. Hence, quite naturally, resend makes part of the prototypical behaviour of method activations.

The use of lookup arguments in the implementation of resend is hidden in the (resender-activation) form (line 3 of snippet 5.1), which is defined as:

```
(lookup-argument (current-activation) 0)
```

The current activation of resend is obtained with (current-activation) and its first lookup argument is returned. This lookup argument corresponds to the

---

26 We find that this is not only technically, but also intuitively sound. We think of method activations as kinds of contexts in which method bodies execute. In this view of things, it makes perfect sense to define resend as behaviour that is available in method execution context.
The Ambient Object System

(defmethod resend ()
  (let* ((activation (resender-activation))
         (method (owner activation))
         (selector (selector activation))
         (arguments (arguments activation))
         (next-method
          (or (lookup-method selector arguments
                        :resender method)
              (error 'not-understood
                      :selector selector
                      :arguments arguments)))
  (invoke next-method selector arguments)))

Snippet 5.1: resend method definition.

method activation in which resend was found during method lookup.

Thanks to lookup arguments, the resend method needs not be physically
cloned for every new method activation, even though it makes integral part of
the activation’s behaviour. Such cloning would be prohibitive. There is exactly
one copy of the method in the system, whose specialiser remains constant (the
prototypical method activation), but thanks to lookup arguments, the method
is able to find the (fresh) method activation in which it has been invoked.

5.2.8. Targeted Resends

AmOS includes a construct called targeted resend. Targeted resends make
it possible to resend the current message to arbitrary applicable methods, in-
stead of only the next most-specific method as in the case of plain resends.
Targeted resends are useful to invoke methods deep in the delegation graph of
the specialisers of the currently executing method, when such methods cannot
be accessed by means of a simple resend.

Snippet 5.2 shows the definition of resend-as, the implementation of tar-
geted resends. The user provides the arguments in which method lookup should
start (line 1). On line 3, those explicit arguments are complemented with an
implicit one, the sender’s activation —as usual, the sender activation is always
the first, implicit argument of every message. The targeted method is looked
up using the complemented arguments provided by the user (line 7). Note that
target-method might exist, but it might not be part of the applicable method
list for the current invocation arguments. The user might provide whatever ar-
guments he wants and the only commonality between the currently executing
method and the chosen method will be their selector. To disallow this usage and
therefore enforce truly “resend” semantics, we impose that the method invoked
by resend-as must be applicable to the arguments of the currently executing
method (read on line 5), besides being applicable to the lookup-arguments
provided by the user. This rule is encoded in lines 8 and 9. If the chosen
5.2. Opening Up AmOS

(defmethod resend-as (lookup-arguments)
  (let ((activation (resender-activation)))
    (prepend-slot lookup-arguments (argument activation 0))
    (let* ((selector (selector activation))
           (arguments (arguments activation))
           (target-method
            (lookup-method selector lookup-arguments)))
      (if (member target-method
                   (find-applicable-methods selector arguments))
          (invoke target-method selector arguments)
          (error 'not-understood
                  :selector selector
                  :arguments arguments))))

Snippet 5.2: Definition of targeted resend.

method is valid, it is invoked with the arguments of the currently executing method, not the lookup arguments given by the user. This again enforces the “resending” nature of the operation. The provided lookup arguments are just a means of adjusting method lookup to target specific versions of methods, hence our coining of the term targeted resend.

To draw a parallel with a well-known language, targeted resends can be seen as a sort of multiple-dispatch cousin of the scope resolution operator denoted by two colons (::) in C++ (which is singly dispatched). On the dynamic languages front, resend-as bears resemblance to Filtered Dispatch [31] —a technique that also exploits the idea of having the lookup mechanism use arguments different from invocation arguments. A similar construct is also available in Slate.27

5.2.9. Bypass Resends

AmOS introduces a novel resend operation, similar in spirit to targeted resends: bypass resend. Bypass resends are a kind of message resend that “skips” given contexts (if they are active) when looking up the next most-specific method that should be executed. Snippet 5.3 shows the implementation of bypass resend. The main difference with respect to regular resending (shown in snippet 5.1) is that the next method is looked up in a context combination that does not include the given contexts (line 7). These contexts are thus “skipped”. Nonetheless, the chosen method is invoked in the (unchanged) context in which the bypass resend has been used (line 13). The context-oriented version of invoke receives three arguments instead of two; the third argument on line 9 is the context in which the closure will be executed.

All methods used in snippet 5.3 are explained in section 5.3. Section 7.2.2 on page 161 shows an example of the use of bypass resends.

5.2.10. Sealed Slots

Sealed slots are slots that cannot be altered indirectly by means of delegation. The only difference with respect to plain slots is that sealed slots have special “create-on-write” writer methods, instead of regular writer methods. An attempt to change the value of a sealed slot will result in invocation of the special writer method. If the object passed as actual argument is exactly the same object on which the writer is defined—that is, if actual argument and lookup argument coincide—the writer behaves as a normal writer: it just changes the value of the slot. However, if the actual argument is not the owner of the slot (i.e. if actual argument and lookup argument are different), the special writer method will create a new slot on the actual argument, associating to it a regular writer with the same selector, so that the new slot can be written normally. The sealed slot remains unchanged. This does not mean that sealed slots are constant: as explained previously, a sealed slot can be modified if written directly (not indirectly by delegation).

Objects can have a mixture of plain slots and sealed slots. When all slots are sealed, the object itself is said to be sealed. There is a prototypical @sealed object representing this kind of objects. The add-slot method is specialised on @sealed so that it adds sealed slots instead of plain slots. Hence, any object that delegates to @sealed will respond to add-slot by adding sealed slots.

The “prototypical prototype”, @prototype, delegates to @sealed. This way, slots added to prototypes cannot be altered by delegating objects, thus avoiding the prototype corruption problem [12]. A sample use of sealed slots is given in section 7.2.2 on page 163. There seems to be a similar mechanism in the Io language.\footnote{See http://www.iolanguage.com. The minimalistic style of Io’s documentation, though elegant, makes it difficult to fully assess the similarities.}
5.3. Main Protocols

This section describes the main API of AmOS. The purpose of this section is threefold. Firstly, it documents systematically the protocols that are used in the examples of this chapter, and in the validation cases of chapter 7 for the sake of completeness and understandability. Secondly, it succinctly shows the functionality exposed to programmers, thus serving as a quick overview of the tools AmOS puts in their hands. Thirdly, the API description serves as a complement to previous explanations that gives a more “programmatic understanding” of the object model discussed so far.

Only the most relevant protocols are explained, namely the Meta Object Protocol (MOP) and the Context Object Protocol (COP).\textsuperscript{29} Most methods have an annotation on the margin referring to the section that discusses their functionality in depth.

5.3.1. Meta-Object Protocol

AmOS exposes some of its underlying mechanisms to the user through a Meta Object Protocol (MOP), much like Smalltalk \[58\] and the Common Lisp Object System do \[70\]. The most important MOP methods are explained in this section, divided in two categories depending on whether they implement structural or behavioural reflection \[100, 51\].

**Structural MOP**

*extend object*

Creates a copy of the slots of object, but instead of also copying the behaviour as well, it places a delegation link from the extended object to the given object.

*extend-many objects*

Creates an empty object that delegates to extensions of the given objects. The extensions are made with *extend*.

*slot-value object (index @natural)*

Reads the slot with given index from object. This method is used for instance by accessor methods.

*delegates object*

Returns the delegation object of the given argument.

*linearise-delegates object*

Returns the linearisation of the delegation graph starting at the given object.

*make-closure names values parent code*

Returns a closure with a prototypical activation that has the given parent

\textsuperscript{29}A pun on Context-Oriented Programming \[65, 68\].
scope, and arguments with given names and prototypical values. The code is given as an s-expression.

**activation** (closure @closure)
Retrieves the prototypical activation of the given closure. 5.2.2

**arguments** (activation @activation)
Retrieves the arguments of the given activation. 5.2.2

**parent** (activation @activation)
Yields the parent object of the given activation. 5.2.2

**owner** (activation @method-activation)
Retrieves the method that owns the given activation. 5.2.3

**selector** (activation @method-activation)
Retrieves the selector associated to the given method activation; the result corresponds to the selector of the message that caused invocation of the method owning activation. 5.2.3

**Behavioural MOP**

**invoke** (closure @closure) arguments
Invokes the given closure, passing the given arguments array. 5.2.2

**invoke** (method @method) arguments
Analogous to the invoke method for closures. A separate invoke method is needed because the call protocol for method deals with dispatch information, with which closures are not concerned. Even so, the user needs not make the distinction: the most appropriate version of invoke will be used depending on the run-time type of the first argument.

**lookup-method** (selector arguments)
Looks up a method with the given selector that is applicable for the given arguments. The selector argument is not specialised on @symbol, because any object can serve as selector —its only important property for this purpose is its identity. 3.4

**send** selector arguments
Sends a message with given selector and arguments. For efficiency, the default implementation performs a native method lookup followed by a native method invocation —that is, it does not invoke the lookup-method and invoke methods described previously. However, a version of send that performs such invocations can be defined in reflective context; if that context is active, all message sends will use the reflective protocol.

**resend**
Invokes the successor of the currently executing method in the linearisation that was used to find such method. 5.2.7
resend-as lookup-arguments
5.2.8 Resends the current message to the method that is most specific for the given arguments, among all methods that are applicable to the current message.

5.3.2. Context Object Protocol
For clarity, we divide the protocol in different categories.

Behavioural reflection
The behavioural MOP presented previously is extended with two methods that deal with contexts.

invoke (closure @closure) arguments (context @context)
The closure is invoked with the given context as current context. The context that was current prior to invocation remains unchanged once the closure returns, whether the closure terminates normally or is aborted by a control transfer of some kind (e.g. an exception).

resend-bypassing-contexts contexts
5.2.9 Resends the current message to the next most-specific method for the current message, as if the message had been sent in a context where the given subcontexts were inactive.

Context creation and combination
The following methods facilitate the task of creating new contexts and combining already existing ones.

define-context name dependencies
Creates a new context object that delegates to each object in the given list of dependencies. A plain slot referring to the newly created context is added to the current context with the given name, so that the new context can be accessed using that name. The defcontext macro used in the examples expands to a define-context invocation.

combine-contexts contexts
4.3.2 Returns a context object representing the combination of the given list of super-contexts (i.e. more general contexts). The resulting combination is not activated, it is simply returned. If a combination of the given contexts had been formed before (irrespective of their order in the contexts list), the exact same combination object is returned.

combination-including combination contexts
4.3.2 This is and “additive” version of combine-contexts, which returns a com-

---

30 An inner accessor is created for the slot (recall section 5.2.6 on page 101). As discussed in section 5.2.1 on page 90, slots do not really have names per se, but the selector of the inner accessor is used as a name.
The combination of the contexts comprised in the already formed combination plus the contexts in the given list. As with combine-contexts, if a combination of the given contexts had been formed before, the exact same combination object is returned.

combination-excluding combination contexts

Reciprocal of combination-including.

Context queries

Two methods can be used to make queries about combinations:

find-combinations-including context

Returns a list of all context combinations that delegate to the given context. This is just a query method, it does not create new combinations.

find-combinations-excluding context

Reciprocal of find-combinations-including.

Two methods are enough to make enquiries about the current context:

current-context

Returns the current context object. This will be a combination object that delegates to all domain-specific subcontexts that are currently active.

active-p (context @context)

Determines whether the given context is active — that is, whether it is part of the current context delegation graph.

Current context redefinition

The following methods permit the manipulation of the current context in an “absolute way”, without taking active contexts into account. These methods are mostly used at development time, since defined methods must be specialised on very precise contexts, and not on the casual context combinations that might be active at that moment.

use-contexts contexts

Makes the combination of contexts the currently active one, irrespective of the contexts that were active prior to the call. This call thus completely overrides the current context combination.

(setf current-context) (context @context)

The (setf current-context) method sets the current context to the given context object. This method is the lowest-level mechanism to change the context, since it bypasses the context management framework provided by AmOS. The responsibility of maintaining system consistency is left entirely to the user.

31 In Common Lisp, the -p suffix of names such as active-p stands for “predicate”; it is analogous to the ? suffix typical of Scheme style.
Current context adaptation

In contrast to the “absolute” methods just presented, the following methods permit gradual context adaptation by adding and removing individual contexts from the current context graph. This way of manipulating the context is used mostly at run-time, to adapt behaviour according to detected changes in the surrounding physical and logical environment.

**switch-on (context @context)**

Combines the given context with the list of currently active contexts, and switches the current context pointer to that combination. This effectively renders the context active. Rather than be explicitly called by the user, the switch-on method is meant to be called by activate as explained below; switch-on is meant more as a hook method that the user can override to trap context changes (chapter 7 contains examples of such use). Unless the user wishes to overrule the context switch, the overriding version of switch-on should call resend to invoke the original version and make the context change effective.

**switch-off (context @context)**

Reciprocal of switch-on. Removes the given context from the list of currently active contexts, and switches the current context pointer to the combination of the remaining contexts. This will render the given context inactive.

**activate (context @context)**

If the given context is not already active, this method switches it on by invoking switch-on. If the given context is active already, the method does nothing at all. Hence, if a user-defined version of switch-on is applicable in the current context, the context change will be trapped by such user-defined method; on the contrary, if there is no actual context change because context was already active, user-defined hooks will not be invoked.

**deactivate (context @context)**

The reciprocal of activate; it calls switch-off to signal the effective deactivation of a context. If the context was inactive already, the method is a no-operation.

Context destruction

Transient contexts need to be removed from the system when no longer needed. One method is available for context removal.

**forget (context @context)**

Removes the given context from the system. The slots and methods that were defined in this context will no longer be available. Context combinations that depend on (delegate to) the given context are removed from the system as well.
6 Formal Semantics

This chapter presents the operational semantics of the Ambient Object System (AmOS). This formal specification is expressed as a context-sensitive term rewriting system, following Felleisen and Hieb’s approach [50]. An introduction is given in section A.1 on page 197.

The semantics is small-step: it describes in fine-grained detail the meaning of method creation, method invocation, parameter access, and so on. This implies that even the most basic operations take many computational steps to complete. Although seemingly an unnecessary complication, this property is essential for considering model extensions that have to deal with thread interleavings, such as transactions [83].

We have proof-tested the semantics of AmOS using PLT Redex, a domain-specific language for context-sensitive rewriting that provides a graphical browser for exploring reduction graphs [78]. PLT Redex helped us to detect grammar ambiguities and reduction rule inconsistencies. The figures in this chapter containing the grammar and reduction rules of our model have been generated automatically by translating the definition of the semantics from PLT Redex syntax.\(^1\)

6.1. Background

We start with a rough guide to the formalism partly borrowed from Matthews and Findler [77]. The semantics is defined via a relation on program terms, where the relation corresponds to a single step of an abstract machine. The relation is defined using evaluation contexts, namely terms with distinguished subterms in them, called holes, where the next step of evaluation occurs. A term \( e \) decomposes into an evaluation context \( E \) and another term \( e' \) if \( e \) is the same as \( E \) but with the hole replaced by \( e' \). We write \( E[e'] \) to indicate the term obtained by replacing the hole in \( E \) with \( e' \) and we write \( \square \) to indicate the hole.

As an example, consider the following reduction rule:

\[
\begin{align*}
PT[(send l_s l_1 \cdots)] & \rightarrow \\
PT[(invoke (lookup l_s (activation) l_1 \cdots) l_1 \cdots)]
\end{align*}
\]

Subscripts are used to distinguish different occurrences of a same non-terminal. The dots \( \cdots \) stand for zero or more repetitions, like a Kleene star. The rule

---

\(^1\)The same approach and tool were used to specify a formal semantics for Scheme R\(^5\)RS [77] and R\(^6\)RS (Revised\(^6\) Report on the Algorithmic Language Scheme), see http://www.r6rs.org.
Formal Semantics states that, if a term can be decomposed into a context $PT$ and a subexpression of the form $(\text{send } l_s l_1 \cdots)$ where $l_s, l_1 \cdots$ are terms matching the $l$ non-terminal of the language’s grammar, then the subexpression can be replaced by the term $(\text{invoke } (\text{lookup } l_s (\text{activation}) l_1 \cdots) l_1 \cdots)$ where the occurrences of $l_s$ and $l_1 \cdots$ in the right-hand side of the rule correspond to the terms that matched the left-hand side of the rule.

Given the previous rule and a suitable definition for context $PT$ (shown in figure 6.2), the following would be a valid execution step of the abstract machine:

$$(\text{return } (\text{send } + 1 2)) \rightarrow (\text{return } (\text{invoke } (\text{lookup } + (\text{activation}) 1 2) 1 2))$$

The whole expression on the left-hand side can be decomposed as a context $PT$ and a subexpression $(\text{send } + 1 2)$. The subexpression can thus be rewritten as prescribed by the right-hand side of the rule. Note that terms which are not part of the subexpression are left untouched (i.e. the $\text{return}$ invocation). Continuing from this point, the next step of the machine will be the substitution of the leftmost-innermost term — that is $(\text{activation})$ — by some other term. Section A.1 on page 197 provides more background on context-sensitive rewriting systems.

6.2. Abstract Grammar

We take a structural approach to defining the semantics: our operational semantics is based on reduction rules that follow the abstract syntax of the language [122]. Hence, the first step towards specifying the semantics is defining the abstract syntax — that is, the valid expressions that the abstract machine can recognise.

6.2.1. Program syntax

The left-hand side of figure 6.1 presents the abstract syntax of AmOS in Extended Backus-Naur Form (EBNF). As mentioned previously, the dots $\cdots$ stand for zero or more repetitions of the leftward term. The right-hand side lists the syntactic sets defined by each production. The $\text{Number}$ and $\text{Symbol}$ sets are parameters of the model, and have thus no explicit definition. An explanation of the non-terminals follows.

**Programs** The $p$ non-terminal represents valid programs. Every program consists of a store $\sigma$ containing bindings and a thread pool $\tau$ of running threads.

**Threads** Threads are pairs $(l \ e)$ of a label $l$ denoting an evaluation environment and an expression $e$ that is to be evaluated by the thread in that environment.
6.2. Abstract Grammar

\[ p ::= ((σ b \ldots) (τ t \ldots)) \quad p \in \text{Program} \]
\[ b ::= (l o) \quad b \in \text{Binding} \]
\[ t ::= (l e) \quad t \in \text{Thread} \]
\[ o ::= ((r \ldots) (l \ldots) (l \ldots) (e \ldots)) \quad o \in \text{Object} \]
\[ r ::= (l l l) \quad r \in \text{Role} \]
\[ e ::= l \mid i \mid (f e \ldots) \quad e \in \text{Expression} \]
\[ i ::= (λ (a \ldots) e \ldots) \mid (η (a \ldots) e \ldots) \quad i \in \text{Literal} \]
\[ a ::= l \mid (e) \mid (l e) \quad a \in \text{Formal} \]
\[ l ::= n \mid q \quad l \in \text{Label} \]
\[ n \in \text{Number} \quad q \in \text{Symbol} \]
\[ f \in \text{Symbol}^* \]

Figure 6.1.: Program abstract syntax.

Stores, bindings and labels  Stores are sequences of bindings \((l o)\) relating a label \(l\) to an object \(o\). Labels represent store addresses; they can be either symbols or numbers.\(^2\) A label should appear at most once on the left-hand side of the different bindings of a store. A program could break this uniqueness property and still be well-formed according to the grammar in figure 6.1. However, for every program that starts with a correct initial store, the uniqueness property will be maintained throughout its execution, because there is no operation to change existing binding labels, and all newly introduced labels are fresh (as observed in the specification of the clone primitive in figure 6.4 on page 121).

Numbers  We use numbers in an unusual way, which is inspired by the object-oriented nature of the model. Most formal models regard numbers as basic values, rather than valid store locations. Hence, in those models, numbers appear as values on the right-hand side of store bindings (for example in Matthews and Findler [77]). In our model, numbers appear as labels on the left-hand side, and have a full-fledged object associated to them on the right-hand side of the binding. That is to say, numbers are objects in our model, and the number itself is just the label of the object in the store.

Symbols  We use the notion of symbol rather than variable to stress their role of mere names or designators in the model. For example, the symbols \(+\) and send can be used to name primitive operations of the language, even if they are not associated to objects in the store. The concept of “variable” appears nowhere in our model, though we could say that a variable is any label that has a binding in the store of a given program (i.e. when there is something that can effectively variate associated to that name). This notion of variable is relative to each particular program — hence our preference for the notion of symbol.

\(^2\)The distinction is actually immaterial in the syntactic formalism we are using. Numbers are syntactic entities devoid of any (mathematical) meaning, just as symbols are. This is in keeping with the untyped nature of the model.
**Objects** Objects, represented by the non-terminal $o$, have the following four components:

Roles – Roles are a concept originally found in the Prototypes with Multiple Dispatch (PMD) model [93]. A role is a triplet composed of a selector that identifies a collaboration in which the object can take part, an argument position telling which place is occupied by the object in the collaboration, and a method that prescribes the collaboration.

Delegations – Delegations are acquaintances to which requested behaviour is relayed if not implemented locally by the host object. Delegations are used by the method lookup mechanism explained in section 6.4.8.

Slots – Slots store the “plain” acquaintances an object needs for normal operation. Hence, leaving the behavioural part aside, objects are essentially arrays of references to other objects.

Body – Every object in the model can (potentially) contain an executable body made up of a sequence of expressions. Objects have thus a dual nature: they are at the same time data entities and executable entities. The data portion can be used as an environment to evaluate the executable portion. This design leads to straightforward definition of closures.

**Expressions** There are three kinds of valid expressions $e$: labels $l$, literals $i$ and primitive invocations $(f \; e \; \cdots)$. The latter consists of a label identifying the primitive and a sequence of arguments. Literals are objects that have a special notation in the language — a mere syntactic facility to avoid otherwise lengthy sequences of primitive operations to construct the same objects denoted by those literals. These literals are inessential to the model, albeit they make the notation lighter. There are two kinds of them as shown in the definition of the $i$ non-terminal: $\lambda$-expressions and $\eta$-expressions. These are used to create closures and methods with the given arguments and body. The $\lambda$-expressions are end-user constructs akin to $\lambda$-abstractions from $\lambda$-calculus, except that they receive an implicit parameter (due to the context-oriented nature of the model); $\eta$-expressions are similar except that all parameters are explicitly defined, and are most often generated internally rather than given by the user. The reduction of these literals to primitive forms is explained in sections 6.4.5 and 6.4.6.

---

3 We like to think here of a theatrical analogy inspired by the Actor model [2]: objects are the actors in the play, selectors are the name of acts, argument positions tell what character the object is able to play in the act, and methods are the script. In this model behaviour is described from an external, third-person perspective, instead of a first-person perspective as in singly dispatched languages. Following our analogy, the script is the same for all actors playing a role in the act.

4 Many models privilege a particular data structure. In the case of AMOS it is object arrays; Lisp is based on lists (made up of cons cells) [79]; Python is based on associative arrays; Oz is based on the similar notion of records [118].

5 In a way, the design is aimed at unifying object orientation and functional programming. The fundamental concept in the former paradigm is the object, whereas in the latter it is the function. In our model, they are realised as one entity.
6.2. Abstract Grammar

The $a$ non-terminal describes the formal arguments of $\lambda$-expressions and $\eta$-expressions: a formal can be an argument name $l$, a prototypical argument value expression $(e)$, or a combination of both argument name and prototypical value expression $(l\ e)$. The result of evaluating the expression $e$ is used as prototypical value of the argument. Arguments with only a name $l$ have the prototypical object as value (defined later on); arguments with only a value expression $(e)$ are anonymous.

**Basic non-terminals** The basic non-terminals of figure 6.1 are numbers $(n)$, symbols $(q)$, and a reduced set of symbols$^6$ $(f)$ used to avoid the default evaluation semantics for certain kinds of expressions (including for instance the literals $\lambda$ and $\eta$). As mentioned before, the Number and Symbol syntactic sets are parameters of the model.

### 6.2.2. Evaluation contexts

Figure 6.2 presents the grammar for evaluation contexts in EBNF on the left-hand side, and the defined syntactic sets on the right-hand side. The brackets $[]$ are integral part of the set name, used as a “pun” or reminder that the set contains contexts.$^7$ The name in brackets indicates the kind of entity matched by the contexts in the set. For instance, contexts in $[Activation]$ match activations, those in $[Binding]$ match bindings, and so forth.

As illustrated in figure 6.2, contexts can be divided in two main groups, following the syntactic structure of programs. They can be about the part of a program related to execution, or about the data portion of a program. The following presentation follows this categorisation.

---

$^6$The reduced set of symbols is obtained by subtracting from Symbol all symbols appearing in the grammar productions shown in figures 6.1 and 6.2.

$^7$This naming scheme is lighter than having names such as ProgramThreadContext, ProgramActivationContext, etc.
Contexts related to execution

The contexts in the “execution” category, explained next, help decomposing the thread portion of programs.

Program thread context  $PT$ decomposes expressions irrespective of the store, taking into account the thread context ($T$) only.

Program activation context  $PA$ decomposes expressions regardless of the store, but in contrast with $PT$, $PA$ takes the activation context ($A$) into account.

Thread context  $T$ decomposes expressions found in individual threads, irrespective of the particular activation that is current for the thread —that is, irrespective of the left-hand side component of the pair ($l\ e$) representing a thread.

The thread context $T$ is defined such that any number of threads $t$ can appear to the right and to the left of the matched subexpression ($l\ E$). This models non-determinism in the choice of threads: the abstract machine can choose any thread among the reducible ones for the next reduction step. The execution of different threads is thus interleaved. Actually, the abstract machine chooses all possible paths, and reduces them in parallel. At the end of execution, a list of resulting programs is obtained. If the program is deterministic, the result list will contain only one element.

Since the semantics is small-step, thread interleavings can happen with the finest possible level of granularity. For instance, one thread can make a step while another one is in the process of entering a method (the invocation of methods counts many steps, as explained in section 6.4.5). This can come as an advantage when studying the properties of the model with respect to parallelism [83].

Activation context  $A$ focuses on individual threads as $T$ does, but it takes the whole thread ($l\ e$) into account rather than the expression part only. It does so by having a hole in thread position (whereas $T$ has a hole in the expression position of a thread). Context $A$ expresses reductions that depend on the current activation and the current expression of any thread.

The name activation is a shortcut for activation record (often called also a stack frame). Activations are the environments that hold the actual arguments and local variables during a closure or method invocation. In our model, activation records are first-class objects, and therefore, a thread’s activation can be specified simply by a label designating an object in the store.

Expression context  $E$ matches individual expressions. Note that a left-to-right evaluation order is enforced in the evaluation of invocation arguments by permitting only labels $l\cdots$ to the left of $E$, and whole (yet unevaluated) expressions to the right. With this definition of $E$, evaluation occurs in a
leftmost-innermost manner. Intuitively this means that a primitive’s arguments are always reduced before the primitive itself is. The base semantics specifies thus a strict, applicative order evaluation strategy [1].

The write-code primitive has a different evaluation strategy: only the first argument is evaluated, and the rest are left untouched. A call to write-code is not performed in a strict fashion thanks to the use of the $f$ non-terminal, which precludes the occurrence write-code.

**Contexts related to storage**

The contexts in the “storage” category help decomposing the store portion of programs, as explained next.

**Binding context** $B$ matches store bindings. Most often reductions depend both on the label and the associated object of the binding. Hence, there are no more fine-grained contexts to match either of the two individually.

Since bindings $b$ appear to the left and to the right of the hole in context $B$, the context selects any binding in the store irrespective of its position. This is the same idiom used to choose threads in context $T$. However, bindings are always selected by specifying a label in context $B$,$^8$ whereas threads are not selected by their current activation in context $T$.$^9$ Hence, unlike threads, the matching of bindings in the store is deterministic.

**Object structure contexts** $R$, $D$, $S$ and $C$ occur respectively in the role, delegation, slot and body portions of objects. They can be used to match any of those structural constituents of an object.

Thanks to these contexts, reduction rules are oblivious of the internal structure of objects, as will become apparent in section 6.4. If the object format changed, the rules would need no modification whatsoever, provided that the object structure contexts are adjusted appropriately.

### 6.3. Canonical Store

Any non-trivial program needs to create the objects it needs for normal operation. However, in our semantic model there is no ex nihilo creation of objects [85]. The only means to create an object is by cloning an existing one. Therefore, programs must be launched with a store that includes at least a few initial prototypes that can be cloned. Furthermore, these initial prototypes should exhibit suitable behaviour that gives them some meaning beyond mere existence. Hence, we define a canonical store that comes pre-loaded with

---

$^8$In all reduction rules shown in section 6.4, $B$’s hole is filled with a term $(l\;o)$ where $l$ is a known, given label. Hence, only bindings with that same label will match.

$^9$This is enforced by the definition of $T$: the hole occurs in subexpression position (via $E$), making it impossible to fix the thread’s activation label as is done for the binding label in $B$. 

a number of prototypes and their associated behaviour, explained in this section. This store constitutes the canonical environment in which programs are launched.

The canonical store is built starting from an initial store, shown in figure 6.3. This initial store makes a number of basic properties apparent. Firstly, the prototypical closure delegates to the lobby. The lobby is thus the prototypical scope of closures: it plays the role of the “top level” found in languages such as Python, Ruby and the various Lisp dialects —basically, languages with a Read-Eval-Print Loop (REPL). To make the delegation link accessible, suitable accessor methods are defined as part of the construction process of the canonical store:

```
(define-delegation-accessors closure parent 0)
```

The semantics of this operation is discussed in section 6.4.2, but briefly put, it defines reader and writer methods with selectors parent and parent! that enable manipulation of the delegation slot at position 0 of the prototypical closure.

The second apparent property in figure 6.3 is that the lobby delegates to a context object. This delegation link can be modified in order to attain dynamic adaptation to context.\(^{10}\) To allow access to the delegation link, accessor methods are defined as shown previously the scope of closures:

```
(define-delegation-accessors lobby current-context 0)
```

The “current context” is thus the parent scope of the lobby, and can be read and written by sending current-context and current-context! messages.

The third property illustrated by figure 6.3 is the nature of numbers (in this case, number 0). Numbers are simply the label of an object in the store; this object delegates to the number prototype. The dots (... ) stand for all numbers programs can possibly use. Hence, in this semantic model the store is (countably) infinite.

Some reduction rules are based on the prototypes and behaviour just de-
6.4. Reduction Rules

This section presents the reduction rules of the semantics divided by categories. The data-oriented reductions that manipulate the structure of objects are presented first, followed by reductions dealing with object behaviour.

6.4.1. Storage

The reductions dealing with object storage are shown in figure 6.4. The clone primitive adds a new binding to the store and returns its label. The new binding contains a shallow copy of the original object—that is, a textual copy of the roles, delegations, slots and body (which are not cloned recursively). The clone primitive is the only means of adding new bindings to the store. The cloning rule is a first sample use of context $T$ to match the next expression to be executed by some thread (chosen non-deterministically).

The second operation, array, is simply a shortcut for the creation of an object with given slots $l\cdots$. This shortcut increases the legibility of a few reduction rules shown later on. An array form is reduced to the cloning of the empty prototype, to which the given slots are subsequently added starting at position 0. The semantics of insert-slots is specified in section 6.4.2. In contrast with $T$, context $PT$ is used to express context-insensitive transformations (which do not depend on surrounding terms such as the store).

6.4.2. Slots

The model provides primitives to add, read, write and remove slots from objects. These are shown in figure 6.5. The rule side conditions overload the mathematical notation of set cardinality: $|l_1\cdots|$ stands for the number of elements that match the $l_1\cdots$ pattern.

In all slot manipulation rules, context $B$ refers to a particular binding with label $l_t$ in the store. The slots of the object are matched thanks to a nested $S$ context (inside $B$ context). This use of context $S$ makes the rules oblivious of the particular format of objects. If we decided to change the object format—for instance, to place the slots earlier in the object representation, or embed
slot insertion
\[(B(\langle l_t, S[\langle l_1 \ldots l_j \ldots \rangle] \rangle) T[(\text{insert-slots} \langle l_t \rangle n_i \langle l_i \ldots \rangle)] \rightarrow (B(\langle l_t, S[\langle l_1 \ldots l_i \ldots l_j \ldots \rangle] \rangle) T[l_t])\]
provided that \(|l_1 \ldots | = n_i

slot access
\[(B(\langle l_t, S[\langle l_1 \ldots l_i l_{i+1} \ldots \rangle] \rangle) T[(\text{read-slot} \langle l_t \rangle n_i)] \rightarrow (B(\langle l_t, S[\langle l_1 \ldots l_i l_{i+1} \ldots \rangle] \rangle) T[l_i])\]
provided that \(|l_1 \ldots | = n_i

slot modification
\[(B(\langle l_t, S[\langle l_1 \ldots l_i \ldots l_j \ldots \rangle] \rangle) T[(\text{write-slots} \langle l_t \rangle n_i l_j \ldots)] \rightarrow (B(\langle l_t, S[\langle l_1 \ldots l_j \ldots l_k \ldots \rangle] \rangle) T[l_t])\]
provided that \(|l_1 \ldots | = n_i \land |l_i \ldots | = |l_j \ldots |

slot removal
\[(B(\langle l_t, S[\langle l_1 \ldots l_i \ldots l_{i+c} \ldots \rangle] \rangle) T[(\text{remove-slots} \langle l_t \rangle n_i n_c)] \rightarrow (B(\langle l_t, S[\langle l_1 \ldots l_{i+c} \ldots \rangle] \rangle) T[l_t])\]
provided that \(|l_1 \ldots | = n_i \land |l_i \ldots | = n_c

Figure 6.5.: Slot manipulation reductions.

them in some deeper structure inside the object— the rules would need no modification at all; only the definition of context S would need adjustment.

Besides decomposing the store by means of B and S, the rules consider also the next subexpression that is to be executed by a thread, via the T context. The four possible expressions are invocations of the insert-slots, read-slot, write-slots and remove-slots primitives. All take as first parameter the label \(l_t\) of the object matched in B context. All take an index \(n_i\) of the slot at which the requested operation takes place. Finally, all return \(l_t\) as result, except for read-slot which is reduced to the value of the slot at position \(n_i\).

For brevity, we have omitted from figure 6.5 the reductions dealing with errors in slot manipulation (e.g. attempting to read a nonexistent slot); these are shown in figure A.2 on page 199.

The rules dealing with manipulation of delegations (i.e. addition, accessing, modification and deletion), though not identical, are similar in spirit to the ones presented in this section for manipulation of slots. These rules are shown in figure A.3 on page 199.

6.4.3. Roles

The reduction rules for role manipulation are shown in figure 6.6. Instead of referring to slots through S as in section 6.4.2, these rules use context R, nested in context B, to refer to the roles of the object in the binding matched by B. The side condition of the first rule, role addition, can be read as: “unless a role with the same selector, position and method exists in the object”. Note that this proposition specifies the conditions of applicability of the role addition rule, but the rule defines no semantics if its condition is not satisfied. A separate rule
role addition
(B[lt R[(l1 n1 l1 · · · )] ] T[(add-role lt ls np lm) ] ) →
(B[lt R[(l1 n1 l1 · · · (ls np lm) )] ] T[l])

unlesst (s p m) ∈ (l1 n1 l1) · · · : s = ls ∧ p = np ∧ m = lm

existent role addition
(B[lt R[(l1 n1 l1 · · · (ls np lm) (lk npk mk) · · · )] ] T[(add-role lt ls np lm) ] ) →
(B[lt R[(l1 n1 l1 · · · (ls np lm) (lk npk mk) · · · )] ] T[l])

role removal
(B[lt R[(r1 · · · (ls np lm) r2 · · · )] ] T[(remove-role lt ls np lm) ] ) →
(B[lt R[(r1 · · · r2 · · · )] ] T[l])

nonexistent role removal
(B[lt R[(l1 n1 l1 · · · )] ] T[(remove-role lt ls np lm) ] ) →
(B[lt R[(l1 n1 l1 · · · )] ] T[l])

unlesst (s p m) ∈ (l1 n1 l1) · · · : s = ls ∧ p = np ∧ m = lm

Figure 6.6.: Role manipulation reductions.

read current activation
PA[lt E[(activation)] ] → PA[lt E[l]]

change current activation
PA[lt E[(activation! l2)] ] → PA[l2 E[l2]]

Figure 6.7.: Reductions related to threads.

is needed for cases in which a role already exists. The existent role addition
rule specifies that an object cannot have repeated roles. If a role exists, adding
the exact same role again is a no-operation. Rules role removal and nonexistent
role removal are complementary in a similar way.

6.4.4. Execution

Having explained the data-oriented reductions of the model, we move on to
reductions related to object behaviour. We start with basic rules needed to
define the execution semantics of closures and methods.

The definitions of activation and activation! are shown in figure 6.7.
These primitives respectively read and modify the activation of the thread in
which they are being executed. Both rules use the program activation (PA)
context to read and write the label denoting the activation that is current for
their thread. Actually, these are the only two rules dealing with PA context
—all the rest rely on invocations of activation and activation! instead of
manipulating the current activation directly.

Figure 6.8 shows the meaning of the seq construct used to sequence execution
of expressions. The argument expressions are evaluated from left to right in
the usual applicative order (explained in section 6.2.2 on page 119), before their
result is passed to seq; the seq primitive simply returns the last result.
return last result
\[ PT[(seq\ l_1 \cdots l_n)] \rightarrow PT[l_n] \]

Figure 6.8.: Control flow reductions.

**implicit argument addition**
\[ PT[(\lambda (a \cdots e \cdots)] \rightarrow PT[(\eta ((\text{send current-context})) a \cdots e \cdots)] \]

**anonymous argument completion**
\[ PT[(\eta (a \cdots (e_v) e \cdots)] \rightarrow PT[(\eta (a \cdots (\text{anonymous} e_v)) e \cdots)] \]

**named argument completion**
\[ PT[(\eta (a \cdots l) e \cdots)] \rightarrow PT[(\eta (a \cdots (l \text{ object})) e \cdots)] \]

**named argument substitution**
\[ PT[(\eta (a \cdots (l_{ak} e_{vk}) e_b \cdots)] \rightarrow PT[(\text{define-accessors} (\eta (a \cdots (\text{anonymous} e_{vk})) e_b \cdots) l_{ak} | a \cdots)] \]
unless \( l_{ak} = \text{anonymous} \)

**anonymous argument substitution**
\[ PT[(\eta (a \cdots (\text{anonymous} e_a)) e_b \cdots)] \rightarrow PT[(\text{insert-slots} (\eta (a \cdots e_b \cdots) | a \cdots | e_a)] \]

**trivial closure substitution**
\[ PT[(\eta () e \cdots)] \rightarrow PT[(\text{write-code} (\text{clone closure}) e \cdots)] \]

**code writing**
\[ (B[(l\ C[(e_b \cdots)])])\ T[(\text{write-code} l\ e_{nb} \cdots)]) \rightarrow (B[(l\ C[(e_{nb} \cdots)])] )\ T[l]) \]

Figure 6.9.: Reduction rules for closure literals.

### 6.4.5. Closures

In this model, the behavioural unit is the object. Recall from figure 6.1 on page 115 that every object can have a body. Closures are objects that have been specially configured for execution of that body. A closure is basically a parametrised evaluation environment for the body it encloses. Further, closures are constructed such that they delegate to a parent object, meant to be the enclosing scope of the closure. The parent object will respond to the messages that are not understood locally in the closure.

**Closure construction**

There are two different literal representations for closures in the grammar shown in figure 6.1; we call the version with \( \lambda \) a user closure, whereas the version with \( \eta \) is a system or normalised closure. Both forms are just syntactic sugar for otherwise lengthy sequences of primitives that would create the same objects. Figure 6.9 shows the rules that reduce these literals to their primitive form.

In the implicit argument addition rule, \( \lambda \) forms are transformed to their \( \eta \) equivalent by prepending a formal argument. This implicit addition of an argument is the only difference between \( \lambda \) and \( \eta \); all other rules are expressed in
6.4. Reduction Rules

terms of $\eta$. The added argument is anonymous, and has as prototypical value
the current context. Since the prototypical arguments of closures are taken
as specialisers when used for method definition, this means that methods are
specialised on their context of definition. The current context is read by sending
a `current-context` message.\(^\text{11}\) Thanks to suitable accessor methods\(^\text{12}\) defined
on the lobby, the result of the message send will be the invocation of a reader
method. This method will return the object to which the lobby is currently
deleagating (recall section 6.3), which by definition is the current context.

The next four rules in figure 6.9 work on the rightmost argument of $\eta$ literals.
The information contained in the argument specification is processed by the
rules and the argument is then discarded. The rules work thus progressively
towards the left until no argument remains.

In the first argument-processing rule, **anonymous argument completion**, the
given argument value expression is tagged as **anonymous**. In the **named argu-
ment completion** rule, **object** is added as prototypical argument value if a value
has not been given explicitly. These two “completion” rules are thus meant to
extend argument declarations to canonical (**name value**) form.

Named arguments are processed in the **named argument substitution** rule by
defining corresponding accessor methods, so that arguments can be accessed
by sending messages.\(^\text{13}\) The **define-accessors** primitive is specified in sec-
tion 6.4.7. Having defined suitable accessors, the named argument is replaced
by an **anonymous** argument, since the name is of no use anymore. The argu-
ment name is an integral part of the $\eta$ syntactic facility, and is meaningless in
the core model where all $\eta$ expressions have been translated to their primitive
form.

Anonymous arguments are processed by the **anonymous argument substitu-
tion** rule. The prototypical argument value $e_a$ is added to the closure by means
of **insert-slots**. The slot position is the argument’s position $|a\cdots|$ (i.e. the
amount of preceding arguments). Hence, the first argument will be at slot po-

tion 0, the second argument at slot position 1, and so on. Having processed
all relevant information from the argument specification, the rule discards the
argument, thus making progress towards the left.

Once no arguments are left, the **trivial closure substitution** rule replaces the
$\eta$ literal by a clone of the prototypical closure, substituting the original body
by the body of the given $\eta$ form thanks to a **write-code** invocation.

The final **code writing** rule specifies the behaviour of the **write-code** primit-
itive. The given body $e_{nb}\cdots$ overrides the old body $e_b\cdots$ of object with label
$l$. In the case of $\eta$ forms, the old body is the prototypical closure body (which
is empty). This rule does not actually make part of the $\eta$-reduction rule set
—it is a generic rule to write the body part of an object, and could be used for
purposes other than reducing $\eta$ forms. However, the reduction of $\eta$ forms

\(^{11}\) Section 6.4.8 specifies the semantics of **send**. Note that the semantics does not have syn-
tactic sugar for message sending — messages need to be sent explicitly.

\(^{12}\) Section 6.4.7 specifies the semantics of accessor methods.

\(^{13}\) Contrast this with the $\lambda_v$-calculus, where the association of actual to formal arguments is
made by means of variable substitution (i.e. the $\beta_v$ rule) [122].
constitutes its main use.

**Closure invocation**

Figure 6.10 shows the semantics of closure invocation. In the closure invocation rule, the given closure $l_i$ is cloned, so that the code is executed in a fresh environment and can be re-entrant, instead of potentially modifying the original closure.\(^\text{14}\) The write-slots invocation copies the actual arguments to the freshly created closure. Note that the current activation is read by means of an activation invocation and prepended as an implicit argument before the explicit arguments $l_j \ldots$.\(^\text{15}\) The activation! call then changes the current thread activation (i.e. the current evaluation environment) to the fresh, initialised copy of the closure. After changing the current activation, the expressions $e \ldots$ contained in the body of the invoked closure are executed. These expressions will be evaluated under the fresh activation that has been created and activated for that purpose.

Rule closure termination in figure 6.10 specifies the way a closure’s execution is finished. The invoking activation that was passed as first (implicit) argument is read from the current activation (as shown in rule read invoking activation), and set —by means of activation!— to be the current one. The argument given to the return primitive, placed at the end of seq, will be the result left after exiting the closure environment.

### 6.4.6. Methods

Figure 6.11 shows the rules for method definition. The arguments of define-method in rule method definition are a selector and a closure. Defining a method means installing the closure on each argument specialiser, using the given selector $l_s$. The argument specialisers are the prototypical arguments of the closure, which are stored as slots $l_1 \cdots$ in the closure $l_m$ (recall section 6.4.5).

\(^\text{14}\) As an optimisation for “throw-away” closures that are invoked just once and are not re-entrant, the cloning could be omitted.

\(^\text{15}\) The implicit formal argument corresponding to this implicit actual argument is added to the closure by the implicit argument addition rule of figure 6.9.
6.4. Reduction Rules

method definition
\[(\sigma b_1 \cdots (l_m S[(l_1 \cdots)]) b_2 \cdots) T[(\text{define-method } l_s l_m)] \rightarrow
(\sigma b_1 \cdots (l_m S[(l_1 \cdots)]) b_2 \cdots) T[(\text{install-method } l_m l_s l_1 \cdots)]\]

install method
\[PT[(\text{install-method } l_m l_s l_1 \cdots l_k)] \rightarrow
PT[(\text{seq} (\text{add-role} l_k l_s | l_1 \cdots | l_m) (\text{install-method } l_m l_s l_1 \cdots))]\]

classend of method install
\[PT[(\text{install-method } l_m l_s)] \rightarrow PT[l_m]\]

Figure 6.11.: Method reductions.

As specified by rule install method, installing a method means adding a role to each argument specialiser (so that each specialiser knows how to play its role in the collaboration). This operation “gives the knowledge” to each specialiser on how to behave when collaboration \(l_s\) is requested to the group of objects \(l_1 \cdots\). The definition of install-method is recursive. Rule end of method install defines the base case of the recursion.

It follows from these rules that defining a method that has no argument specialisers is a no-operation. Methods have to be known by some object to be accessible at all. Furthermore, in this model a method cannot be invoked “out of the blue”: the user must have a reference to all necessary argument specialisers to invoke it, or to objects that delegate to those specialisers. Should the user lack access to the right acquaintances, there is no ambient authority from which he could gain access to arbitrary methods [81].

As a final remark, note that methods do not have an inherently associated selector: methods are not named per se. The selector is a property of the connection between the specialisers and the method by means of roles. In particular, the model does not preclude definition of a same method with different selectors or specialisers. This permits optimisations in the implementation, such as reusing accessor methods.

6.4.7. Accessors

The \(\lambda\)-calculus and its extensions are based on the \(\beta_v\) variable substitution rule for application of \(\lambda\)-abstractions (shown in figure A.1(b) on page 197). Our model is unlike most others in this regard. In our model, there is no such thing as variable substitution. Closure arguments (and hence, also method arguments) are accessed by sending messages, which result in the invocation of accessor methods. Besides sticking to a strict message-passing semantics, this design avoids variable captures.

The rules for creating and defining accessor methods are shown in figure 6.12. The define-accessors primitive is a shortcut for the installation of both a reader and a writer method for slot at position \(n_i\) of object \(l_t\). The reader is installed with selector \(l_n\). In the case of the writer, the selector is transformed by means of a \(W\) meta-function —otherwise, if the writer used the same selector,
define accessors
\[ PT[(\text{define-accessors } l_t l_n n_i)] \rightarrow PT[(\text{seq } (\text{install-method } (\text{reader-method } n_i) l_n l_t) (\text{install-method } (\text{writer-method } n_i) \ W[l_n] l_t \text{ object}) l_t)] \]

reader method
\[ PT[(\text{reader-method } n_i)] \rightarrow PT[(\eta \text{ (anonymous) } (\text{read-slot } \text{invoker} \ n_i))] \]

writer method
\[ PT[(\text{writer-method } n_i)] \rightarrow PT[(\eta \text{ (anonymous anonymous) } (\text{write-slots} \ \text{invoker} \ n_i (\text{read-slot } \text{activation} \ 1)))] \]

Figure 6.12.: Accessor reductions.

define outer accessors
\[ PT[(\text{define-outer-accessors } l_t l_n n_i)] \rightarrow \]
\[ PT[(\text{seq } (\text{define-method } l_n (\text{outer-reader-method } n_i l_t)) \ (\text{define-method} \ \W[l_n] (\text{outer-writer-method } n_i l_t)) l_t)] \]

outer reader method
\[ PT[(\text{outer-reader-method } n_i l_t)] \rightarrow \]
\[ PT[(\lambda ((\text{target } l_t)) (\text{read-slot} \ (\text{send target}) \ n_i))] \]

outer writer method
\[ PT[(\text{outer-writer-method } n_i l_t)] \rightarrow \]
\[ PT[(\lambda ((\text{target } l_t) \text{ value}) (\text{write-slots} \ (\text{send target}) \ n_i \ (\text{send value})))] \]

Figure 6.13.: Outer accessor reductions.

it would override the reader. \( \mathcal{W} \) is a parameter of the model. The particular definition we use in the semantics returns a symbol whose name is the result of appending an exclamation sign \(!\) to the name of its argument. For example, \( \mathcal{W}[x] = x! \).\(^{16}\)

Two primitives \texttt{reader-method} and \texttt{writer-method} in rules \texttt{reader method} and \texttt{writer method} create generic reader and writer methods for a slot at the given position \( n_i \). An \( \eta \)-form is used to generate the corresponding implementation of the accessor. The \( \eta \)-form uses anonymous arguments to avoid the (infinitely) recursive creation of accessors for the arguments of the accessor. The body of a reader simply reads the slot at position \( n_i \) of the invoker; the body of a writer modifies the slot.

The accessors just defined are \textit{inner accessors}. The model features also \textit{outer accessors}, shown in figure 6.13. Thanks to the inner accessor machinery already in place—which makes it possible to read the arguments of \( \lambda \) forms—the definition of outer accessors is simpler. Outer accessors can be defined as plain methods, by means of \texttt{define-method} and \( \lambda \) forms.

In rules \texttt{outer reader method} and \texttt{outer writer method}, the \texttt{target} argument is specialised on object \( l_t \) for both readers and writers, so that the object understands the accessor-invoking messages it receives. For writers, the \texttt{value}

\(^{16}\)In Ambience (which has Self-like syntax [115]), \( \mathcal{W} \) appends a colon (:) to writers; in AMOS (Common Lisp syntax [98]), it makes a “\texttt{setf}” name, for instance, \( x \) becomes \( \texttt{(setf } x) \).
message sending
\[PT[\langle \text{send} \ l_s \ l_1 \ldots \rangle] \rightarrow PT[\langle \text{invoke} \ (\text{lookup} \ l_s \ (\text{activation}) \ l_1 \ldots) \ l_1 \ldots \rangle]\]

method lookup
\[PT[\langle \text{lookup} \ l_s \ l_1 \ldots l_n \rangle] \rightarrow PT[\langle \text{choose} \ l_1 \ldots l_n \ | \ (\text{candidates} \ (\text{array}) \ l_s \ l_1 \ldots l_n) \rangle]\]

find methods
\[PT[\langle \text{candidates} \ l_r \ l_s \ l_1 \ldots l_n \rangle] \rightarrow PT[\langle \text{candidates} \ (\text{rank-next} \ l_r \ l_s \ | \ l_1 \ldots \ | \ (\text{linearise} \ l_n)) \ l_s \ l_1 \ldots \rangle]\]

trivial method search
\[PT[\langle \text{candidates} \ l_r \ l_s \rangle] \rightarrow PT[l_r]\]

choose
\[((\sigma \ b_1 \ldots \ (l_r \ S_1[\langle l_1 \ldots \rangle]) \ b_2 \ldots) \ T[\langle \text{choose} \ n_a \ l_r \rangle]) \rightarrow ((\sigma \ b_1 \ldots \ b_2 \ldots) \ T[\mathcal{B}[\langle n_a \ l_1 \ldots \rangle]])\]

Figure 6.14.: Dispatch reductions.

argument is implicitly specialised on \textbf{object}. Outer readers simply read the slot at position \(n_i\) from the given target, outer writers modify this same slot. The arguments of the \(\lambda\) form are accessed through regular message passing; these messages result in invocation of inner accessors. This design is aimed at simplifying the definition of reduction rules as much as possible.\(^{17}\)

Since outer accessors are plain methods, they are specialised on their context of definition. This means that the slot can be accessed only from such perspective (when the context is active). The model thus features context-specific slots.

Next to the plain slot accessors just described, the semantics features accessor methods that read and write delegation slots. The rules for delegation accessor methods are analogous to the ones for plain slot accessors and are available in the appendix (figure A.5 on page 200).

\subsection{6.4.8. Dispatch}

The multiple dispatch semantics is defined by the set of rules in figure 6.14. Message sending consists in looking up the selector among the given explicit arguments and the current activation which is prepended as implicit argument. The current activation takes part in method lookup as any other argument does. It thereby influences the choice of methods according to the state of the caller. This subjective dispatch semantics constitutes the foundation of dynamic behaviour adaptation to context.\(^{18}\) The method resulting from lookup is then invoked; the implicit argument is not passed to the \texttt{invoke} primitive in this rule, because the primitive will add it when reduced (recall figure 6.10).

\(^{17}\)Outer accessors can be defined so that they do not depend on inner accessors, but the rules become lengthier.

\(^{18}\)Recall that activations ultimately delegate to the current context.
Further decomposing the dispatch process, the method lookup rule states that looking up a method consists in finding all candidate methods and then choosing the most specific among the applicable ones. The candidates primitive receives as first argument an empty array where the results of the search will be stored. This result array is returned by candidates and therefore passed to the choose primitive once the search is over.

The candidates primitive can yield partially applicable methods that apply to only some of the arguments but not all of them. The choose primitive will filter partially applicable methods — those found in less than $|l_1 \cdots l_n|$ specialisers — and consider fully applicable methods only. To this end, the arity of the sought method $|l_1 \cdots l_n|$ is passed as first argument to choose.

The candidates primitive is defined recursively in rules find methods and trivial method search (the latter being the recursion base case). The last argument $l_n$ is ranked at each iteration. The ranking is performed by the rank-next primitive, which receives the object where the result of the ranking should be put, the selector $l_s$, the position of the argument being ranked (which is the amount of preceding arguments $|l_1 \cdots |$), and a linearisation of the argument — that is, a topological sorting of the object’s delegation graph. The linearise primitive encodes the C3* linearisation described informally in section 3.5.2 on page 46. A formal definition is provided in section 6.5.

Finally, the choose primitive is defined by means of metafunction $B$, which is a parameter of the model. $B$ yields the label of the most specific method found, or otherwise a special no-applicable-methods label to signal that no method is applicable. $B$ receives as arguments the sought method arity $n_a$ and the method ranking information $l_1 \cdots$ produced by rank-next. This ranking information consists of zero or more groups of labels $l_o l_p l_m$ telling the delegation distance $l_o$ and argument position $l_p$ for which method $l_m$ has been found.

Our definition of $B$ chooses the method with the best ranking by minimising the delegation distance for each argument position. It confers left-to-right priority to arguments (i.e. goes for the lowest $l_p$) to solve contention in the minimisation process. We have opted to omit the rather lengthy formal definition of $B$, which would not add much beyond the description just given. For further illustration, refer to section 3.4.3 on page 42.

**Method ranking**

Figure 6.15 shows the primitives for method ranking. The purpose of the rank-next primitive, defined recursively in rules rank next and rank end, is to iterate the rank primitive for each label in the linearisation $l_{a1} \cdots l_{an}$. This linearisation is that of object $l_{a1}$, because an object is always the first to appear in its own linearisation. The linearisation is stored in the slots of an object with label $l_a$ that has been created by linearise. The rank-next primitive will...
\[ \text{rank next} \]
\[
(B[\langle l_a, S_1[l_{a1} \cdots l_{an}\rangle]\mid T[(\text{rank-next} \ l_r, l_s, n_p, l_a)]) \rightarrow \\
(B[\langle l_a, S_1[l_{a1} \cdots\rangle]\mid T[(\text{rank-next} \ (\text{rank} \ l_r \ | \ l_{a1} \cdots | l_s, n_p, l_{an}) \ l_s, n_p, l_a)])
\]

\[ \text{rank end} \]
\[
((\sigma b_1 \cdots (l_a, S_1[()]) \ b_2 \cdots) \ T[(\text{rank-next} \ l_r, l_s, n_p, l_a)]) \rightarrow ((\sigma b_1 \cdots b_2 \cdots) \ T[l_r])
\]

\[ \text{rank} \]
\[
((\sigma b_1 \cdots (l_r, S[\langle l_1 \cdots\rangle]) \ l_a R[\langle\langle l_{s1} n_{p1} l_{m1} \cdots\rangle\rangle]) \ T[(\text{rank} \ l_r, n_o, l_s, n_p, l_a)]) \rightarrow \\
((\sigma b_1 \cdots (l_r, S[\mathcal{F}[\langle l_1 \cdots\rangle \ n_o, l_s, n_p, \langle\langle l_{s1} n_{p1} l_{m1} \cdots\rangle\rangle]) \ l_a R[\langle\langle l_{s1} n_{p1} l_{m1} \cdots\rangle\rangle]) \ T[l_r])
\]

Metafunctions:
\[
\mathcal{F}[\langle l_1 \cdots\rangle \ n_o, l_s, n_p, \langle\langle l_{s1} n_{p1} l_{m1} \cdots\rangle\rangle] = \\
\mathcal{F}[\langle l_1 \cdots\rangle \ n_o, l_s, n_p, \langle\langle l_{s2} n_{p2} l_{m2} \cdots\rangle\rangle]
\]

Figure 6.15.: Method ranking reductions.

take the last object \( l_{an} \) form the linearisation and rank it by invoking the \text{rank} primitive. This primitive receives the result object \( l_r \) where rank information is to be stored, the order \(|l_{a1} \cdots|\) of the object being ranked (i.e. the delegation distance from \( l_{a1} \)), the selector \( l_s \), argument position \( n_p \) and finally the object \( l_{an} \) being ranked. The result of \text{rank} is accumulated in \( l_r \).

The \text{rank} primitive is based on the \( \mathcal{F} \) metafunction for finding roles in object \( l_a \) matching the selector \( l_s \) at the argument position \( n_p \). \( \mathcal{F} \) is given the current ranking result \( l_1 \cdots \), the current order \( n_o \) of the ranking, the message selector \( l_s \), the argument position \( n_p \) and lastly, all the roles of object \( l_a \) being ranked. As shown in the bottom part of figure 6.15, metafunctions are defined by cases, which are considered in order. A case is applied only if the previous one did not match. In the first case of \( \mathcal{F} \), the function returns an augmented result set in which the group \( n_o, n_p, l_{m1} \) will be included if a role matching \( l_s \) and \( n_p \) is found in the first position of the roles being processed (this first role is then discarded). In the second case, if the first role does not match, it is discarded without accumulating a result. The last case in the definition of \( \mathcal{F} \) is the base case, in which no roles remain to be processed.

Defining \text{rank} purely in terms of reduction rules is possible, although the rules would be long and difficult to read. \( \mathcal{F} \) permits a shorter specification for \text{rank}, with the only drawback that it is large step: the function returns its result in one step, without any intermediate steps being performed by the abstract machine. Hence, the execution of \text{rank} never interleaves. A similar remark applies to the other metafunctions used in the semantics, namely \( \mathcal{B} \) and \( \mathcal{M} \). Metafunctions introduce “large steps” in the semantics, which are not bad \textit{per se} — they make the semantics more understandable — but as mentioned in the beginning of the chapter, small-step semantics can prove useful when the
study of thread interleavings becomes important.

This concludes the specification of the reduction system that encodes the semantics of AmOS, except for the \texttt{linearise} primitive, whose specification has been deferred to the following section.

\section{6.5. Delegation Graph Linearisation}

This section presents a formal definition of C3*, the linearisation at the heart of the \texttt{linearise} primitive used in figure 6.14 on page 129. We could specify C3* in terms of reduction rules and metafunctions,\footnote{We have such rules and metafunctions in our PLT Redex model.} as the rest of the semantics is specified, but the definitions are somewhat lengthy, making them more difficult to comprehend than the definition given next. Section 3.5.2 on page 49 discusses C3* informally.

C3* is a relaxed version of C3 \cite{8} that allows the linearisation of arbitrary delegation graphs. For graphs that C3 can linearise, C3* behaves exactly as C3 does. For graphs that are considered non-linearisable by C3, C3* uses local precedence as tiebreaker — that is, the linearisations of delegates that appear earlier in the delegation list of an object have precedence over the linearisations of later delegates.

\subsection{6.5.1. Notation and Terminology}

Even though we use fairly standard notation, we prefer to establish it explicitly:

- \( A^* = \cup_n A^n \) is the set of finite sequences of elements in \( A \).
- \( \mathcal{P}(A) = 2^A \) denotes the power set of \( A \).
- \( o \cdot a \) denotes the sequence formed by prepending object \( o \) to sequence \( a \).
- \( a \setminus S \) denotes the sequence obtained by removing from sequence \( a \) all elements found in set \( S \), without reordering of remaining elements.
- the pattern \((a_1, \ldots, a_n)\) denotes an arbitrary sequence \( \{a_i\}_{i=1\ldots n} \), including the empty sequence which is denoted by ( ).

Regarding terminology, we say that two strict order relations \( <_1 \) and \( <_2 \) are \textit{compatible} if \( a <_1 b \implies b \not<_2 a \) — that is, if there are no contradictions in the way they order elements.

\subsection{6.5.2. General definitions}

Let \( G(O, E) \) be a finite edge-labelled digraph, with vertices \( O \) (objects) and edges \( E \subseteq O \times O \) (delegations). The labelling function \( \rho : E \rightarrow \mathbb{N} \) assigns a numeric label to each delegation.
Given an object \( o \in O \), we define \( \text{d}_{\text{out}}(o) : O \rightarrow E^* \) as the sequence of outgoing delegations of object \( o \), \( \text{d}_{\text{out}}(o) = ((o, d_1), \ldots, (o, d_n)) \), where the order of the delegations \( e_i = (o, d_i) \) in the sequence is induced by the order of the edge labels \( \rho(e_i) \). We define the function \( \delta : O \rightarrow O^* \) as \( \delta(o) = (d_1, \ldots, d_n) \) —that is, \( \delta \) yields the sequence of direct delegates of object \( o \), with an order that is induced naturally by the order of the delegations in \( \text{d}_{\text{out}}(o) \). We now proceed to define \( C_3^* \) in terms of \( \delta \) and a few auxiliary functions.

### 6.5.3. Definition of \( C_3^* \)

#### Object linearisation

We define \( C_3^* : O \rightarrow O^* \) as \( C_3^*(o) = o \cdot \mathcal{L}_{\setminus}(\delta(o), \{o\}) \)

\( \mathcal{L}_{\setminus} \) linearises the delegates given as first argument, using a helper “memory” set of objects that have already been linearised, given as second argument.

#### Removal of linearised objects

We define \( \mathcal{L}_{\setminus} : O^* \times \mathcal{P}(O) \rightarrow O^* \) as \( \mathcal{L}_{\setminus}(a, S) = \mathcal{L}(a \setminus S, S) \)

\( \mathcal{L}_{\setminus} \) makes sure that none of the delegates to be linearised has been linearised already, by passing a stripped delegate sequence to \( \mathcal{L} \). This makes it possible to linearise cyclic graphs.

#### Recursive delegate linearisation

We define \( \mathcal{L} : O^* \times \mathcal{P}(O) \rightarrow O^* \) as

\[ \mathcal{L}((d_1, \ldots, d_n), S) = \mathcal{M}(l_1, \ldots, l_n, (d_1, \ldots, d_n)) \]

where \( l_i = d_i \cdot \mathcal{L}_{\setminus}(\delta(d_i), S \cup \{d_i\}) \)

Each object \( d_i \) is linearised recursively, and the resulting linearisations \( l_i \) are merged together. The merge function \( \mathcal{M} \) yields a sequence that contains all elements of the sequences given as arguments, and if possible, in a compatible order. The sequence of delegates itself is passed as last argument of \( \mathcal{M} \) so that the local precedence order is respected (again, if possible).

The recursive calls of \( \mathcal{L}_{\setminus} \) receive as second argument an augmented set of objects that have already been linearised. Since this set grows strictly with each application of \( \mathcal{L}_{\setminus} \), and the sequence passed as first argument to \( \mathcal{L} \) never contains elements from the set, it follows that eventually this first argument will become the empty sequence, ending the mutual recursion between \( \mathcal{L}_{\setminus} \) and \( \mathcal{L} \). Note that we recur on \( \mathcal{L} \) and \( \mathcal{L}_{\setminus} \) rather than on \( C_3^* \) directly because \( C_3^* \) does not carry the “memory” of already linearised objects as second argument.

#### Relaxed compatible merge

We define \( \mathcal{M} : (O^*)^* \rightarrow O^* \) as
Figure 6.16.: Sample delegation graphs.

\[ M() = ( ) \]
\[ M(l_1, \ldots, l_n) = M(l_1, \ldots, l_n) \]
\[ M(d_1 \cdot l_1, \ldots, d_k \cdot l_k, \ldots, d_n \cdot l_n) = d_k \cdot M( (d_1 \cdot l_1) \setminus \{d_k\}, \ldots, l_k, \ldots, (d_n \cdot l_n) \setminus \{d_k\}) \]
where \( k = \min_{i=1 \ldots n} (\forall 1 \leq j \leq n: d_i \notin l_j) \) if such \( k \) exists, or \( k = 1 \) otherwise.

The first case is the trivial case. The second case drops sequences that have been processed completely. The rationale behind the third case is as follows. If \( d_i \) belongs to the tail \( l_j \) of some sequence, then the head of that sequence \( d_j \) should appear in the final linearisation before \( d_i \) does: \( d_i \) cannot be chosen. If the graph is inconsistent, \( C_3^* \) relaxes this constraint and chooses \( k = 1 \) as tiebreaker, which gives priority to earlier delegates.

### 6.5.4. Examples

To give a better feeling of the previous formal definition and how \( C_3^* \) works, in this section we present three sample linearisations of the graphs shown in figure 6.16.\(^{21}\) The derivations in the first example contain more details to help understanding thoroughly the definitions, so that the omitted details in the other two examples can easily be reconstructed if needed.

#### Linearisation that is not monotonic in CLOS

For the first example, consider figure 6.16(a), which depicts a graph that is not linearised monotonically by the Common Lisp Object System (CLOS). The \( C_3^* \) linearisation of object 1 is monotonic, as shown by the following derivation. For brevity, we omit the applications of \( L \), and show only those of \( L \setminus \). The omitted applications do not contribute to this particular example.

\(^{21}\)These graphs are retaken from section 3.5.2, where \( C_3^* \) is discussed informally.
The derivation continues as follows:

\[ C_3^*(1) = 1 \cdot L\((2,4), \{1\}\) \]
\[ = 1 \cdot M(2 \cdot L\((3,6), \{1,2,4\}\), 4 \cdot L\((5), \{1,2,4\}\), (2,4)) \]
\[ = 1 \cdot M(2 \cdot M(3 \cdot L\((5), \{1,2,3,4,6\}\), 6 \cdot L\((7), \{1,2,3,4,6\}\), (3,6)), 4 \cdot M(5 \cdot L\((7), \{1,2,4,5\}\), (5)), (2,4)) \]
\[ = 1 \cdot M(2 \cdot M(3 \cdot M(5 \cdot L\((7), \{1,2,3,4,5,6\}\), (5)), 6 \cdot M(7 \cdot L\((7), \{1,2,3,4,5,6,7\}\), (7)), (3,6)), 4 \cdot M(5 \cdot M(7 \cdot (7), (7)), (2,4)) \]
\[ = 1 \cdot M(2 \cdot M(3 \cdot M(5 \cdot M(7 \cdot (7), (7))), 6 \cdot (7), (3,6)), 4 \cdot M(5 \cdot (7)), (2,4)) \]
\[ = 1 \cdot M(2 \cdot M(3 \cdot M(5 \cdot (7), (6,7), (3,6)), 4 \cdot (5,7), (2,4)) \]
\[ = 1 \cdot M(2 \cdot M((3,5,7), (6,7), (3,6)), (4,5,7), (2,4)). \]

For \( M((3,5,7), (6,7), (3,6)) \) we have

\[ M((3,5,7), (6,7), (3,6)) = 3 \cdot M((5,7), (6,7), (6)) \]
\[ = 3 \cdot 5 \cdot M((7), (6,7), (6)) \]
\[ = 3 \cdot 5 \cdot 6 \cdot M((7), (7)) \]
\[ = 3 \cdot 5 \cdot 6 \cdot 7 \cdot M((7), (7)) \]
\[ = (3,5,6,7). \]

The derivation continues as follows:

\[ = 1 \cdot M(2 \cdot (3,5,6,7), (4,5,7), (2,4)) \]
\[ = 1 \cdot M((2,3,5,6,7), (4,5,7), (2,4)) \]
\[ = 1 \cdot 2 \cdot M((3,5,6,7), (4,5,7), (4)) \]
\[ = 1 \cdot 2 \cdot 3 \cdot 4 \cdot M((5,6,7), (5,7), (1)) \]
\[ = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot M((6,7), (7)) \]
\[ = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot M((7), (7)) \]
\[ = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot M((7), (7)) \]
\[ = (1,2,3,4,5,6,7). \]
Linearisation that cannot preserve monotonicity

Consider figure 6.16(b), illustrating a graph that cannot be linearised by C3, because it is impossible to preserve monotonicity. The C3* linearisation is given by the following derivation. As before, we omit the applications of $\mathcal{L}$ for brevity.

$$C3^*(1) = 1 \cdot \mathcal{L}((2, 3), \{1\})$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{L}((4, 5), \{1, 2, 3\}), 3 \cdot \mathcal{L}((5, 4), \{1, 2, 3\}), (2, 3))$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(4 \cdot \mathcal{L}((6), \{1, 2, 3, 4, 5\}), 5 \cdot \mathcal{L}((6), \{1, 2, 3, 4, 5\}), (4, 5)), 3 \cdot \mathcal{M}(5 \cdot \mathcal{L}((6), \{1, 2, 3, 4, 5\}), 4 \cdot \mathcal{L}((6), \{1, 2, 3, 4, 5\}), (5, 4)), (2, 3))$$

omitting the straightforward applications of $\mathcal{M}$:
$$= 1 \cdot \mathcal{M}(2 \cdot (4, 5, 6), 3 \cdot (5, 4, 6), (2, 3))$$
$$= 1 \cdot \mathcal{M}((2, 4, 5, 6), (3, 5, 4, 6), (2, 3))$$

after two straightforward applications of $\mathcal{M}$:
$$= 1 \cdot 2 \cdot 3 \cdot \mathcal{M}((4, 5, 6), (5, 4, 6))$$

tiebreaker case (favouring earlier delegates):
$$= 1 \cdot 2 \cdot 3 \cdot 4 \cdot \mathcal{M}((5, 6), (5, 6))$$
$$= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot ()$$
$$= (1, 2, 3, 4, 5, 6)$$

Linearisation of a cyclic graph

Consider the cyclic graph depicted in figure 6.16(c). It is impossible to compatibly sort it according to any topological ordering. The C3* linearisation is well-defined for cyclic graphs, although the result cannot possibly be compatible with the graph. The following derivation shows the result of linearising object 1. Unlike the previous examples, in this derivation we show the relevant applications of $\mathcal{L}$.

$$C3^*(1) = 1 \cdot \mathcal{L}((2), \{1\})$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{L}((3), \{1, 2\}), (2))$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(3 \cdot \mathcal{L}((4), \{1, 2, 3\}), (3)), (2))$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(3 \cdot \mathcal{M}(4 \cdot \mathcal{L}((2), \{1, 2, 3, 4\}), (4)), (3)), (2))$$

linearised object removal:
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(3 \cdot \mathcal{M}(4 \cdot \mathcal{L}((2) \setminus \{1, 2, 3, 4\}, \{1, 2, 3, 4\}), (3)), (2)), (2))$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(3 \cdot \mathcal{M}(4 \cdot \mathcal{L}(() \setminus \{1, 2, 3, 4\}, \{1, 2, 3, 4\}), (4)), (3)), (2))$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(3 \cdot \mathcal{M}(4 \cdot \mathcal{M}(), (4)), (3)), (2))$$
$$= 1 \cdot \mathcal{M}(2 \cdot \mathcal{M}(3 \cdot (4), (3)), (2))$$
$$= 1 \cdot \mathcal{M}(2 \cdot (3, 4), (2))$$
$$= 1 \cdot (2, 3, 4)$$
$$= (1, 2, 3, 4)$$
6.6. Relationship to the Implementation

We conclude the chapter with a few notes on the relationship between the formal semantics defined in this chapter and the informal description of AmOS given in chapter 5, which corresponds to the implementation of the model.

**Closures** In the semantics, closures are activations, whereas in the implementation closures and their activations are separate objects (see section 5.2.2 on page 92). The implementation goes even further by separating the arguments from the activation, as shown in figure 5.3 on page 92. Thus, what is normally one object in the semantics, becomes three different objects in the implementation. By having separate objects we improve performance, and we allow first-class access to (e.g.) the prototypical arguments of closures — which are also the specialisers of methods. The Meta Object Protocol (MOP) of AmOS thus becomes finer grained.

AmOS preserves the semantics by treating the separate activation and arguments as part objects of the closure [11]. This means that two different closures will never share the same activation or arguments objects, and if they do (for optimisation reasons), they will perform only read accesses to those shared objects. There is no risk that one closure will override information belonging to the activation of some other (cloned) closure.

**Delegations** In the semantics, delegation slots are an integral part of the object, and are distinguished from plain slots. In the implementation, delegations are kept as plain slots of a separate object, and there needs not be a distinction between plain and delegation slots (see section 5.2.1 on page 91). As in the case of closures, the semantics is preserved by treating the separate delegation object as a part object. This is achieved by implementing a copy-on-write policy for delegation objects, described in section 5.2.1. Hence, there is no risk that two objects will overwrite each other’s delegations.

Modelling delegations as plain slots stored in a separate object would spare the semantic definition of rules for addition, removal and accessor creation for delegations. However, the rules dealing with method lookup need to access object delegations, and would therefore become more complex. The economy in amount of rules would not compensate the loss of clarity. Besides, the way delegations are modelled in the semantics makes the structure of objects apparent, as can be observed in figure 6.1.

**Roles** Like delegations, roles are integral parts of objects in the semantics, whereas they are separate (first-class) objects in the implementation. In the semantics, the structure of objects is more explicit, partly for documentation

---

22 This is related to the traditional distinction between composition and aggregation in object-oriented programming: part objects are considered integral part of an object (e.g. a university and its departments), whereas other objects are just acquaintances (e.g. a university and its professors). If an object is cloned, its parts are (conceptually) cloned as well. If an object is disappears, its parts disappear as well.

23 Shown in figures A.3, A.4 and A.5 on page 199.
purposes, partly for simplicity of reduction rules. In the implementation, the reification of roles is in keeping with the open implementation policy of AmOS, in which all basic model entities are available through the MOP. Regarding consistency, the semantics is preserved in the implementation by handling role objects as part objects with a copy-on-write policy.

**Accessors** Whereas new reader and writer methods are created for every invocation of `reader-method` and `writer-method` in the semantics, the implementation memoises accessor methods so that at most one accessor method exists in the whole system for each slot index. The same applies to other accessor kinds (e.g. outer accessors, delegation accessors) This memoisation greatly reduces memory load and saves processing time. Some other optimisations are possible. For instance, outer accessors do not work by invoking inner accessors as is the case in the semantics. This would be prohibitively slow.

**Lookup arguments** Lookup arguments, described in section 5.2.5 on page 99, are not defined by the semantics. As a consequence, accessor methods in the semantics read from and write to their actual arguments, instead of their lookup arguments. As shown in figure 6.12, they use the invoking activation, which is the accessor’s actual argument at position 0, instead of the lookup argument at the same position.

Without lookup arguments, the semantics cannot support argument accesses across nested closures:

\[
\lambda (x) \\
\quad (\lambda (y) \\
\quad \quad (+ (send x) (send y)))
\]

In this example, `(send x)` will result in invocation of an accessor method that will read from the sender’s activation, which happens to be the inner closure. Hence, the message incorrectly yields `y`’s value instead of `x`’s.

To define lookup arguments, we could extend objects with one more part besides roles, delegations, slots and body. This part would contain the lookup arguments of methods, and would be empty for objects other than methods. Lookup arguments would be filled in as roles are found during the method lookup process.\(^{24}\) In our current semantic definition, roles are examined thanks to metafunction \(\mathcal{F}\) (see figure 6.15 on page 131). Given that the store cannot be altered by means of metafunctions, \(\mathcal{F}\) cannot do the job: it cannot write the lookup arguments of methods in the store. We would need to redefine \(\mathcal{F}\) as a primitive `find-roles` operation and specify it by means of reduction rules.

\(^{24}\) This is precisely why they are called “lookup” arguments.
7 Assessment of Expressiveness of AmOS

This chapter shows the expressiveness experiments we have carried out in the Ambient Object System (AmOS) to validate the thesis statement given in section 1.3. We have chosen AmOS rather than Ambience to perform these experiments because AmOS contains the core underlying computation model behind Ambience, and it is this core computation model we want to validate. In its current status, Ambience is little more than a syntactic layer on top of AmOS.\footnote{The choice is driven also by pragmatical matters: by using Lisp we have better IDE support (code navigation, syntax highlighting, etc.) than in Ambience.}

The validation has been performed by implementing a number of scenarios, each highlighting particular features or advantages of our approach. We start by explaining our validation approach in section 7.1. Section 7.2 then presents the different case studies. Each case study includes a discussion of how it validates the points we make.

7.1. Validation Approach

Even though assessing the expressiveness of a programming language is not trivial \cite{49}, we have tried to take as systematic an approach as possible in the validation of our computation model. In this section we present the points we wish to validate, and the approach to carrying out such validation.

7.1.1. Points to Validate (the What and the Why)

We need to validate the thesis put forward in section 1.3. We started our work from the premise that true ambient applications require dynamic behaviour adaptation to context. Motivated by that premise, we made our \textit{general} thesis statement:

\begin{quote}
Dynamically adaptable context-aware applications can be written elegantly\footnote{Read: concisely, legibly, with simplified control flow and with little or no tangled code \cite{71}.} thanks to specific linguistic support to deal with behavioural context dependencies.
\end{quote}

AmOS—a simple yet powerful computation model based on classless objects and multimethods—readily provides such linguistic support, exhibiting the following properties. These properties are the \textit{specific} thesis statements that together imply the general thesis statement:
V1 Dynamic adaptations: application behaviour can effectively be adapted to different contexts at run time, as context changes are detected.

V2 Clean application logic: base application logic can be written in a straightforward manner, orthogonally to extrinsic context adaptation concerns.

V3 Simplified application logic: the expression of base application logic can be simplified thanks to exploitation of context-oriented programming for intrinsic application contexts or states.

V4 Non-intrusive adaptations: adaptation code can be incorporated in a non-intrusive fashion (i.e. without modifying base application code).

V5 Straightforward architectures: software architectures are not biased towards context adaptation. A good indicator of “straightforwardness” is reusability: whether base code is tied to one specific use, or it could be part of other applications in the same domain, but with different context adaptation needs.

Our model has one more property that derives from item V4:

V6 Dynamic addition of software features: software features can be modularised cleanly as bundles of extensions of known application objects. Once installed, features can activated and deactivated dynamically.

Item V4 is about basic behavioural object extension — about introducing new methods non-intrusively. Item V6 is about the modularisation of these extensions such that they can be installed, activated, deactivated and generally treated as one whole unit.

7.1.2. Validation Setup (the How)

Our main validation points V1–V6 are about the ability to express context-oriented applications elegantly. Therefore, we have chosen as main validation method the implementation of scenarios that allow us to assess the expressiveness of our model. Table 7.1 shows the validation points addressed by each scenario. By implementing different kinds of scenarios we reduce the likelihood that our general conclusions are biased by the particular intricacies of any one of them.

Limitations

There are a number of limitations in the validation we have carried out. Firstly, because of a lack of resources, we could not carry out comparative experiments in which we would implement the same scenarios in other programming languages. By having two parallel implementations, we could have made a subjective judgement of expressiveness in each language. A second alternative would be to make a quantitative comparison by means of metrics — by counting lines
Table 7.1.: The rows correspond to the criteria we want to validate, and the columns to the different scenarios we implemented to this end.

<table>
<thead>
<tr>
<th>Video codec</th>
<th>Phone extensions</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>V2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>V3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>V4</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>V5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>V6</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The case studies presented in this section are aimed at validating the points V1–V6 presented in section 7.1.1 on page 139. The case studies share a common structure:

- a scenario describing a typical use of context-aware software,
- a description of expected software functionality,
• a description of the implementation exhibiting the required functionality, and
• a discussion on how the case study validates the criteria shown in table 7.1.

7.2.1. Adaptive Video Player

In this case study we assess on-the-fly behaviour adaptation to changes in the operating environment of applications. The case study is motivated by the following scenario.

The ACME company is developing a new portable video player. Their player has, among others, embedded support for MPEG-1 videos. As a hardware constructor, ACME does not develop MPEG-1 decoders. Rather, the decoder is outsourced to 3ivx, a company specialised in video codec technology for embedded systems. As experts in this domain, 3ivx’s engineers have created an adaptive decoder. The decoder switches to a degraded mode when the battery charge is low; this mode requires less CPU cycles and therefore saves battery power, at the cost of some loss of quality. The decoder switches back to normal operation mode once the battery has been recharged.

By implementing this scenario, we illustrate both dynamic adaptation to context and interoperation of services developed independently by different parties (ACME for video playback and 3ivx for video decoding).

Expected Functionality

Before explaining the underlying implementation, we show the expected observable behaviour. Videos are played by invoking a standard `play` method, which receives a video object and the player on which to play it:

```
( play giovanni-sollima-mpeg-1-video host-acme-video-player )
```

As a result, the given video will be played on the current host device, producing the output shown in snippet 7.1. The stream has 10 video frames in total. For the sake of demonstrating the codec’s dynamic adaptation to remaining battery power, we simulate the gradual loss of battery charge by decrementing the charge by 10% for each displayed video frame. Initially, the battery charge is set to 100%. When the battery reaches a threshold of 30% remaining charge, 3ivx’s decoder switches to degraded mode. Hence, after 7 displayed frames, the quality of service is lowered, as can be seen on line 8. From that point on, the decoder uses a lighter weight algorithm, albeit delivering less quality.

The battery is recharged as follows:

---

3MPEG-1 is a video and audio compression standard from the Moving Picture Experts Group, a working group of ISO/IEC charged with the development of video and audio encoding standards.
7.2. Case Studies

1. Opening MPEG-1 stream (10 frames).
2. Playing MPEG-1 stream on ACME (tm) video player
3. High-quality frame decoding from MPEG-1 stream
4. Showing video frame on video display
5. High-quality frame decoding from MPEG-1 stream
6. Showing video frame on video display
7. [5 more times]
8. 3ivx: degrading service
9. Quick and dirty frame decoding from MPEG-1 stream
10. Showing video frame on video display
11. [2 more times]

Snippet 7.1: Video playback using ACME’s portable video player and 3IVX’s decoder.

\[
\text{(setq (charge (battery host-acme-video-player))) 100)}
\]

The battery getter method is used to grab the battery object from the player. The (setq charge) setter method\(^4\) is used to set the charge back to 100%. This update of the charge level is captured by 3IVX’s codec, triggering the following message:

3ivx: upgrading service

After recharging, video frame decoding is set back to high quality.

**Implementation**

The adaptive functionality just explained can be expressed in existing languages. What makes the case study special, as shown next, is that the implementation in AmOS has no conditional statements, and the application’s logic is not tangled with context-adaptation concerns. Further, the code is cleanly modularised such that the code from ACME and 3IVX could be replaced by code from any other provider.

Figure 7.1 shows the resulting context graph structure of the implemented solution. For clarity, the diagram is divided in three parts: the system contexts containing all basic functionality of AmOS, the framework contexts where standard prototypes and behaviour are defined for each specific domain, and the provider contexts which represent the availability of the particular services furnished by the ACME and 3IVX companies. The arrows depict delegations. These delegations decode naturally the dependencies one would expect: mobile devices are devices that require autonomous power sources (i.e. batteries); video players are devices concerned with video; ACME’s portable video player

---

\(^4\)AmOS uses the **setq** generalised assignment operator from Common Lisp. In Common Lisp, the list (setq charge) composed of the two symbols **setq** and **charge** is a valid function name; **setq** functions are the only ones that can have non-atomic names. Consequently, in AmOS (setq charge) is a valid method selector. AmOS even goes beyond, by allowing any object to be the selector of a method.
is a mobile device for playing videos; finally, 3IVX’s codec is about decoding MPEG-1 video, but it has a link to the power source context because the codec’s adaptation policy takes power consumption into account. Note that video players are not coupled to mobile devices (e.g. standard video players have no batteries), it is only in ACME’s portable video player context that the video player and mobile device contexts are put together.

Next we proceed to explain the prototypes and methods that are specific to each context shown in figure 7.1. Some of these contexts are reused in some of our other case studies. We omit the explanation of the system context which contains core AmOS functionality already explained in section 5.3 on page 107, and the standard context, which contains platform (non-core) functionality for e.g. dynamically loading modules —code units containing context, prototype and method definitions. For brevity, we also omit the context definitions themselves: such information is already depicted in figure 7.1. Here are two examples of the omitted context definitions:

(defcontext @mobile-device-context
  :uses (@device-context @power-source-context))

(defcontext @acme-video-player-context
  :uses (@video-player-context @mobile-device-context))

The :uses keyword is a shortcut to declare delegations.⁵ All definitions avail-

---

⁵The order of delegations shown in figure 7.1 has been changed for visual clarity, but the order is immaterial to this case study.
7.2. Case Studies

The power source context is shown in snippet 7.2. On line 1 the current context is switched to `@power-source-context` so that all subsequent definitions are specific to that context. The `@battery` prototype is defined on line 2, with an internal charge measure. The `charge` reader method defined on line 6 reads the current remaining charge, and the `(setf charge)` writer method defined on lines 9–17 changes the remaining charge, activating or deactivating the `@low-battery-context` according to a 30% threshold level.\(^6\) The `@low-battery-context` context can be used to specialise behaviour for situations where the current battery charge is low, as is shown later on. This prototypical behaviour for changing the charge of a battery can be overridden by battery constructors if needed —if a constructor sells a battery with different criteria for deciding on low charge conditions. Finally, the `print-object` method on line 19 is analogous to the `toString` method of Java: it writes the given object to an output stream in a human-readable way (in this case, it is

---

\(^6\)The activation of an already active context, and the deactivation of an already inactive context, are harmless ineffective operations.
Device and Mobile Device Contexts

We use a minimal model of ambient computing devices, shown in snippet 7.3. The generic \texttt{@device} prototype is created on line 2. The \texttt{@mobile-device} prototype is created on line 5, by extending \texttt{@device}. In our minimalistic model, mobility simply implies that the device has autonomous power. Hence, a \texttt{battery} slot is added on line 6, with the \texttt{@battery} prototype shown previously as value.

Video Context

ACME uses the prototypical model of the video context provided by AmOS. In video context, there are videos containing a sequence of still images (normally with accompanying synchronised sound, and information such as the title and authors, but we omit those for simplicity). A video stream can be obtained by opening a video (e.g. from a file or from a network connection). The stream is decoded by a video codec, producing a series of video frames that can be shown on a computer display.

The video context just described is modelled as shown in snippet 7.4. Prototypes of a video, a video codec, a video stream and a video frame are created on lines 2, 3, 4 and 6. The number of remaining frames in the stream is added with an \texttt{add-slot} call. Finally, methods \texttt{open} and \texttt{exhausted} define prototypical behaviour of a video stream; method \texttt{next-frame} defines a prototypical interaction between a video codec and a video stream to obtain a video frame.

Note that the \texttt{@video-context} is oblivious to portable video players (or any other kind of device). It could be reused “as is” in any situation that involves manipulating videos. To be more fine-grained, we could have distinguished between two more specific contexts, video playback and video recording, both
of them delegating to a more generic video context where only common functionality would be defined. Although generally a good idea, having such level of refinement is not necessary for making our point in this case study (we use only video playback functionality).

Video Player Context The video player context, shown in snippet 7.5, is about devices which are able to display videos. The definition of a minimal video player is shown in lines 2–3: a video player has at least a display on which to show videos. The stream and codec invocations on lines 6 and 7 are framework methods, specialised by concrete providers as explained next.

ACME Video Context A particular company can model its own business domain. In our current example the ACME company is a constructor of mobile video players. The company thus produces the ACME video context shown in snippet 7.6, in which it defines the necessary software that drives the video players. Line 3 with the extend-many call is worth noticing: it shows how a prototype can be the extension of more than one existing prototype (in this case, it is two of them). The extend-many operation is explained in section 5.3.1 on page 107. This way, the @acme-video-player prototype naturally models a mobile video player. On line 5 the print-object is overridden so that a user-friendly representation is displayed on the console.
Assessment of Expressiveness of AmOS

(in-context @video-player-context)
(defproto @video-player (extend @device))
(add-slot @video-player 'display @display)
(defmethod play ((video @video) (player @video-player))
  (let ((stream (stream video)))
    (codec (codec video)))
  (format t "Playing ~a on ~a-%" stream player)
  (loop
    until (exhausted stream)
    do (show (next-frame codec stream)
      (display player))))
(defmethod show ((frame @video-frame) (display @display))
  (format t "Showing ~a on ~a-%" frame display))

Snippet 7.5: Video player context.

(in-context @acme-video-player-context)
(defproto @acme-video-player (extend-many (list @video-player @mobile-device)))
(defmethod print-object (stream (object @acme-video-player))
  (cl:format stream "ACME(tm) video player"))

Snippet 7.6: ACME’s video context.

3ivx Video Context 3ivx, a provider of advanced video codecs, defines the
two codecs shown in snippet 7.7. The defslot calls in lines 2–3 add two slots
named 3ivx-quality-mpeg-1-codec and 3ivx-lousy-mpeg-1-codec to the current
context. Since these are not prototypes (i.e. they are not meant to
be cloned), their names are not prefixed with @. Two next-frame methods
are specialised on the high-quality codec and the low-quality one. The resend
call in both cases will invoke the generic next-frame method which decrements
the remaining frame count. Note that the code in snippet 7.7 is not coupled to
power management in any way. The two codecs can be used for many purposes,
for instance the low-quality codec could be used in mobile phones with small
screens, since those devices would not be able to take advantage of the high-
quality codec anyway. The code in snippet 7.7 is thus part of 3ivx’s general
library and can be reused in many of their products.

For devices that can take advantage of high-quality decoding, but wish to
adapt to low-power situations, 3ivx provides the adaptive codec shown in snip-
ippet 7.8. An adaptive codec is defined in line 2, with a parent delegation link
that initially points to the high-quality codec. Hence, the adaptive codec is
high-quality by default. On line 6 the codec framework method is implemen-

7The defslot call is identical to defproto, but having a separate defslot method makes
the intention clear in the code.
7.2. Case Studies

Snippet 7.7: 3IVX’s video context.

1 (in-context @3ivx-video-context)
2 (defslot 3ivx-quality-mpeg-1-codec (refine @mpeg-1-codec))
3 (defslot 3ivx-lousy-mpeg-1-codec (refine @mpeg-1-codec))
4
5 (defmethod next-frame ((codec 3ivx-quality-mpeg-1-codec)
6 (stream @mpeg-1-stream))
7 (format t "High-quality frame decoding from ~a~%" stream)
8 (resend))
9
10 (defmethod next-frame ((codec 3ivx-lousy-mpeg-1-codec)
11 (stream @mpeg-1-stream))
12 (format t "Quick and dirty frame decoding from ~a~%"
13 stream)
14 (resend))

ted by 3IVX. This method is invoked by framework clients to obtain a codec with which to play a given video (see line 7 of snippet 7.5 on the preceding page). Since the device will run constantly in 3IVX’s context, this version of the codec method will be invoked for MPEG-1 videos, therefore returning the adaptive codec. Finally, snippet 7.8 shows two contexts for normal and degraded operation mode, and two versions of next-frame specialised on those contexts. These methods switch the delegation link of the codec dynamically so that high-quality decoding is used in normal operation mode, and low-quality decoding is used in degraded operation mode. It is a well-known property of prototypes that behaviour can be changed dynamically when some event occurs thanks to dynamic inheritance [85, 93], without recurring to hard-coded if statements inside method bodies.

Finally, we show in snippet 7.9 the simple context meta-programming needed to capture context changes related to battery charge and (de)activate the normal and degraded operation modes accordingly. Suitable overridden versions of the switch-on and switch-off methods\(^8\) capture the activation of @low-battery-context in 3IVX’s context. If 3IVX’s video context would be deactivated for some reason (for instance if their codecs are replaced by those of some other provider), the traps shown in snippet 7.9 would be disabled automatically, since the methods defined therein would not be applicable.

Discussion

The validation points are addressed as follows:

**Dynamic adaptation (V1)**

The case study is about run-time quality of service adaptation of a video

---

\(^8\)These methods are part of the standard context management framework provided by AMOS; they are called by the system during context activation and deactivation as explained in section 5.3.2.
(in-context @3ivx-video-context)

(defslot 3ivx-adaptive-mpeg-1-codec (extend @object))

(add-delegation 3ivx-adaptive-mpeg-1-codec 'parent 3ivx-quality-mpeg-1-codec)

(defmethod codec ((video @mpeg-1-video))
  3ivx-adaptive-mpeg-1-codec)

(defcontext @3ivx-normal-operation-context)
(defcontext @3ivx-degraded-operation-context)
(with-context @3ivx-normal-operation-context
  (defmethod next-frame ((codec 3ivx-lousy-mpeg-1-codec)
    (stream @mpeg-1-stream))
    (format t "Switching to high-quality mode ~%")
    (setf (parent codec) 3ivx-quality-mpeg-1-codec)
    (resend)))

(with-context @3ivx-degraded-operation-context
  (defmethod next-frame ((codec 3ivx-quality-mpeg-1-codec)
    (stream @mpeg-1-stream))
    (format t "Switching to low-battery mode ~%")
    (setf (parent codec) 3ivx-lousy-mpeg-1-codec)
    (resend)))

Snippet 7.8: 3ivx’s adaptation code.

decoder, according to an internal, physical property of the device, namely remaining battery power. The adaptation is of a discrete nature: the decoder works either in high-quality or in low-quality mode. Having gradual levels of quality, or continuous degradation strategies9 might be desirable, specially in the domain of video decoding, and even more so taking into account that battery depletion is a gradual process rather than a discrete one. Albeit an interesting point for discussion and research, in this dissertation we are not concerned with the distinction between discrete and continuous adaptations to context. Continuous adaptations must always be implemented via one parametrised algorithm,10 whereas the point we want to make is about adapting behaviour by switching different algorithms.11 A mixture of discrete and continuous adaptations is of course possible. In the validation case, if the decoder supported gradual quality levels instead of two quality modes, our approach could still prove useful for example to switch

9 As opposed to discrete degradation shown by the case study.

10 If there are various algorithms that must be selected according to some criteria, the adaptation is no longer purely continuous: it is a mixture of continuous and discrete. Discrete behaviour adaptation, precisely, is about supporting more than one possible algorithm.

11 Discrete adaptations can also be achieved via parametrisation (i.e. having algorithms depends on discrete values passed as parameters), but our approach seeks to avoid this solution in the general case, for the reasons mentioned in section 5.2.4 on page 96.
(in-context @3ivx-video-context)

(defmethod switch-on ((context @low-battery-context))
  (format t "3ivx: degrading service-%")
  (deactivate-context @3ivx-normal-operation-context)
  (activate-context @3ivx-degraded-operation-context)
  (resend))

(defmethod switch-off ((context @low-battery-context))
  (format t "3ivx: upgrading service-%")
  (deactivate-context @3ivx-degraded-operation-context)
  (activate-context @3ivx-normal-operation-context)
  (resend))

Snippet 7.9: 3ivx’s context meta-programming.

from “default” non-adaptive operation to gradual “power-aware” operation, if the video player passes from being powered by a DC adaptor to running on batteries.

To conclude, the case study shows that it is easy to express dynamic stepwise adaptations to context, by changing the internal composition of the application (e.g. by switching parent delegation link of the decoder dynamically).

Clean application logic (V2)

None of the snippets shown in the case study contains tangled code. For example, 3ivx’s software, even if dealing with two different concerns —video decoding and power consumption — is modularised so that frame decoding (snippet 7.7) is independent of the power-aware part of the decoder (snippets 7.8 and 7.9).

Non-intrusive adaptations (V4)

The framework code belonging to power management (snippet 7.2) and to video manipulation (snippets 7.4 and 7.5) needs no intervention to accommodate 3ivx’s adaptive video codec. This is mainly due to the use of multimethods and open objects: in our approach, methods can be added and specialised without intervening existing code. In particular, and adaptive version of the next-frame method is introduced by 3ivx non-intrusively in snippet 7.8.

Straightforward architectures (V5)

The case study shows the way contexts can be put in direct correspondence with an application’s domain. There are separate contexts for video players, for power management and for video decoders. As illustrated in figure 7.1 on page 144, delegation relationships among contexts naturally encode dependencies of specific contexts on more general ones.

As illustrated in figure 7.1, framework contexts can be decoupled from provider-specific contexts (ACME’s and 3ivx’s) which implement the frame-
work. This allows independently developed modules to interoperate and be part of a same adaptable application.

Regarding architecture on the framework level, the frameworks related to video and video playing in snippets 7.4 and 7.5 are independent of (e.g.) power consumption and the particular video format. The same code could be reused with a decoder that is not aware of power consumption, and with video streams in formats other than MPEG-1.

Regarding architecture on the provider level, we note that code from 3ivx is modularised so that video decoding (snippet 7.7) is reusable independently of power adaptation concerns.

Other than the validation points, the case study shows an example of context meta-programming (snippet 7.9), in which context switches related to power management are intercepted and put in relation to contexts dealing with video decoding quality. This coupling of contexts takes place only when 3ivx’s adaptive decoder is active (i.e. when context $\texttt{3ivx-video-context}$ is active). A last highlight is the use of dynamic inheritance in snippet 7.8 to change object behaviour (in this case, decoding quality).

### 7.2.2. Software Feature Interaction in a Mobile Phone

This second case study is motivated by the following scenario.

The ACME company has been developing throughout the years a standard line of consumer mobile phones. By default, the phones do not exhibit any kind of dynamic behaviour adaptability. Like most mobile phones, their main functionality consists in receiving possibly simultaneous phone calls. At most one call can be on-hook (i.e. active), and the rest of the calls are kept on hold.

Observing that the default functionality is too basic to meet current user expectations, ACME has developed new software extensions that render the mobile phones smarter with respect to the context. The Discretion extension detects situations in which the phone should be silent (e.g. in libraries and hospitals). The Call Forwarding extension handles calls received at inconvenient times by directing them to another number (e.g. forwarding them to a secretary if the user is taking part in a meeting).

Besides dynamic adaptation to context, with this scenario we aim at illustrating dynamic addition of software features to an existing application.

#### Expected Functionality

Before showing the actual code, this part explains the concrete software functionality expected from the scenario, and it provides a discussion of the semantics related to context activation and deactivation in AmOS whose importance becomes particularly apparent in this scenario.
The mobile phone developed by ACME exhibits the following basic call handling functions:

- **receive** signals a new incoming call,
- **answer** connects an incoming call,
- **suspend** puts the active call on hold,
- **resume** retrieves a suspended call, and
- **hang-up** disconnects the currently active call.

For the sake of this case study, we concentrate on call reception behaviour of the phone. The call-handling rules for call reception are as follows:

- By default, new incoming calls are signaled by playing a predefined ringtone.
- If the phone is off-hook (in use), a call waiting signal is played instead.
- Calls can be suspended and resumed arbitrarily.

### Basic functionality

When a call is received, the default behaviour is simple:

\[
\text{receive (make-call 'alice) *bobs-phone*) } \rightarrow \\
\text{Playing ringtone through phone speaker}
\]

The state of the phone can be consulted by means of a **describe** operation:

\[
\text{(describe *bobs-phone*) } \rightarrow \\
\text{1 incoming calls}
\text{0 ongoing calls}
\text{0 missed calls}
\text{0 terminated calls}
\]

The phone has four call queues. As shown in the output, the incoming call from Alice has been enqueued. The **answer** method picks up the first call from the **incoming** queue (thus the incoming call that has waited the most):

\[
\text{(answer *bobs-phone*) } \rightarrow \\
\text{Answering phone call from Alice on +32 10 74 19 80}
\text{(describe *bobs-phone*) } \rightarrow \\
\text{0 incoming calls}
\text{1 ongoing calls}
\text{0 missed calls}
\text{0 terminated calls}
\]

When the call is answered, it is put in the **ongoing** call queue.

If a second call is received at this point, the speaker cannot play a ringtone. The phone rather plays a soft short tone to let the user know that another call is coming in:
Assessment of Expressiveness of AmOS

(receive (make-call 'carol) *bobs-phone*) → Playing call waiting signal through phone speaker

(describe *bobs-phone*) → Current: phone call from Alice
1 incoming calls
1 ongoing calls
0 missed calls
0 terminated calls

The user can leave the current call on hold and pick up the new one:

(suspend *bobs-phone*) → Putting phone call from Alice on hold

(answer *bobs-phone*) → Answering phone call from Carol on +32 10 74 19 80

(describe *bobs-phone*) → Current: phone call from Carol
0 incoming calls
2 ongoing calls
0 missed calls
0 terminated calls

At this point two calls are connected simultaneously, but only one call is active. The user can finish the call with Carol and resume the call with Alice:

(hang-up *bobs-phone*) → Terminating phone call from Carol

(resume (first (ongoing (calls *bobs-phone*)))) *bobs-phone*) Resuming phone call from Alice

To finish the interaction:

(hang-up *bobs-phone*) → Terminating phone call from Alice

(describe *bobs-phone*) → 0 incoming calls
0 ongoing calls
0 missed calls
2 terminated calls

Discretion extension The Discretion extension can be installed as follows:

(require-module 'acme/telephony/discretion)

The require-module call will load the designated file. The code within that file can extend the state and behaviour of existing objects so that the software
This new software feature makes the phone aware of silent environments:

\[(\text{activate-context @silent-context}) \rightarrow \text{Switching @silent-context on}\]

\[(\text{receive (make-call 'alice) *bobs-phone*)} \rightarrow \text{Activating phone vibrator}\]

Now incoming calls activate the phone vibrator instead of playing the ringtone when the phone is in a silent context. If the silent context is deactivated, behaviour reverts to the default playing of a ringtone.

**Call Forwarding extension**  With the Call Forwarding extension calls will be forwarded to another preset number when a meeting is taking place. This feature is installed by means of a \texttt{require-module} call as was the case for Discretion:

\[(\text{require-module 'acme/telephony/call-forwarding})\]

The \texttt{acme/telephony/call-forwarding} module will load other two modules on which it depends: \texttt{acme/telephony/phone} and \texttt{std/activities/meeting}. The former contains basic functionality for ACME phones that was already installed for basic operation; it will not be reinstalled. The latter will be newly installed since the \texttt{std/activities/meeting} feature was not already available. From this point on, the phone will be aware of meetings —that is, there will be a \texttt{@meeting-context} prototype defined in the system.\(^\text{13}\) When the context discovery subsystem detects a meeting situation, it will activate this prototypical context:

\[(\text{activate-context @meeting-context}) \rightarrow \text{Switching @silent-context on} \text{ Switching @meeting-context on}\]

As can be seen, activating the \texttt{@meeting-context} implies activating related contexts as well, due to delegation relationships. Notably, \texttt{@meeting-context} delegates to \texttt{@silent-context}. Hence, behaviour that is adapted to silent environments will be active during meetings as well.

The Call Forwarding extension changes the call reception behaviour of the phone. The number to which calls should be forwarded can be specified naturally:

\[(\text{setf (forward-number *bobs-phone*)})\]
\[\quad (\text{make-phone-number 32 2 6470585})\]

\(^\text{12}\)The module is loaded in \texttt{@standard-context}, irrespective of the context that is current when the \texttt{require-module} call is made. The module must be able to acquire a reference to the objects that it will extend —it cannot forge references to extend arbitrary objects.

\(^\text{13}\)There will be a representation of meetings, but we do not include context detection functionality in the module to actually detect meeting situations in the external world.
### Table 7.2: Call receiving behaviour according to context combinations and call-forwarding setting.

<table>
<thead>
<tr>
<th>Off-hook</th>
<th>Silent</th>
<th>Meeting</th>
<th>Forwarding</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Ringtone</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>Vibrator</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>Vibrator</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Call forwarding</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Call waiting signal</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>Call waiting signal</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>Call waiting signal</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Call forwarding</td>
</tr>
</tbody>
</table>

The extension is slightly more complicated than it appears at first glance. Prior to installing the extension, ACME’s phone did not have a `forward-number` slot, and would not have understood the `(setf forward-number)` message just shown.\footnote{Setter methods in AmOS are invoked using Common Lisp’s `setf` special operator. The selectors of such setter methods are not atomic symbols, they are rather lists of the form `(setf field-name).`} When Call Forwarding is installed, existing objects such as `*bobs-phone*` are extended with new slots they did not have previously. The precise means by which this is achieved is explained further on.

Once a number has been set, received calls are forwarded to that number:

```
(receive (make-call 'alice) *bobs-phone*)  →  
Forwarding phone call from Alice to +32 2 6 47 05 85
```

This software extension overrides the behaviour specified for the default and silent contexts. Instead of activating the phone’s vibrator or ringtone, the call is forwarded to another number. If a forwarding number were not set (i.e. if it were `nil`), the extension has no effect whatsoever.

**Extension interaction** Table 7.2 illustrates the interactions of three contexts, namely `@off-hook-context`, `@silent-context` and `@meeting-context`. The “Forwarding” column represents the `forward-number` setting of the phone, rather than a context activation state. Not all 16 boolean combinations are interesting or even possible, and have thus been omitted from the table. In particular, the combinations where `meeting` is active and `silent` is inactive are impossible, because the activation of `@meeting-context` implies the activation of `@silent-context` by way of delegation. Further, the forwarding slot is unimportant when the `meeting` context is inactive. The interactions are thus reduced to eight possible and relevant cases, with four associated behaviours that can be exhibited by the phone.

ACME’s Discretion and Call Forwarding features are deployed as separate modules that can be installed at will by the user. These extensions are independent, meaning that they do not need each other to work correctly: none,
7.2. Case Studies

A meeting starts
\[\text{activate-context } \text{@meeting-context} \rightarrow \]
\begin{align*}
\text{Switching @silent-context on} & \quad \checkmark \text{ meetings are in particular silent} \\
\text{Switching @meeting-context on} & \\
\end{align*}

The meeting ends
\[\text{deactivate-context } \text{@meeting-context} \rightarrow \]
\begin{align*}
\text{Switching @meeting-context off} & \\
\text{Switching @silent-context off} & \quad \checkmark \text{ no silence any more} \\
\end{align*}

Snippet 7.10: Induced activation of contexts through delegation.

one or the two of them can be installed at any given time on the phone. Nonetheless, independence does not mean lack of interaction. The extensions do interact if both are installed on the same system, as can be observed in table 7.2. For example, behaviour from the Discretion extension (i.e. phone vibration) is observed in meeting context when there is no forwarding number set (third row of table 7.2); when forwarding is set, behaviour from the Call Forwarding extension is expressed (fourth row). If Discretion were not installed, the phone would ring during the meeting (the third row would be “Ringtone”); if Call Forwarding were not installed, the phone would vibrate during the meeting (the fourth row would be “Vibrator”).

Induced, nested and interleaved context activation To finish the explanation of the functionality illustrated by this scenario, one final note is in order on the interaction between context activations. As mentioned previously, context (de)activations must observe delegation relationships. The activation of @meeting-context induces the activation of @silent-context. Correspondingly, when @meeting-context is deactivated, @silent-context should be deactivated as well. Snippet 7.10 illustrates this case.\(^{15}\) Naturally, there is no reason to keep the phone’s behaviour adapted to quiet environments once a meeting is over. There are however situations in which @silent-context should not be deactivated transitively as happened in the previous example. Suppose the meeting was between doctors in a hospital. The hospital’s policy is that mobile phones should be kept silent within the floor where the meeting room is located. In such a scenario, the phones would be in @silent-context before the meeting even started. Once the meeting is finished, their original context should be restored to silent mode, instead of leaving the phones in default (noisy) mode. In other words, context activations must nest. Snippet 7.11 illustrates a nested context activation sequence. In this example, the dynamic extent of the meeting context is within the dynamic extent of the silent context. AmOS includes support for handling such situations correctly. It works also for interleaved context activation, as shown in snippet 7.12. In this case, the interleaved deactivation of @silent-context has no effect because

\(^{15}\)Notice in snippet 7.10 that the induced activation order is inverse to the induced deactivation order. In this scenario the order is unimportant, but it can matter in some situations.
the doctors enter the silent floor of the hospital
(activate-context @silent-context) →
Switching @silent-context on

the meeting starts
(activate-context @meeting-context) →
Switching @meeting-context on ✓ silence is already active

the meeting ends
(deactivate-context @meeting-context) →
Switching @meeting-context off ✓ silence is maintained

the doctors leave the floor
(deactivate-context @silent-context) →
Switching @silent-context off

Snippet 7.11: Nested activation of contexts.

(activate-context @silent-context) →
Switching @silent-context on

(activate-context @meeting-context) →
Switching @meeting-context on

(deactivate-context @silent-context) ✓ no effect
(deactivate-context @meeting-context) →
Switching @silent-context off ✓ correct at this point
Switching @meeting-context off

Snippet 7.12: Interleaved context activation.

the @meeting-context is still in force, and it depends on @silent-context to work correctly. By deactivating @meeting-context, @silent-context is “freed” and can be deactivated as well.

The sequences of context activations and deactivations discussed previously are performed by the context manager subsystem illustrated in figure 4.1 on page 58, rather than by base application logic. The mechanism that enables support for induced, nested and interleaved context activation is described in section 4.3.3 on page 77.

Implementation

We proceed to show the implementation of the most important objects and associated behaviour for the functionality described previously. To start with, the context configuration of this scenario is illustrated in figure 7.2. Most contexts are standard framework contexts, except for acme telephony, where the specific extra functionality of ACME’s phones is defined. The delegation link from phone to telephony means that every application which is in phone context is more generally in telephony context. The dependency of phone context on data and sound will be explained next.
Prototype definitions

For the sake of this scenario, phones are defined to have a speaker, a phone number (the one assigned by the operator), and a call manager:

1. `(defproto @phone (extend @prototype))`
2. `(add-slot @phone 'speaker (clone @phone-speaker))`
3. `(add-slot @phone 'number (clone @phone-number))`
4. `(add-slot @phone 'calls (clone @call-manager))`

The use of a speaker on line 2 explains the dependency of the phone context on the sound context illustrated in figure 7.2. The @phone-speaker prototype extends a more general @speaker prototype defined in sound context.

As shown on line 4, phones have a call manager. The call manager handles incoming, ongoing, terminated and missed calls:

1. `(defproto @call-queue (clone @queue))`
2. `(defproto @call-manager (clone @object))`
3. `(add-slot @call-manager 'active nil)`
4. `(add-slot @call-manager 'incoming (clone @call-queue))`
5. `(add-slot @call-manager 'ongoing (clone @call-queue))`
6. `(add-slot @call-manager 'terminated (clone @call-queue))`
7. `(add-slot @call-manager 'missed (clone @call-queue))`

The @call-queue prototype, defined on line 1, is a specialised form of queue for storing calls. The use of call queues explains the dependency illustrated in figure 7.2 of phone context on data context—the latter contains the definition of the @queue prototype. The manager has a slot which holds the currently active call (line 3), if there is any, and it features four queues for call management.
(defmethod receive ((call @call) (phone @phone))
  (advertise call phone)
  (enqueue call (incoming (calls phone))))

(defmethod advertise ((call @call) (phone @phone))
  (format t "Playing ringtone through ~a~%"
          (speaker phone)))

(with-context @off-hook-context
  (defmethod advertise ((call @call) (phone @phone))
    (format t "Playing call waiting signal through ~a~%"
            (speaker phone))))

(defmethod suspend ((phone @phone))
  (let ((call (active (calls phone))))
    (when call
      (format t "Putting ~a on hold~%" call)
      (setf (active (calls phone)) nil)
      (deactivate-context @off-hook-context))))

(defmethod resume ((call @call) (phone @phone))
  (format t "Resuming ~a~%" call)
  (setf (active (calls phone)) call)
  (activate-context @off-hook-context))

(defmethod answer ((call @call) (phone @phone))
  (format t "Answering ~a on ~a~%" call phone)
  (suspend phone)
  (enqueue call (ongoing (calls phone)))
  (resume call phone))

(defmethod hang-up ((call @call) (phone @phone))
  (format t "Terminating ~a~%" call)
  (when (eq (active (calls phone)) call)
    (suspend phone))
  (remove call (ongoing (calls phone)))
  (enqueue call (terminated (calls phone))))

;; These versions fetch the call from the phone's state:
(defmethod answer ((phone @phone))
  (let ((call (dequeue (incoming (calls phone)))))
    (if call
      (answer call phone)
      (format t "There are no incoming calls~%"))))

(defmethod hang-up ((phone @phone))
  (when (active (calls phone))
    (hang-up (active (calls phone)) phone)))

Snippet 7.13: Standard phone behaviour. It is defined in phone context.
(add-delegation @acme-telephony-context @acoustics-context)

(in-context @acme-telephony-context)

(with-context @silent-context
  (defmethod advertise ((call @call) (phone @acme-phone))
    (format t "Activating phone vibrator-%"))
)

(with-context (@silent-context @off-hook-context)
  (defmethod advertise ((call @call) (phone @acme-phone))
    (resend-bypassing-contexts (list @silent-context)))))

Snippet 7.14: ACME’s Discretion extension.

Behaviour definitions The basic behaviour of standard phones and phone calls is shown in snippet 7.13. Note that base application code is written in a natural style that is not concerned with context adaptation or particular extensions such as Discretion and Call Forwarding. The only occurrence of adaptation code on line 9 (i.e. the with-context construct) is for an intrinsic context, namely @off-hook-context. This adaptation code is defined in the same module where the basic behaviour is defined, and is fundamental to the basic operation of phones. Intrinsic contexts are defined during the domain engineering work performed at module design time. Contexts defined outside the module are considered extrinsic, and for most cases cannot and should not be foreseen. Anticipation is difficult and would probably result in overdesigned code. Even so, writing clean interfaces with respect to the application domain is still very important. In snippet 7.13, there is a clear separation of the different operations for call handling (receive, advertise, suspend and so on). This cleanly designed interface helps writing future extensions and extrinsic adaptation code without hassle, as shown next.

Extension definitions Although simple in terms of code size, the extensions illustrate the use of two advanced language features, namely bypass resends and sealed objects.

Discretion extension The Discretion feature is implemented as shown in snippet 7.14. First of all, the telephony context defined by ACME is extended on line 1 to become aware of acoustics. This makes the @silent-context available in ACME’s telephony context, so that it can be used to specialise the advertise method on silent environments. As seen on line 7, the new behaviour uses the phone vibrator for advertising the call.

When the phone is off-hook (the user is talking), it should not vibrate, even if @silent-context is active. It would feel quite bizarre to suddenly receive vibration on the ear while talking with someone. A call waiting signal should be played instead. To account for such situations, one more version of advertise is defined starting on line 9. When @silent-context and
@off-hook-context are active simultaneously, the Discretion extension will give priority to off-hook behaviour over silent behaviour. This is achieved by means of resend-bypassing-contexts, explained in section 5.2.9. This call chooses the next most specific method that is not specific of silent contexts.\textsuperscript{16} Hence, the specialised version of advertise defined on line 6 will not be applicable, and the default version will be chosen instead (or whatever other version that is applicable at the moment). If the advertise behaviour has not been overridden by some other extension, the phone will play a soft call waiting signal on the speaker as is normal in off-hook context.

There is a second way of achieving the same bypassing effect just illustrated, which does not use the special resend-bypassing-contexts call. It uses more familiar constructs such as without-contexts\textsuperscript{17} and resend:

\begin{verbatim}
(with-context (@silent-context @off-hook-context)
  (defmethod advertise ((call @call) (phone @acme-phone))
    (without-context @silent-context
      (resend)))))
\end{verbatim}

Although for the scenario this version would work just as well, it has a problem that could become apparent in other scenarios. The resend call is made with an inactive @silent-context. This means that whatever behaviour is eventually chosen by resend will \textit{not} be adapted to silent environments, but rather be meant for default acoustics. Conceptually, it is wrong to disable the silent context in this rather drastic way, given that the physical environment has not actually changed —there is a potential mismatch between the outside world and the logic encoded in the method.

\textbf{Call Forwarding extension} Though simple, call forwarding behaviour helps illustrating an important feature needed for seamless extension of existing code. The code is shown in snippet 7.15. A new slot forward-number is added to ACME’s phone on line 3. If the slot’s value is \texttt{nil}, it means the user has not set his preference for call forwarding, and the original behaviour will be invoked (line 9). Since meeting context delegates to silent context, the phone will vibrate if Discretion is installed, or a ringtone will be played otherwise during meetings. If on the other hand the user did specify a forwarding number (the slot is not \texttt{nil}), incoming calls will be forwarded during meetings (line 8).

The forward-number slot added to the @acme-phone prototype on line 3 is seen by delegating objects such as *bobs-phone*. Hence, *bobs-phone* will understand the new forward-number message. Suppose now that Bob sets his preference for call forwarding:

\begin{verbatim}
(setf (forward-number *bobs-phone*)
  (make-phone-number 32 2 6470585))
\end{verbatim}

\textsuperscript{16}The resend-bypassing-contexts construct is a kind of context-oriented super call, although more than one context can be specified, whereas super always has exactly one (implicit) argument, the class where it occurs.

\textsuperscript{17}The without-context construct deactivates the given contexts before executing its body, and then activates them back; it is the reciprocal of with-context.
7.2. Case Studies

Snippet 7.15: ACME’s Call Forwarding extension.

Given that *bobs-phone* does not have a forward-number slot, the slot-changing (setf forward-number) message will be delegated to @acme-phone, which will understand the message and change the slot’s value. The preference set by Bob will be seen by all objects delegating to @acme-phone. This is known as the prototype corruption problem [12]. It will happen for all clones of @acme-phone that were created before the Call Forwarding extension was installed. On the contrary, clones created afterwards will feature the new slot and pose no problem.

To seamlessly support extensions such as Call Forwarding without giving rise to the prototype corruption problem, we use the notion of sealed objects discussed in section 5.2.10 on page 106. The slots of sealed objects, called sealed slots, cannot be modified indirectly by delegation. An attempt to write a sealed slot will result in a new slot with the new value being created on the delegating object, instead of changing the state of the sealed object.

Discussion

The validation points are addressed as follows:

**Dynamic adaptation (V1)**

This study case shows adaptation of behaviour to external properties of the device’s logic environment, namely whether the current location is supposed to be silent, and whether a meeting is taking place. Internal physical properties such as remaining battery power are not taken into account.

Given that a direct relationship is established between the device and its external environment, this case study is closer to typical Ambient Intelligence (AmI) scenarios [43] than the previous one. Adaptation to semantic contexts such as being in a silent place or attending a meeting is partly what is intended by “intelligent” in AmI.

**Clean application logic (V2)**

Snippet 7.13 on page 160 shows the simple encoding of call-management
logic, irrespective of the place, the situation, or any other property of the
environment in which such code will run. The extension code in snippets
7.14 and 7.15 is cleanly separated and simple as well.

**Simplified application logic (V3)**
Context-oriented programming is exploited by defining a context to which
call-advertising behaviour adapts, namely `@off-hook-context`. This con-
text is inherent to the inner workings of the phone and is unrelated to ex-
ternal physical and logical factors.

Thanks to the use of `@off-hook-context`, call-advertising logic is not hard-
coded as a conditional statement in the body of `advertise`. If this were not
the case, `advertise` would be less adaptable.

**Non-intrusive adaptations (V4)**
Basic call-management code in snippet 7.13 requires no intervention in or-
der to introduce the adaptations of `advertise` and `receive`. Similar non-
intrusive adaptations for other methods can be thought of: method `suspend`
could be adapted by a Waiting Music extension that plays a tune when a
user is put on hold, for example. However, for the sake of the case study, we
aimed at having extensions that interact with each other.

**Straightforward architectures (V5)**
As illustrated in figure 7.2, framework code is cleanly modularised and in
response with the application’s domain which is about phones, tele-
phony, sound and basic data structures.\(^\text{18}\) Regarding reuse, the framework
code for call management shown in snippet 7.13 does not depend on the
particular constructor of phones.

Provider code is contained in two modules, `acme/telephony/discretion`
and `acme/telephony/call-forwarding`, both running under a same con-
text, `@acme-telephony-context`, since both come from the telephony di-
vision from ACME, and are thus meant to run under that context. Even if
their “home” context is the same, separating the extensions in two different
modules makes it possible to distribute and install them independently.

**Dynamic addition of software features (V6)**
The Discretion and Call Forwarding extensions can be modularised as in-
dependent software bundles and be added to code already running in the
mobile phone. Previously existing objects can be extended with new slots
and new behaviour added by different modules. The addition of a delegation
to `@acme-telephony-context` in snippet 7.14 shows that contexts are not
fixed entities: they may variate as new features are added.

Besides the main validation points V1–V5, this case study explores modular-
isation of extensions for enabling context-aware behaviour, and the way these

---

\(^{18}\text{The source code of some of these modules, in particular std/sound/speaker and std/data/
queue, is not shown since it does not contribute to the case study.}\)
extensions interact. The extensions turn the phone from context-oblivious to context-aware. A second aspect explored in this case study is ordering of activations for contexts that have dependencies (i.e., that delegate to other contexts). Regarding language features, the case study shows the use of the special \texttt{resend-bypassing-contexts} method and explains the need for such construct.

### 7.2.3. Transactional Contexts

The validation cases presented previously are typical examples of the kind of dynamic behaviour adaptation expected from ambient applications. In this section we present a rather different use of our approach. We show how to implement support for lightweight transactions—that is, actions that are \textit{atomic} and \textit{abortable} \[118\]. We start from a brief scenario describing typical requirements to be addressed by a transactional system:

ACME's Ambient Shopping System (AmbiShop) supports automatic payment of goods thanks to Radio Frequency Identification (RFID) tags put on all stock items. AmbiShop uses FIFO accounting for managing the inventory—it regards the first unit that arrived in inventory as the first one sold. Hence, inventories are basically queues from which the dealer draws items and sells them to customers.

Items are drawn from the inventory in order. Each item is accounted for and put in the client's order. If a problem arises in the middle of the checkout operation (for instance, if the client runs out of credit), the whole order is cancelled and the client is notified.

To support proper cancellation of the checkout operation, we use transactions. The transactions we define are atomic ("all or nothing") units of work in which arbitrary objects are manipulated. The modifications made to objects are either effective as a whole, or ineffective altogether. Hence, if it is decided that the changes should not be effective in the middle of execution of a transaction, the programmer needs not care about undoing the modifications that have been done to every single object since the transaction began—this might even be impossible, as the programmer does not necessarily hold a reference to all affected objects.\(^\text{19}\) Transactions can be aborted intentionally or due to an error.

Besides atomicity, a second important property of lightweight transactions is isolation. The effects of operations made within a transaction should be invisible to operations outside the transaction. Isolation is relevant in concurrent systems. This validation case is concerned with atomicity only, as adding proper concurrency abstractions to the model is part of our future work. With concurrency in place, the transaction mechanism we describe could be the basis for supporting Software Transactional Memory (STM) \[76\]. However, the goal of this validation case is not to show a full STM system; rather, it is to assess

\(^{19}\)Even more so in a system that has been designed with the Principle of Least Authority in mind \[80, 81\].
the expressiveness of our approach. In doing so, we show its usefulness and applicability as a possible foundation for STM systems.

**Expected Functionality**

Before explaining the underlying implementation of the transactional system, we show user code and expected observable behaviour of the system. To implement the inventory, the user creates the `@inventory` and `@item` prototypes shown in snippet 7.16. In this case study an inventory is simply a specialised queue. Inventory items have a description and two timestamps: the date in which the item entered the inventory, and the date in which it was taken out. If the entrance date is `nil`, the item has not entered the inventory yet; if the exit date is `nil`, it has not left the inventory yet.

In snippet 7.17 an inventory and order are created. The inventory contains a few initial items. The state of inventories can be inspected:

```lisp
describe *inventory* →
  potatoes (in: 2008/06/02 21:53)
  tomatoes (in: 2008/06/02 21:53)
  oysters (in: 2008/06/02 21:53)
```

The description shows the time at which the item entered the inventory.

During the sell operation, items are dequeued from `*inventory*`, processed in some way, and enqueued in `*order*`. This logic is succinctly encoded as shown in snippet 7.18. There are three main steps being performed in the body

20 Admittedly, the sample stock items in snippet 7.17 are not typical of an AmI scenario—it is hard to imagine potatoes, tomatoes and oysters with RFID tags. The examples are inspired on a song by Ella Fitzgerald and Louis Armstrong, “Let’s Call the Whole Thing Off”.
7.2. Case Studies

Snippet 7.18: Atomic treatment of inventory and order queues.

```lisp
(atomic
  (loop
    until (emptyp *inventory*) do
      (enqueue (process (dequeue *inventory*)) *order*))
)
```

Snippet 7.19: Simple item processing.

```lisp
(defun process ((item @item))
  (format t "Processing -a-%" item)
  (setf (exit-date item) (get-universal-time))
  item)
```

of the loop; two of them — enqueue and dequeue — are standard queue operations. The third is the process method which abstracts away the bookkeeping performed for each item that is being transferred from the inventory to the order. In this case study, process simply timestamps the item with an exit time and returns the item, as shown in snippet 7.19.

The loop in snippet 7.18 is repeated until the inventory is empty. The most important part in this example is the atomic form that surrounds the loop; atomic ensures that the enclosed body is performed atomically. Once the atomic block finishes, either all changes made are visible at once, or, in case of error, no single change is visible, as if the atomic block never executed. If an error arises while processing an item, the programmer needs not determine the point at which the loop stopped working and have a “reverse” loop undo the changes. Further, the programmer needs no “unprocess” operation to undo the processing performed by process for already transferred items. In complex situations, implementing reverse application logic can be difficult and error prone.

If the transaction encoded in snippet 7.18 executes successfully, all items will have been transferred:

```lisp
(describe *inventory*) →
  empty inventory
(describe *order*) →
  potatoes (in: 2008/06/02 21:53 out: 2008/06/02 21:56)
  tomatoes (in: 2008/06/02 21:53 out: 2008/06/02 21:56)
  oysters (in: 2008/06/02 21:53 out: 2008/06/02 21:56)
```

The process method has timestamped all items with an exit time.

To show the behaviour in case of failures, we create a @failing-context that implements a faulty version of process, shown in snippet 7.20. The call to random yields either 0 or 1 with the same probability. Hence, the new version of process will fail roughly half of the time. If it does not fail, it simply invokes the correct behaviour by means of a resend call. This use of contexts shows in passing one more interesting application of our approach. The creation of
Assessment of Expressiveness of AmOS

(defcontext @failing-context :uses (@standard-context))

(with-context @failing-context
  (defmethod process ((item @item))
    (when (> (random 2) 0)
      (error "spurious error"))
    (resend)))

Snippet 7.20: Faulty item processing logic.

(activate @failing-context)

(handler-case
  (unwind-protect
    (atomic
      (loop
        until (emptyp *inventory*) do
          (enqueue (process (dequeue *inventory*)) *order*))
    ;; print this even in case of error:
    (format t "*** After ***-%")
    (format t "Inventory:-%")
    (describe *inventory*)
    (format t "Order:-%")
    (describe *order*)
    (error (e)
      (format t "Caught -a-%" e))))

Snippet 7.21: Test code of atomic transaction execution.

testing contexts can be used to inspect or intercede the behaviour of the system without cluttering production code with testing code. Furthermore, the testing context can be activated at certain points and deactivated at others to reduce the scope of the test, produce less logging messages, and so forth.

Starting from the inventory of snippet 7.17, but having this time faulty process behaviour (after having activated @failing-context), we get different output from snippet 7.18 if there is an error:

(describe *inventory*) →
  potatoes (entrance: 2008/06/02 22:44)
  tomatoes (entrance: 2008/06/02 22:44)
  oysters (entrance: 2008/06/02 22:44)
(describe *order*) →
  empty inventory

Both queues are left untouched, as well as the items they contain — a sign of this in the output is that none of the items has an exit date.

To give a more concrete feeling, a non-simplified version of the code we used to execute the transaction is shown in snippet 7.21. The unwind-protect
7.2. Case Studies

Processing potatoes
Abort transaction
*** After ***
Inventory:
potatoes (entrance: 2008/06/02 22:44)
tomatoes (entrance: 2008/06/02 22:44)
oysters (entrance: 2008/06/02 22:44)
Order:
Empty inventory
Caught spurious error

Snippet 7.22: Extended output from atomic transaction execution.

call form standard Common Lisp helps us print a trace even when an error is signalled by process. The handler-case form catches the error and prints it on the console, instead of popping up a debugger in the Common Lisp IDE.\(^\text{21}\) The output produced by snippet 7.21 is shown in snippet 7.22. In this extended output it can be seen that the first item (potatoes) is processed successfully. Hence, the first item has been transferred from *inventory* to *order* and the item has been timestamped. The next call to process made by the loop fails, leading to abortion of the current transaction. The state of the inventories is printed (showing no modification to the objects), and the final line shows the spurious error generated by the faulty process routine.

Implementation

Language support for lightweight transactions in modern object-oriented languages is relatively recent [62]. Harris et al. [63] show an atomic construct that is similar to the one presented here. The mechanisms used to support such atomic transactions are however quite different. We use three advanced features of AmOS:

Contexts Above all we wish to highlight in this case study the use of contexts: contexts are at the base of our implementation of transactions. To the extent of our knowledge, this is the first approach of this kind.

Become We use a become operation inspired on the Actors model [3]. Thanks to this operation, an object can supplant another one, without losing its identity. Conceptually the object remains the same, but it assumes the behaviour of another given object, like a human does while acting. A become-like operation is available in many models that have drawn inspiration from Actors, and a few others that have not, such as Smalltalk.

Computational reflection We rely heavily on the reflective facilities AmOS provides, since reflection makes the implementation rather succinct and easy.

\(^\text{21}\)The handler-case form is like try/catch in Java. The unwind-protect form is like the finally portion of try/catch [98].
Assessment of Expressiveness of AmOS

(defun atomic (&body body)
  (let ((transaction (clone @transaction)))
    (handler-case
      (with-context transaction
        ,@body)
      (error (e))
      (abort transaction)
      (error e))
    (:no-error (result)
      (commit transaction)
      result))))

Snippet 7.23: Implementation of \texttt{atomic}.

to understand. As always there is no free lunch: the cost is paid in
performance. Reflection is nevertheless not essential and could be traded
for speed, at the expense of generality. This is discussed at the end of the
case study. Nevertheless, we decided for the elegance brought by reflection
and meta-object protocols [75, 70].

The following presentation is divided in a number of distinguished paragraphs
to ease comprehension.

\textbf{Transactions as contexts} The transactions we have defined are simply con-
texts. We are thereby able to harness the existing context machinery to execute
code in \textit{transaction context}. Thanks to existing machinery, the \texttt{atomic}
form is implemented easily as a thin wrapper around \texttt{with-context} —the usual con-
struct for execution of a body in a particular context. The implementation of
\texttt{atomic} is shown in snippet 7.23. The \texttt{atomic} form is simply syntactic sugar (a
macro) to make managed invocations of \texttt{with-context}, as explained next. The
code wrapped by \texttt{atomic} is received in quoted form\textsuperscript{22} as the \texttt{body}
parameter of the macro (line 1). A clone of the prototypical \texttt{@transaction} is created
and the body is executed in that fresh transaction context (lines 4 and 5). If
an error is signalled, the transaction is aborted and the error is forwarded to
outer error handlers (by signalling it again in line 8). Hence, the \texttt{atomic}
form is transparent with respect to error handling. If the body completes without
error, the transaction is committed and the result of the body’s execution is
returned.

As shown in snippet 7.24, transactions are specialised forms of contexts that
have a \texttt{log} (line 2). The use of \texttt{amos-core:add-slot} is explained further on.
The log is a hash table mapping objects to their \texttt{alter-objects}.\textsuperscript{23} Alter-objects are
the transactional versions of objects that have been modified in transaction
context. Given a transaction and an object, the \texttt{alter-object} method defined
on line 4 returns the transactional counter part of the object, by fetching it

\textsuperscript{22}Also called an s-expression in Lisp parlance.
\textsuperscript{23}By analogy to \textit{alter ego}, “the other I” in Latin.
7.2. Case Studies

```lisp
(defcontext @transaction)
(amos-core:add-slot @transaction 'log (make-hash-table))

(defmethod alter-object ((transaction @transaction) object)
 (let ((log (log transaction)))
   (or (gethash object log)
       (let ((alter-object (clone-object object)))
         (setf (gethash object log) alter-object
               (setf (gethash alter-object log) alter-object)))))))
```

Snippet 7.24: Transactional context definition.

from the log (line 6). If the object does not have an alter-object yet, a new
one is created (line 7), and added to the log. Not only the original object is
associated to its alter-object (line 8), but also the alter-object is associated
to itself (line 9).\(^\text{24}\) Without the second association, the system would create
transactional versions of alter-objects (alter-objects of alter-objects), and the
chain would continue endlessly this way, producing transactional objects of
transactional objects (alter\(^n\)-objects). There is no reason to keep protected
versions of objects that already belong to, and have been modified in, the
current transaction.

The use of `amos-core:add-slot` instead of the regular `add-slot` method on
line 2 of snippet 7.24 has to do with stopping infinitely recursive calls when
the transaction log is read by `alter-object`. The `amos-core:add-slot` call
creates a direct log accessor method that bypasses regular message passing,
so that its invocation is not intercepted by transaction management logic.
Hence, a recursive invocation of `alter-object` on line 5 is avoided. The use
of `amos-core:add-slot` implies that the case study is not implemented purely
on top of the object model; this is discussed later on.

**Isolation of destructive operations** In transaction context, destructive opera-
tions such as `add-slot`, `remove-slot` and `(setf slot-value)` are intercepted
and recorded in the current transaction log, instead of letting them modify their
target object. Destructive operations are intercepted by defining transaction-
specific versions, as shown in snippet 7.25. When a transaction context is act-
ive, the specialised versions will be applicable (and they will be more specific
than the default versions). The three operations use the `alter-object` method
shown previously to create a transactional version of their `object` argument,
or fetch it from the log if it already exists. The `host-activation` calls used
as argument of `alter-object` yield the transaction that is currently active, as
explained below. For each destructive method, the `object` argument is set to
the corresponding alter-object and the message is resent. When invoked, the

\(^{24}\)Common Lisp’s `(setf place value)` general assignment operator can be explained by
analogy: an assignment that in some language would look like `v[5] = 4711` is expressed
in Common Lisp as `(setf (aref v 5) 4711)`. (Example taken from comp.lang.lisp.) In
snippet 7.24, the assignment on line 8 is analogous to `log[object] = alter-object`. 
(with-context @transaction
  (defmethod add-slot (object name value)
    (setf object (alter-object (host-activation) object))
    (resend))
  (defmethod remove-slot (object index)
    (setf object (alter-object (host-activation) object))
    (resend))
  (defmethod (setf slot-value) (value object (index @natural))
    (setf object (alter-object (host-activation) object))
    (resend)))

Snippet 7.25: Destructive operations specialised on transactional context.

(with-context @transaction
  (defmethod send (selector arguments)
    (let ((log (log (host-activation))))
      (do-slots (arguments argument :index index)
        (setf (amos-core:slot-value arguments index)
          (gethash argument log argument)))
      (resend)))

Snippet 7.26: Message sending behaviour for transactional contexts.

original versions of the methods will modify the state of alter-objects, instead of modifying original objects. Non-destructive operations need not be specialised.

The host-activation call yields the first, implicit lookup argument where the method that is currently executing was found. This implicit argument is the same one used by inner accessor methods and by resend methods, for instance. Therefore, “host activation” is a synonym of “host context”. Since transactions are contexts, this means that “host activation” is just a synonym of “host transaction” —the transaction where the currently executing method has been found.

Besides destructive operations, one more reflective method needs to be specialised on transaction context, although not for logging purposes. As application code executes, the system must make sure that any method invoked in transaction context sees the transactional versions of modified objects (i.e. the alter-objects), rather than the original versions. The send method is specialised on @transaction context for this purpose, as shown in snippet 7.26; send is part of the behavioural Meta Object Protocol (MOP) of AmOS explained in section 5.3.1 on page 108. The do-slots form iterates over the arguments. It replaces each argument of the original message by the version in the current transaction’s log. If the argument has not been modified in transaction context and thus has no associated alter-object, it remains unchanged (gethash returns its third parameter if the given key is not found in the hash table; in this case the third parameter is the original argument of the message). Note that the (setf amos-core:slot-value) primitive is used on line 5 of snippet 7.26 to
set the value of each argument, instead of using the regular \texttt{(setf slot-value)} message. Sending a message within the \texttt{send} method leads to infinite recursion. For this reason, \textit{all} invocations shown in snippet 7.26 are primitives, except the \texttt{resend} invocation on line 7, which is done through regular message passing, although not reflectively (i.e. not by invoking \texttt{send}).\footnote{The specific case of \texttt{resend} is hard-wired in AmOS, due to its common use.} Hence, no infinitely recursive calls occur.

This completes the explanation of how reflective methods are specialised on transactional context to manage destructive operations and thus achieve transaction isolation. Unlike other study cases presented in this chapter, our support of transactions is not implemented \textit{purely} on top of the object-oriented model. As illustrated throughout the previous presentation, we must use primitives at certain places to avoid infinite transaction logging chains and recursion chains. This is a natural consequence of interceding the most fundamental mechanisms of the model such as \texttt{send} and \texttt{(setf slot-value)}. This case study is aimed at illustrating the potential of context-oriented programming in its intersection with computational reflection, rather than concentrating on a pure use of the object model.

\textbf{Transaction management} Snippet 7.27 shows the implementation of main transactional operations. The \texttt{clone} method is overloaded for transactions so that each newly created transaction has a fresh log (created by \texttt{reset}). Aborting a transaction simply means forgetting the context (a standard context operation explained in section 5.3.2); \texttt{forget} is specialised on transactions to reset the log, thereby freeing held resources (i.e. letting the garbage collector reclaim the alter-objects referenced by the log).

When a transaction is committed, the modified versions of objects recorded in the log replace unmodified versions. To this end, the transactional log is traversed by means of a \texttt{maphash}\footnote{Common Lisp's standard \texttt{maphash} function calls its first argument for each key/value pair stored in the hash table passed as second argument.} call on line 16; the Actor-inspired \texttt{become} operation \cite{3} is used to substitute the modified version stored in the log for the unmodified one.

Normally, having every alter-object become the original object as explained previously should be sufficient to commit the transaction. However, in our implementation, contexts are not protected by transactions. This means that the information held in transactional contexts needs to be copied back to non-transactional contexts by hand. We explain this process next. The \texttt{dolist} loop iterates over all context combinations that include the transaction as component context. The \texttt{find-combinations-including} call on line 18 finds these combinations. The combinations depend on the transaction because they delegate to it. Figure 7.3(a) shows a sample $A+B+C$ combination of three component contexts $A$, $B$ and $C$. When transaction $T$ is activated, a new $A+B+C+T$ combination is formed, which delegates to the original combination and to $T$. This is just the normal workings of the context combination mechanism of AmOS, there.
is nothing particular to \( T \) being a transactional context. For each dependent combination, the original combination that does not include the transaction is found (or created otherwise) thanks to a combination-excluding call. The transactional combination is merged with the original combination, as depicted in figure 7.3(b). The merging operation copies the roles of the source object (the first argument) to the destination object (the second argument), so that behaviour defined on the transactional combination becomes visible on the original combination. Once the information has been copied from all transactional contexts to their non-transactional counterparts, the former are dropped to free resources (lines 23 and 24).

Discussion

This case study is mainly a lateral thinking experiment on novel ways to use Context-Oriented Programming (COP). Instead of showing yet another typical AmI scenario, we set out to explore the feasibility of using contexts as transactions. Conceptually, we find it intuitive to think that a transaction constitutes a special context in which code is run. Objects are seen from a “transactional perspective”, and from this special point of view, their slots and behaviour...
are different than those observed from non-transactional perspectives. This matches neatly the notion of subjective objects by Smith and Ungar [101]. Technically, the simple transactional system we have explored provides evidence that contexts and transactions make indeed a good match. We intercept all message sends and all calls to destructive operations by specialising reflective operations, so that every single object is protected in transaction context. If activation contexts²⁸ were supported, the transactional system described in this case study would automatically provide isolation (besides atomicity), because transactional contexts would be local to a closure’s activation, and thereby isolated from any other activation.

The use of reflection to intercept all sent messages might seem overkill in some cases, and above all, it can be slow. However, the specific approach we took by using reflection is not the only possible one. Efficiency can be increased significantly if `send` and `(setf slot-value)` (two very frequent operations) were not reflective. Supposing they were not, it would still be possible to protect selected slots of an object by installing “protective” accessor methods that are specialised on transaction context. These protective accessor methods (for example, accessors for the slots `age` and `height` of a person) would read from and write to a slot located in the transaction on which they are specialised, instead of affecting the original slot of the object. If the transaction were committed, the value of all protected slots would be written back to the original object; if it were aborted, nothing would need to be done. Once the transaction context were deactivated, the protective accessors would not override the original accessors anymore. A construct like the following could be devised:

```lisp
(with-atomic-slots person (age height) ...)
```

This solution would trade speed for generality, but could still be useful in some situations.

²⁸Contexts that are local to a given closure, instead of global. Activation contexts are part of our future work, described in section 8.3 on page 190.
Validation points

This case study is different from the previous ones in that it “goes meta”. In this case study, the application is AmOS itself, and the adaptation is the transactional system we have defined. The validation points are addressed as follows:

Dynamic adaptation (V1)

The case study shows the adaptation of system behaviour to an internal property of the logic environment, namely being in transactional context. The \texttt{@transaction} context is much like the \texttt{@low-battery} context defined in the case study about mobile phones, except that battery charge is a physical base-level property, rather than a logical meta-level property of the internal environment.\footnote{Recall figure 4.2 on page 59 about context types.} Fundamental system behaviour —the behaviour of AmOS— is adapted to transactional context at run time.

Clean application logic (V2)

Transactions are an \textit{extrinsic} kind of context: they were not part of AmOS when we set out to make this case study, and they do not make part of AmOS’s main application logic. Hence, the implementation of AmOS is oblivious to transactions. Its logic is expressed cleanly (although we do not show the source code of AmOS in the dissertation).

Regarding adaptation code, the adapted versions of MOP methods in snippet 7.25 on page 172 are written straightforwardly. It could be thought that validation point V2 is weakened by the use of a primitive operation, namely \texttt{amos-core:slot-value}, in snippet 7.26 on page 172. This adaptation code needs to bypass the computation model for proper implementation. We believe however that this should not be regarded as a paradigmatic shortcoming of our approach. Rather, it is a paradigmatic shortcoming of reflection: such paradoxes are observed regularly in systems that are about themselves, as witnessed by the metaclass regression problem encountered in Smalltalk. Such regression problems must be shortcuted in some way. We are not aware of an elegant way of placing a shortcut \textit{within} the model, but again, this is an exercise in reflection,\footnote{The problem stated generally is: when messages are sent within an overloaded version of \texttt{send}, can we avoid the natural infinite recursion that arises?} not a problem of our COP approach.

Adaptation code which is not reflective, in snippets 7.23, 7.24 and 7.27, also encodes transaction logic cleanly.

Simplified application logic (V3)

This very case study does not show the way context-oriented programming is exploited within AmOS, simply because it does not show the implementation of AmOS, but parts of this implementation are available elsewhere, in particular in snippet 5.1 on page 104. The \texttt{with-context} construct is used to define \texttt{resend} in method activation context. Hence, the \texttt{resend} method is
specific to “situations in which a method is being executed” —it can be seen, at the meta level, as adaptation code. The use of `@method-activation` as an intrinsic context for the implementation of `resend` in snippet 5.1 is analogous to the use of `@off-hook` context for the implementation of `advertise` in snippet 7.13 on page 160 for “situations in which a call is taking place”, with the difference that the former intrinsic adaptation occurs at the meta level, whereas the latter is a base-level intrinsic adaptation.

**Non-intrusive adaptations (V4)**

The implementation of AmOS did not need any modification to accommodate the transactional extension shown in the case study. This was made possible thanks to its MOP. At the meta level, the MOP of AmOS can be seen as a context-oriented framework which allows adaptation to different contexts (in this case, transactional context). This is analogous to base-level framework code presented for other case studies, for instance snippet 7.13 on page 160, in which the behaviour of mobile phones is defined through a well-designed set of prototypes and methods. Having clean interfaces, whether a meta-object protocol or a mobile phone object protocol, increases the possibilities for context adaptation.

**Straightforward architectures (V5)**

Clearly, we did not bias the meta-level architecture of AmOS (i.e. our computation model) towards supporting transactions when we developed it. To provide additional evidence for this validation point, we would need to show that AmOS is “reusable”. We would therefore need to implement another meta-extension to this end, which “reuses” the MOP. However, we deem it sufficiently clear that the architecture of AmOS has been designed cleanly and straightforwardly, following the simplicity, homogeneity and flexibility principles discussed in section 1.4 on page 5 —this is supported by chapters 5 and 6.

This last case study is an indication of what seems to be an interesting field of research, *context-oriented reflection* —a particular case of context-oriented programming in which the object model itself becomes adaptable.

### 7.3. Wrap Up

Through the case studies presented in this chapter, we have shown how our approach permits the clean expression of applications that can adapt dynamically to changing intrinsic, extrinsic, physical, logical, internal and external contexts. Further, adaptation logic can be introduced without modifying application logic.

The software architecture of context-oriented applications in our approach is divided in modules —contexts bundling a number of prototypes and their behaviour. Dependencies among modules are encoded as delegation links which can be managed dynamically, for instance to add new modules which we regard
Assessment of Expressiveness of AmOS

Prototypes
- Idiosyncratic contexts
- Separate software features
- Unanticipated new features
- Dynamic composition of features*
- Dynamic behaviour adaptation
- Dynamic software feature (de)activation*
- Context-oriented composition

Table 7.3.: Summary of language features (vertical) and their related advantages (horizontal) as a result of the validation presented in this chapter.

Modules are layered in three categories, namely system, framework and provider. The system layer contains basic AmOS functionality, including the Context Object Protocol and the Meta Object Protocol. The framework layer provides a common ground for cooperation of independently developed modules. This is achieved by defining standard prototypes, including their prototypical behaviour. These prototypes constitute shared vocabularies that enable application interoperation. The case studies show that cleanly designed framework protocols facilitate reuse and adaptability to context. Finally, the provider layer contains specific code that uses or extends framework code to furnish vendor-specific services. These services might depend on (e.g.) specific hardware modules, and generally encode the intricacies of each provider’s logic.

Table 7.3 summarises the language features of AmOS we have used in the validation cases and their associated advantages. The advantages marked with an asterisk are made possible by our approach, but proper support requires further refinement of our techniques for the following reasons:

- Dynamic composition of features is currently limited by the linearisation semantics of method specificity. Having many small behavioural pieces (i.e. multimethods) that might be applicable for any requested interaction (i.e. messages), behaviour composition might become an issue. Flexible method combination techniques might become necessary to deal with all behaviour that is applicable for a given message. AmOS does not yet incorporate advanced method combination techniques as those of Common Lisp Object System (CLOS) for example, or as suggested by Harrison and Ossher [64]. In AmOS, all applicable methods are linearised, and more spe-
Specific methods can decide at their discretion to invoke less specific methods by means of constructs such as `resend`, `resend-bypassing-contexts` and `resend-as`. The downside is that automatic method linearisation does not necessarily yield the “fittest” order in which to execute applicable behaviour [104]. More declarative and intentional approaches such as `predicate dispatch` [48] and `filtered dispatch` [31] could be used instead of, or in complement to, automatic linearisation.

- Regarding dynamic software feature activation and deactivation, we still need to provide adequate support to prevent the concurrent deactivation of contexts that are being used; this problem is discussed in section 4.3.3 on page 74.

Despite the limitations mentioned previously, the systematic assessment of properties V1–V5 we claimed about our approach, and the verification that they do hold for the three case studies presented in this chapter, gives us confidence on the suitability of our programming model for the development of dynamically adaptable applications and clean expression of behaviour dependence on context. Needless to say, more advanced case studies and further validation need to be carried out in the future.
8 Conclusions

Appealed by the perspective of Ambient Intelligence (AmI) [43], our holy grail since the beginning has been the support of dynamic behaviour adaptation to context. We decided to approach this objective from a language engineering perspective. Proof that the time is ripe for this endeavour is the current emergence of a new programming paradigm, Context-Oriented Programming (COP) [65]. COP is aimed at supporting dynamic software adaptation to context through the design of dedicated language abstractions.

In this chapter we summarise the fruits of our research, discuss related work, and finish by showing salient open paths that require further exploration.

8.1. Contributions

As a whole, the contribution of this dissertation is the exploration of a novel computation model that is particularly well suited to COP. We framed that exploration in a more general view of context-aware systems and their architecture, so that the interactions with other parts of the system were well understood, and the scope of our work could be delimited clearly.

Having explained our approach in detail throughout the dissertation, we are now able to pinpoint precisely the contributions that were put forward in the introduction. We revisit the high-level view of contributions in terms of combining different programming paradigms, but this time we interweave supporting discussions and technical details in hindsight.

Subjective Programming

In subjective programming, exhibited object behaviour not only depends on the objects sent along in a message, but also on the sender of the message. The subjective “point of view” of the sender thus affects perceived behaviour of applications. This way, a same application can behave differently depending on the particular user and on the particular situation in which the user is participating.

Objective behaviour is a particular case of subjective behaviour, merely by keeping a fixed point of view. As this dissertation demonstrates, departing from this trivial case brings many benefits that make a case for adoption of subjective behaviour in object-oriented languages—even more so when applied to areas such as Ambient Intelligence in which the execution environment of applications is of prime importance. Following this line, the dissertation shows the way our approach can be used to implement typical AmI scenarios. This not
only constitutes a validation of the approach, but also it contributes insight to an area that is largely unexplored and thus little understood to date, namely subjective programming and subjective object behaviour. We have contributed to subjective programming by pushing subjectivity to the front and exploring it in advanced ways that we have hitherto not seen elsewhere.

**Prototypes with Multiple Dispatch (PMD)**

We implemented the PMD model through the Ambient Object System (AmOS), and enriched it with a novel mechanism, *lookup arguments*, which allows the elegant expression within the object-oriented model of accessor methods, resend methods, and other fundamental mechanisms, rather than have them be special constructs, or oblige us to “hack” parts of the model. Lookup arguments helped us homogenising the implementation of the model, and provided support for implementation of an advanced extension, namely lightweight transactions.

We contribute by providing a small-step operational semantics for our computation model of prototypes with multimethods. The model differs from many others by not being an extension of the $\lambda$-calculus. Even at this formal level homogeneity is at its best: for instance, $\lambda$ forms are objects which can be manipulated freely (e.g. be cloned, or be enriched with additional delegations), and their arguments are accessed through regular message passing. To the extent of our knowledge, no other formal model achieves the same level of homogeneity for a similar set of language features. The semantics can help exploring future extensions thanks to its fine granularity, it is executable, and the reduction process can be traced using the graphic browsers furnished by PLT Redex [78].

The syntax accompanying our computation model, provided by Ambience, is consequent with our quest for homogeneity. It has no special cases —everything is a message. Further, it is a better match for multiply dispatched languages than previous syntaxes. We present what we believe is the cleanest Smalltalk-like syntax for an object-oriented language with multiple dispatch.

**Context-Oriented Programming**

Upon undertaking our research work, we took for granted that application behaviour should depend on its execution environment —our initial research question was rather how this should be done. This dissertation presents the first prototype-based computation model for COP we are aware of. We took the

---

1. Save for the recent related work mentioned in section 8.2.
2. This makes it possible to define unit tests such that changes that break intended semantics are detected when exploring model extensions.
3. There is no single reserved keyword, there is no reserved syntax for constructs such as method returns.
4. Namely, because it supports messages that start by a keyword rather than an argument —the first syntactical element of messages is not as central as it is in singly dispatched languages. This permits more natural method names and messages. Besides that, the specification of method specialisers as parenthetical expressions further contributes to the intuitive looks of Ambience.
8.1. Contributions

strand of subjective dispatch that was left unexplored by the designers of the PMD model [93] and extended it into a full approach to develop dynamically adaptable applications. The result is a highly dynamic computation model: it features dynamic dispatch,\(^5\) dynamic inheritance, dynamic typing, and dynamic method scoping. The specific contributions of this computation model for COP are as follows:

4.2.1 Context representation This dissertation is the first one to propose a context representation based on prototypes and dynamic, multiple inheritance. Such representation has a number of advantages:

- The representation of context is simple and concrete. This helps creating a sense of tangibility and malleability [102] of context. By exposing the representation to the programmer, it becomes possible to have a direct mental picture, and a clear programmatic understanding of what context is and how to manipulate it.

- The connection between context and behaviour is immediate, making it easy to understand how context affects behaviour. Causality between context and behaviour comes as a natural consequence of regarding the context as an object (graph).\(^6\)

- Idiosyncratic contexts are supported naturally. Our approach inherently (paradigmatically) supports behaviour that is adapted to very specific contexts, such as one particular room of a building. Thanks to delegation, those specific contexts can exhibit more general behaviour as well.

4.2.2 Dynamic behaviour adaptation Overall system behaviour can be adapted by manipulating the delegation topology of the context graph dynamically. We provide suitable mechanisms to reflect detected physical and logical environment changes on the computational representation of context on the fly. The following original techniques are described in the dissertation:

4.3.2 Context combinations Our approach allows programmers to define behaviour that depends on the combination of one or more contexts. Delegation relationships among different context combinations are maintained automatically so that behaviour specific to certain combinations is seen in more specific ones. The run-time system also takes care of preserving combination identity so that recurrent composition of the same contexts always yields the same combination object. Automatic delegation management and identity preservation are the two minimal features that any system supporting context combinations must provide. The combination mechanism is not only used when programmers are defining behaviour,

\(^5\)This synonym of multiple dispatch emphasises the fact that behaviour selection depends on the dynamic value of all arguments.

\(^6\)In a black-box view, the context is simply an object with behaviour, irrespective of whether this behaviour comes from delegation or not; in an open view of context, context structure is revealed and it becomes apparent that the context is actually a graph of delegating objects.
but also constantly at run time when contexts are switched on and off dynamically.

**Induced, nested and interleaved activation** The dissertation shows how to support different context activation interleavings that can occur due to environment changes, for example due to user mobility.

**Language abstractions** We introduce *bypass resends*, which allow forwarding a message to a less specific version of the currently executing method, version which will be chosen as if a number of given contexts were inactive. This construct helps avoiding mismatches between the logic encoded in methods and the phenomena that are actually occurring in the outside world. This construct is an example of the kind of language abstractions that help solving impedance mismatches between computational systems and their description as context-oriented programs, mentioned in the introduction.

**Experience with COP** We have unveiled some of the possibilities of COP while working with our approach:

**Thesis validation** The most basic contribution to COP is embodied by the thesis validation. We show the way our approach can be used to implement typical AmI scenarios in an intuitive, elegant way. The validation cases show that dynamic adaptation to context can be achieved with architectures that closely follow the application’s domain, rather than being biased by context adaptation concerns. Application logic code can be expressed cleanly, and be oblivious to extrinsic adaptation concerns. It can even be simpler by exploiting context-oriented programming for adaptation to intrinsic contexts that are inherent to the inner workings of the application. Finally, adaptation code can be expressed non-intrusively.

**Context meta-programming** We show how to intercept context changes simply by overloading suitable methods belonging to the Context Object Protocol of AmOS, and possible uses of this technique.

**Features** We show the way contexts can be exploited as a modularisation mechanism. Prototypes, methods and contexts can be packaged in a “feature”; a feature is a context that contains these definitions and can be switched on and off dynamically. Thanks to context, the composition of software effectively varies at run time.

**Transactions** We show that the model is expressive enough to support the elegant implementation of advanced language features. Specifically, we have found that subjectivity provides the right stratum for the adequate (paradigmatic) support of lightweight software transactions. This gives a first taste of the possibilities that could break open with further research on COP.
8.2. Related Work

Given that COP is an emerging field, there are only a few languages that seek to support dynamic adaptation to context by featuring dedicated language abstractions. To the best of our knowledge, there are only three: Us [101], ContextL [29] and Slate [93]. Besides these languages we discuss a few others that have been sources of inspiration. The presentation starts by discussing languages on the Smalltalk camp, followed by languages on the Lisp camp.

8.2.1. Us

Us [101] is a proof of concept implementation on top of Self [115], aimed at exploring subjective object behaviour. Since Us extends Self, it features prototypes, multiple delegation-based inheritance, and single dispatch. The very idea of supporting subjective behaviour as envisioned by Us has inspired our work since the early stages. In the model of Us, the state and behaviour of objects depend on a “perspective” from which the objects are seen; this perspective is reified as a layer object. Us extends Self’s message dispatch mechanism to account for this layer. In these regards, layers are akin to contexts in our approach. There are however fundamental differences in the way subjectivity is achieved, and the way it is exploited.

Firstly, layers have special status in Us. Even though Self supports multiple inheritance, Us supports only single inheritance in layers. The authors admit that “this is a simplification of sorts” and that “multiple layer parents are conceptually reasonable: a point of view can derive from several other viewpoints”. AmOS supports multiple inheritance of contexts and exploits this mechanism at its best—in particular for context combinations, a cornerstone of context-oriented programming in AmOS.

Secondly, in Us there are no layer combinations as there are context combinations in AmOS. To alleviate this problem, for each requested combination layers could be arranged in chains through the parent delegation link (effectively linearising “by hand” what would otherwise be a delegation graph, if multiple layer inheritance were supported). However, method definitions would still be specialised on the topmost layer, rather than being specialised on a distinct combination of layers. Such approach would thus be unsound, as methods would be visible when the topmost layer of their layer combination is active as part of some other (probably unrelated) layer combination chain.

Thirdly, Us subjective objects are composed of pieces. A piece is a possibly empty collection of attributes and methods of an object, and one such piece is associated to each layer. This means that an object’s representation is actually sparse, distributed among its pieces. The way Us handles layers and pieces raises problems that force a distinction between layer delegation and plain object delegation. The authors call the first “layer parent links” and the second “inheritance links”. They state that “reusing the inheritance link as the layer parent link would create problems. There is a difference between the attributes in the pieces a layer holds on behalf of other objects, and the attributes that
reflect the state of the layer itself.” This mismatch arising from intermingling layers and object pieces\(^7\) does not occur in AmOS, and hence, AmOS features only one kind of inheritance.

Fourthly, Us proposes a layer-switching construct denoted by the \(\otimes\) symbol which is akin to `with-context` in AmOS and `with-active-layers` in ContextL. The given layer is in force for all messages sent in the dynamic extent of the layer-switching construct, and the original layer is restored upon return from invocation of this construct. If the layer needs to be adjusted at some point within that dynamic extent, a \(\otimes\) layer switch must be foreseen and hard-coded in the application logic. While this is not necessarily a problem for intrinsic adaptations, it is impossible to foresee all extrinsic adaptations that could possibly be needed. Us does not propose a way to influence behaviour “from the outside” as AmOS does.\(^8\) Once a method has been invoked from a given perspective—and this can mean a whole application that is launched through that method—there is no way to influence its behaviour other than hard-coding of layer switches.

### 8.2.2. Cecil

Cecil [17, 18] has been a main source of inspiration.\(^9\) Cecil is a prototype-based language with multiple dispatch that consolidated the experience gathered by some of its predecessors (notably Self [115]). Cecil features a dynamically typed core, on top of which it lays optional static types. Static types are introduced to enable compile-time checks and optimisations.

Having a design that is aimed at enabling compiler optimisations, Cecil’s main limitation is lack of support for dynamic inheritance. This shortcoming is attenuated by support of predicate objects in Cecil.

Unlike Common Lisp Object System (CLOS) and AmOS, but like Multi-Java [25], Cecil features symmetric dispatch. Ambiguities are not automatically resolved by symmetric dispatch—in face of ambiguities, Cecil raises an exception. Given its relatively static nature, Cecil can detect ambiguities at compile-time that are impossible to detect in the more dynamic models. We regard symmetric dispatch as more elegant than asymmetric dispatch, but give priority to support of dynamic inheritance.

---

\(^7\)The authors state that “the lookup algorithm composes the layer hierarchy first, and then constructs the normal inheritance hierarchy”, but the exact “composition” and “construction” mechanisms of hierarchies are rather obscure in the paper.

\(^8\)The usefulness of extrinsic adaptations is illustrated by the case study in section 7.2.1 on page 142, in which the behaviour of a `play` method is adapted in the middle of execution according to remaining battery power; a second case is explained in section 4.3 on page 68 for adaptation of a running CityMaps application to a Global Positioning System (GPS) service.

\(^9\)Cecil is now known as Diesel.
8.2.3. PMD and Slate

Ambience and its underlying engine AmOS were initially inspired on Self [115] and Cecil [17], but later on adopted the similar, albeit more flexible, PMD computation model implemented by the Slate language [93]. Slate draws inspiration from Self, CLOS, and Smalltalk.

We see PMD as an improved version of Cecil. Coarsely, their two main features are the same, namely prototypes and multiple dispatch. The main contributions of PMD are support of dynamic inheritance, and the introduction of roles, a notion that helps establishing a natural relationship between objects and methods at the conceptual and technical levels. This brings derived advantages to the way method lookup can be defined and implemented.

Semantically, the cores of PMD and AmOS are not far apart. Among the important differences, AmOS uses a C3 linearisation, whereas PMD uses a depth-first linearisation. AmOS introduces new language constructs (mentioned in section 8.1) which PMD does not have. The most important difference however is the context-oriented machinery AmOS lays on top of its core, machinery that is not featured by PMD. This machinery is based on subjective dispatch, originally proposed for the Self extension Us [101]. Although the authors of PMD are well aware of the potential of subjective dispatch, it again faded into oblivion as happened with Us. We know of only one short example showing the potential of subjective dispatch in the PMD model [93]. AmOS on the contrary boosts subjective dispatch, making it as fundamental to the model as prototypes and multimethods.

8.2.4. CLOS

AmOS has drawn some inspiration from CLOS (directly, but also indirectly through the PMD model). To start with, AmOS is also implemented as an object-oriented layer on top of the procedural part of Common Lisp. Like CLOS, we decided to call this layer an “object system”. The first obvious difference is that AmOS is prototype-based, whereas CLOS is class-based.

In CLOS, methods can be specialised either on classes or on particular objects; in the latter case EQL specialisers are used. The availability of EQL specialisers in CLOS shows that in practise idiosyncratic behaviour is usually needed. Thanks to its prototype-based core, AmOS needs only one form of method specialisation, which subsumes the two forms found in CLOS.

In CLOS methods belong to generic functions, whereas AmOS does not have such concept of generic function. In AmOS, method ownership is shared among specialisers: a method belongs to each and every one of its specialisers, both conceptually and technically. This is a natural extension of ownership from traditional class-based (mostly singly dispatched) languages. What makes us eschew generic functions is that they are global resources: objects that can be grabbed from the environment just by naming them. In AmOS we seek to avoid constructs that could provide ambient authority [81].

In some regards, CLOS is more advanced than AmOS. Firstly, there is no
AmOS equivalent when it comes to method combinators, particularly auxiliary :around, :before and :after methods. These auxiliary methods are very handy in practise. Secondly, CLOS supports optional arguments and keyword arguments [98]. Although it would not be a fundamental improvement, extending AmOS with such constructs would render it more practical.

8.2.5. ContextL

Soon after adopting the PMD model we became aware of ContextL\textsuperscript{11} [29], a class-based cousin of AmOS, which also exploits a sort of dynamic scoping mechanism to achieve behaviour adaptation. ContextL—a CLOS extension— not only shares the similar goal of having behaviour depend on context, but also a similar approach, by using an implicit argument that influences method dispatch. There are, however, some important differences. In ContextL there is one layer configuration (analogous to the context graph of AmOS) per thread. Threads cannot modify each other’s layer configurations. Whereas thread locality ensures non-interference with other threads, such interference is sometimes useful. In AmOS, there is a unique context graph that is shared by all threads. AmOS implements immediate reaction to changes in context, whereas ContextL sticks to an initial context while finishing an ongoing computation. Desmet et al. [38] call these promptness strategy and loyalty strategy respectively, giving examples of the usefulness of both. Both approaches have their advantages and disadvantages. In AmOS the concurrent modification of the shared context graph can give rise to inconsistent behaviour [59]. In ContextL, the layer configuration must be adapted locally in the current thread, implying that context-switching constructs such as with-active-layers and ensure-active-layer must be scattered throughout application code if such configuration is to be adapted in some way (since layers cannot be adjusted from outside the thread). As suggested in section 8.3, a conciliation of both approaches is part of our future work.

Other fundamental differences between ContextL and AmOS regard the contributions stated in section 8.1. Firstly, there is a difference in the conception of context. ContextL does not regard the current layer configuration of a thread as a context per se. Furthermore, the structure of such layer configuration is not exposed to the user—rather, a well-defined protocol is provided to work with layers. In AmOS, the context is open: its structure is exposed as a graph of delegating objects and the user is allowed to manipulate such structure (e.g. by adding delegations or switching delegations). Secondly, to our understanding ContextL does not deal with different layer activation inter-leavings, because layer activation follows a stack discipline (dynamic extent of the with-active-layers call).

\textsuperscript{10}When more than one method is applicable to a given set of arguments, the applicable methods are combined into a single effective method. Each individual method definition is then only part of the definition of the effective method.

\textsuperscript{11}And its related variants ContextS and ContextJ.
Further differences between AmOS and ContextL are actually due to the use of CLOS in the latter (notably the use of classes and generic functions). These differences are discussed in section 8.2.4.

8.2.6. Contextual Values

The context-specific accessors of AmOS resemble the *layered accessors* of ContextL [29]. In both cases, observed object attributes can be different when consulted from different perspectives (contexts). A similar effect can be obtained through a fundamentally different mechanism, *contextual values* [47]. In object-oriented terms, a contextual value can be seen as a context-dependent reference: the object to which such reference points can depend on the context in which the reference is used. The difference in expressiveness between our approach and contextual values is not clear to us yet — that of having a *same* object that behaves differently according to context, and that of having *different* objects according to context. At first sight, the approaches seem complementary.

8.3. Limitations and Future Work

Despite the contributions mentioned in section 8.1, many questions remain unanswered. The exploration of the different questions constitute paths for future research. We present them starting from the more specific, and move on to the more general ones.

5.2.6 **Context-specific delegations** Thanks to accessor methods, our approach supports context-specific slots, meaning that the observed value of a slot can vary according to the currently active contexts (i.e. the current “point of view” of the observer). However, delegations are unaffected by context. An object always delegates to the same delegate list, irrespective of the contexts that are active upon message dispatch. It is conceivable to support context-specific delegations. This exploration could unveil new ways of programming context-sensitive behaviour. In particular it would bring subjective objects [101] closer to subject-oriented programming [64].

**Activation contexts** In this dissertation we propose a system-wide context which all threads see. This context is the parent of the top-level activation. However, there are advantages to having local contexts as in ContextL [29]. In our view, support for both is needed. A primary line of future research

---

12 The analogy we draw goes for *implicit*, rather than explicit, contextual values [47]. The original presentation of contextual values in Lisp is in terms of variables and values, instead of references and objects.

13 Subject-oriented programming proposes support for classification of objects in different class hierarchies, depending on the subject or “point of view” from which those objects are seen.
Figure 8.1.: Idea for supporting activation-local contexts.

is the support of global and local contexts. This support is not necessarily complicated, but must still be assessed. At first sight, both types of context, local and global, can be subsumed by one construct, an hypothetical `invoke-with-local-context` method that would take a closure and reparent it so that it has local context, as depicted in figure 8.1. The local context would be a combination of all locally active contexts (in the figure, it is shown as one single box with a + sign). The newly created local context would delegate to the original `parent` activation of the given closure, so that the closure’s lexical scope is visible as is normally expected. As shown in the figure, the global context would simply be the local context of the top-level activation —hence our assertion that both global and local contexts can be subsumed by a notion of activation contexts. In this setting, local contexts override more global ones, which seems intuitive. The proposal of activation contexts just described is a natural generalisation of the approach presented in this dissertation.

**Exception handling** AmOS does not propose an explicit exception handling mechanism. Currently, we use the condition system from the underlying Common Lisp implementation. The condition system of Common Lisp not only supports Java-like exceptions, but also can be used to handle special events during the execution of a program (such as warnings) without unwinding the execution stack. It would not be difficult to support a similar condition system in AmOS. Recall from section 5.2.2 that activations are plain objects that can exhibit behaviour of their own. Method activations, in particular, understand special messages which are targeted at them, such as `resend`. Analogously, conditions could be raised by means of `handle` messages that would be understood by special `condition handler` activations. Such `handle` messages would permeate all activations for which the message is unknown (e.g. plain closure and method activations); the message would ultimately be delegated to the most recent condition handler that is currently linked to the chain of activations. This handler would understand the message and handle the condition.

---

14Exception systems usually provide a two-part division between the code that signals an error and the code that handles it. Common Lisp’s condition system splits the responsibilities into three parts —`signalling`, `handling`, and `restarting`, allowing for more flexible uses [98].
8.3. Limitations and Future Work

**Context-based privilege separation** One more interesting area of research (and again, relatively immediate given the semantic exploration laboratory we already have, AmOS) is to exploit contexts as a tool for modularisation and privilege separation. The problem and solution are perspicuously explained by Smith and Ungar [101]:

In the last few years, many researchers have proposed that the operations on objects as objects *per se* be formally separated from the operations on an object that it provides to stand for something in the problem domain. The domain of objects as objects is called the reflective domain. [...] However, we have found the meta-object approach can at times be awkward, because confusion can arise as to whether to pass in an object or its meta-object as an argument to certain methods. In such situations, sometimes we Self programmers yearn for the “one reference does it all” days of Smalltalk programming. Worse yet, there is more than one kind of reflection; for example structural reflection provides access to structure, and behavioural reflection, which allows for behaviour modification. Modularisation would suggest that each object would need a different associated meta-object for each kind of reflection, and this multiplicity would further exacerbate the problem. Subjectivity offers a way out. Instead of dealing with a cluster of different objects, we can provide different perspectives, one for non-reflective, ordinary computation, and one for each kind of reflection. Since such a system would always pass around pointers to the same kind of object, the reuse barriers erected by meta-objects would disappear.

Generalising from this case about computational reflection [100, 75], contexts could be used as a way of conferring privileges and withdrawing them for many domains (not only the “meta” domain), provided that functionality is carefully modularised among different contexts. Retaking the case shown previously, @introspection and @intercession contexts can contain separate functionality so that applications would be able to perform introspection only when they are in @introspection context (i.e. when such context is active), and similarly for @intercession. Those contexts can be subdivided further in many subcontexts with finer granularity as suggested in the previous excerpt. In this dissertation we do show the way applications can be modularised by means of contexts. However, support of local contexts (described previously) is a must to properly support context-based encapsulation, given that, in the general case, context changes related to privilege separation should not be applied globally.

Context-based encapsulation could be used to support the Principle of Least Authority (POLA) [80]. In default context, available functionality would be the bare minimum applications need to authenticate and gain increased privileges for normal operation. Privileges would be escalated by executing given closures in contexts that give access to more functionality, thanks to
the would-be `invoke-with-local-context` construct described previously. In their local context, applications would have access to the right (minimal) amount of functionality they need, or can possibly acquire. The chief advantage that might make context-based encapsulation superior to plain object-based capabilities [80] is that objects would not need to be split in many parts due to privilege separation concerns, as suggested by Smith and Ungar [101] in their example for reflection.

Finally, the relationship between context-based encapsulation as described previously and encapsulation policies [97, 96] would be worth investigating (assessment of their differences, complementarity, or whether one subsumes the other).

**Context lifetime** Consider this excerpt about mirror objects from a paper by Bracha and Ungar [16], which motivates the need for context lifetime management:  

> When deploying an application, it is not always desirable to deploy it together with all the reflective facilities available in the language. The application may not require these reflective capabilities at all, or it may require them infrequently. In such cases, it may be advantageous to reduce the application footprint by avoiding or delaying the deployment of reflective facilities. This is especially true on small platforms such as mobile phones, PDAs, smart cards or other embedded systems.

In AmOS, it is possible to drop contexts which are no longer needed. All methods specialised on a dropped context are removed from the system, because they cannot possibly be applicable anymore. Likewise, objects such as context-specific prototypes which are no longer referenced will be garbage collected. Resource management in our approach is not limited to reflection, but in fact any unused context can be dropped. An important open question regards context lifetime: when should stored contexts be removed from the system to save resources? For instance, a `@sears-tower` context that is specific to a building could be used every day (e.g. if the user works in that building, the context would usually be activated every morning and deactivated every afternoon); however, the context could be used only one time and then remain inactive in the system forever (if the user visited the building once for business), wasting resources in the latter case. This example shows that the decision on context removal cannot be automatised in the general case. However, we believe that suitable techniques can be devised for context lifetime management (e.g. to have context expiration or leasing policies). This constitutes a future line of research.

**Concurrent context manipulation** Having a system-wide context, which many threads can manipulate concurrently, raises problems regarding behaviour coherence. In this dissertation we have shown an initial approach to man-
aging concurrent context access, but it is not yet satisfactory. A future line of research consists in exploring mechanisms to disallow deactivation of contexts that are currently being used.

**Concurrency support** Our model does not have dedicated concurrency abstractions. An important line of research is to incorporate asynchronous message passing and futures to the model. This line of research requires careful examination. The following are the most important issues and our devised approach to tackle them:

*Uniform active objects* Guided by a uniformity principle, we believe that all objects in the system should be active, conceptually. In Actor parlance [3], this means that all objects should be actors, without distinction. Technically, objects would become active when they receive an asynchronous message, and become passive again when they do not have pending messages in their queue. There might be a cost associated to the promotion of objects from passive to active status, and a corresponding cost of dismantling infrastructure when objects become passive again. The infrastructure could be maintained if the object is expected to be active again soon. Validation of this idea constitutes a first research line.

3.1 *Message scheduling* Given that in our model with symmetric multimethods messages do not have a distinguished receiver as in singly dispatched languages, there is no a priori choice on where to queue received asynchronous messages. Our initial answer is that having the opportunity to choose which argument will process a message is far better than not having such choice at all. Hence, the open question actually unveils an opportunity. We believe that in principle the message could be queued in any of the message argument’s queues. Choosing always the first argument would simulate traditional actor concurrency —multimethod actor concurrency can thus be seen as a generalisation. Another option is to choose an argument that already is an active object, so that the cost of promoting a passive object into an active one is avoided. In face of distribution, the choice becomes more important. For example, the system can choose a message argument that resides on the local machine to decrease communication costs, or an argument that is located on a peer with more processing power, or an argument for which the network link is the fastest. Multimethod actor concurrency seems to open many interesting research opportunities —if the idea is not flawed in the first place. Validation of the idea is of course a first required step.

*Progressive dispatch* If a message contains a future as argument, it is possible that the message cannot be fully dispatched, because resolution of such future argument might affect the final choice of a most-specific method for the message. Our answer (and future research line) is that messages can be dispatched progressively, as future arguments are resolved. In some cases, the system could determine that only one method is applicable to the given

---

16 The same principle that also led us to think that multimethods are a better design choice than singly dispatched methods.
selector and already resolved arguments, and thus could invoke that known method straightaway, passing the partially resolved argument list (where some arguments would be resolved and some would still be futures). This shortcut situation might be more common than one would expect, since methods are often specialised on just a few arguments [93] (the other arguments being specialised on the prototypical @object). In the worst case, if there is more than one contending method, dispatch will be completed when there is a resolution for all future arguments on which possibly applicable methods are specialised.

These ideas are not really specific to context-oriented programming, they could be applied to any multiply dispatched language.

**Distribution support** As concurrency, distribution requires careful consideration. Message arguments need to be examined during dispatch (to find their roles and be able to rank possibly applicable methods). If some arguments are references to remote objects, this means that costly communication would be incurred during dispatch. In singly dispatched languages, only the receiver needs to be examined; if it is local, then no communication cost is incurred (even if all other arguments are remote references). However, for most messages there is a limited number of arguments that need to be consulted—the arguments that are actually polymorphic for the message. This is mentioned in the previous discussion of concurrency and futures: only discriminating arguments (i.e. those whose value can possibly matter for the choice of an applicable method) need to be examined. If all polymorphic arguments are local, no communication cost is incurred, even if all other (non-polymorphic) arguments are remote. This is analogous to the case of singly dispatched languages when the receiver is local. On the other hand, all polymorphic remote arguments of a message do need consultation, with its associated communication cost. We believe that if such messages with distributed polymorphic arguments are sent, is because the distributed choice of behaviour is needed anyway. Furthermore, this mechanism could unveil opportunities for enhanced behaviour adaptation and cooperation, given that exhibited behaviour comes from consultation of objects in different peers. With appropriate caching schemes, it could be possible to reduce communication cost when the same remote objects are used regularly (e.g. by exchanging just one message to see if the delegation graph of the remote object has changed in some way, and use cached information if it has not). For remote objects that are known to be constant (e.g. coming from a standard library), the cache cannot possibly become invalid, and hence no communication is needed at all even if those objects are used in a polymorphic argument position. All these ideas require further examination, and chiefly, validation through implementation.

---

17Our implementation of AMOS already includes a similar shortcut in its dispatch mechanism; method lookup stops as soon as it is found that only one method is possibly applicable; this technique has been adapted from the Slate programming language [93].
### 8.3. Limitations and Future Work

**Shared contexts** The approach we have described assumes a local view on context. Each device is responsible for sensing properties of its surrounding environment, inferring information and managing its local representation of context accordingly. Supporting a local context representation is a minimum requirement for any context-aware system. As shown in this dissertation, this local view on context opens up many possibilities for ambient applications, because it establishes a relationship between a device and its environment, allowing for suitable behaviour adaptations. Nonetheless, more sophisticated forms of adaptation could be possible if we depart from this local view. Support for distribution would open up the possibility of having a group of devices agree on building, activating and deactivating *shared contexts* which they could use to carry out particular collaborative tasks. As an example, a video projector, sound equipment and a palmtop storing a multimedia presentation could agree on running in “presentation context”. Supporting fault-tolerant, peer-to-peer agreement on context is not a trivial problem, but luckily, known techniques exist. A possible line of research is thus to apply reliable distributed programming abstractions [61] to manage shared contexts. We conjecture that certain kinds of advanced collaboration and emergent behaviour can only be achieved when devices are able to align on certain objectives which require shared views on context. However, even if shared contexts are supported, the shared part of context should never stop a device from working properly (e.g. due to network disconnections). To preserve autonomy, the local part of context should always be considered the only dependable one.

**Relationship to software features** Our experience with COP suggests a strong relationship between contexts and software features, and recent initial work on Context-Oriented Domain Analysis (CODA) [37] confirms this impression. We have used the word “feature” liberally in this dissertation when discussing certain contexts that seem to have the status of software features, for instance the `@telephony-context` comes with a number of prototypes and methods that enable support for telephony. However, other contexts are clearly not “features”, for instance `@off-hook-context` just signals that a phone is in use. We need to explore the relationship between contexts and software features more thoroughly, and under the light of the work by Desmet et al. [37]. Quite importantly, advanced methodologies are needed to help the programmer design applications that can maintain coherency in face of run-time switching of contexts. CODA already takes into consideration different policies for maintaining coherence upon switching contexts (e.g. *prompt* context changes versus *loyal* contexts that stick until they can be deactivated). CODA could be extended to help the programmer foresee and handle graceful degradation, for instance.

---

18. Real-time constraints can add up to the challenge, as suggested by the example related to multimedia.
Development tools  Our approach constitutes a paradigmatic shift in the way applications are conceived—namely, with context in mind—and this shift should be supported by tools. To aid context-oriented application development, a number of interesting tools can be envisioned.

*Context browser* – To increase the feeling of tangibility and concreteness of contexts in our approach, it would be interesting to have a graphic *context browser* that permits real-time visualisation and direct manipulation of the context representation, much like the Self programming environment permits a “direct experience” for manipulation of objects [102]. This browser could aid debugging by showing graphically the choice of methods according to context.

*Ambient simulator* – Another important kind of tool would be an *ambient simulator* in which programmers can test various AmI scenarios for the applications they are developing, allowing them to add emulated devices, and simulate different kinds of actions and environmental properties, such as entering determined spaces, passing from sunlight to lamplight, losing network connections, running with low battery, etc.

Design and programming methodologies  In this dissertation we suggest initial guidelines to program in COP. However, a more systematic investigation is needed. Hence, an important path for future research is the development of full-fledged context-oriented design and programming methodologies. A thorough exploration of advanced COP patterns beyond the ones we have suggested would represent a significant contribution to the field. The relationship between contexts and software features suggested previously is another possibility. Such advances in context-oriented software engineering should be accompanied, as suggested previously, by adequate tool support.

This list of possibilities could go beyond. The future of the cross-fertilisation between subjective objects, PMD and COP could hardly be more promising.
A Semantics Companion

This appendix complements the formal semantics described in chapter 6 with a formal definition of the C3* linearisation (section 6.5), a brief introduction to context-sensitive term rewriting systems (section A.1), and a few semantic definitions that have been omitted from that chapter (section A.2).

A.1. Term Rewriting Systems in a Nutshell

The expression language Λ of the λ-calculus and the λv-calculus is the union of a set of values and expression juxtapositions. The set of values is the collection of basic constants (b ∈ B) and functional constants (f ∈ F), variables (x ∈ X) and λ-abstractions [50]. This is shown in figure A.1(a). Constants correspond to built-in algebraic language primitives like numbers, booleans and mathematical functions defined on them; variables are placeholders for values; and λ-abstractions are call-by-value procedures. Expression juxtaposition denotes function application.

The λv-calculus is based on a set of term relations (rewrite rules) on Λ. The two basic relations or notions of reduction are shown in figure A.1(b). The βv relation defines the semantics of function application. The expression e[v/x] is the result of substituting value v for variable x in expression e. The δ relation defines, roughly speaking, application of functional constants f to basic values v —that is, application of primitives.\(^1\)

\(^1\)For a rigorous definition of δ, refer to Plotkin [86] or Felleisen and Hieb [50].
Plotkin’s presentation of λ-calculi relies on a set of inference rules that specify the strategy for applying $\beta_v$ and $\delta$ in a leftmost-outermost manner [86]. These are shown in figure A.1(c).

Felleisen and Hieb’s alternate presentation of λ-calculi also uses $\beta_v$ and $\delta$ as primitive rewriting rules, but it relies on a set of term contexts instead of inference rules [50]. A term context is an expression with a “hole” $\Box$ at the place of a subexpression, or more formally, a term with a distinguished free variable, which can be substituted as needed [122]. For example:

$$C ::= \Box | (e\,C) | (C\,e) | (\lambda(x)\,C)$$

The expression $C[e]$ stands for the result of plugging the expression $e$ in the hole of context $C$ — that is, placing a term in the hole is equivalent to the substitution of the distinguished free variable by the term. Figure A.1(d) shows how to specify Plotkin’s evaluator function with a special kind of term context, the evaluation context $E$. The hole of $E$ is in such a position that a $\delta$- or $\beta_v$-redex\(^2\) inserted in the hole is the leftmost-outermost redex that is not inside of a λ-abstraction.

 Whereas the two specifications of a call-by-value evaluator shown in figure A.1 are similar at first glance, Felleisen and Hieb’s is more suitable for extensions with non-functional constructs such as assignment, exceptions, control, threads, and so on [78].

### A.2. Complementary Reduction Rules

For the sake of completeness, we show in figures A.2, A.3, A.4 and A.5, the definitions of reduction rules that have been omitted from section 6.4.

---

\(^2\)Redex stands for reducible expression. It is a reducible subterm that is to be reduced by a term rewriting system, given some reduction strategy.
invalid slot insertion
\((B[ (l_t \cdot S[ (l_1 \cdots) ] ) ] T[ \text{insert-slots } l_t n_i l_i \cdots ]) \rightarrow \)
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{error invalid-index} ]) \)
unless 0 ≤ n_i ≤ |l_1 \cdots |

invalid slot access
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{read-slot } l_t n_i ]) \rightarrow \)
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{error invalid-index} ]) \)
unless 0 ≤ n_i < |l_1 \cdots |

invalid slot modification
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{write-slots } l_t n_i l_j \cdots ]) \rightarrow \)
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{error invalid-index} ]) \)
unless 0 ≤ n_i ≤ n_i + l_j \cdots ≤ |l_1 \cdots |

invalid slot removal
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{remove-slots } l_t n_i n_c ]) \rightarrow \)
\((B[ (l_t \cdot S[ (l_1 \cdots) ]) ] T[ \text{error invalid-index} ]) \)
unless 0 ≤ n_i ≤ n_i + n_c ≤ |l_1 \cdots |

Figure A.2.: Slot manipulation error reductions.

---

delegation prepending
\((B[ (l_t \cdot D_1[ (l_1 \cdots) ]) ] T[ \text{prepend-delegation } l_t l_i ]) \rightarrow \)
\((B[ (l_t \cdot D_1[ (l_i l_1 \cdots) ]) ] T[l_i] )\)
unless l_i ∈ l_1 \cdots 

delegation reordering
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_i l_{i+1} \cdots) ]) ] T[ \text{prepend-delegation } l_t l_i ]) \rightarrow \)
\((B[ (l_t \cdot D_1[ (l_i l_1 \cdots l_{i+1} \cdots) ]) ] T[l_i] )\)
delegation access
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_i l_{i+1} \cdots) ]) ] T[ \text{read-delegation } l_t n_i ]) \rightarrow \)
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_i l_{i+1} \cdots) ]) ] T[l_i] )\)
provided that |l_1 \cdots | = n_i 
delegation modification
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_i \cdots l_k \cdots) ]) ] T[ \text{write-delegations } l_t n_i l_j \cdots ]) \rightarrow \)
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_j \cdots l_k \cdots) ]) ] T[l_i] )\)
provided that |l_1 \cdots | = n_i ∧ |l_i \cdots | = |l_j \cdots |
delegation removal
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_i \cdots l_{i+c} \cdots) ]) ] T[ \text{remove-delegations } l_t n_i n_c ]) \rightarrow \)
\((B[ (l_t \cdot D_1[ (l_1 \cdots l_{i+c} \cdots) ]) ] T[l_i] )\)
provided that |l_1 \cdots | = n_i ∧ |l_i \cdots | = n_c

Figure A.3.: Delegation manipulation reductions.
invalid delegation access
\[(B \[ l_t \ D[ \[ l_1 \cdots \] ] \]) T[\text{(read-delegation} \ l_t \ n_i \text{)}] \rightarrow\]
\[(B \[ l_t \ D[ \[ l_1 \cdots \] ] \]) T[\text{(error} \ \text{invalid-index})] \]
unless \[ 0 \leq n_i < |l_1 \cdots | \]

invalid delegation modification
\[(B \[ l_t \ D[ \[ l_1 \cdots \] ] \]) T[\text{(write-delegations} \ l_t \ n_i \ l_j \cdots \text{)}] \rightarrow\]
\[(B \[ l_t \ D[ \[ l_1 \cdots \] ] \]) T[\text{(error} \ \text{invalid-index})] \]
unless \[ 0 \leq n_i \leq n_i + |l_j \cdots | \leq |l_1 \cdots | \]

invalid delegation removal
\[(B \[ l_t \ D[ \[ l_1 \cdots \] ] \]) T[\text{(remove-delegations} \ l_t \ n_i \ n_c \text{)}] \rightarrow\]
\[(B \[ l_t \ D[ \[ l_1 \cdots \] ] \]) T[\text{(error} \ \text{invalid-index})] \]
unless \[ 0 \leq n_i \leq n_i + n_c \leq |l_1 \cdots | \]

Figure A.4.: Delegation manipulation error reductions.

---

define delegation accessors
\[PT[\text{(define-delegation-accessors} \ l_t \ l_n \ n_i)] \rightarrow\]
\[PT[\text{(seq} \text{install-method} \text{(delegation-reader-method} n_i \text{)} \ l_n \ l_t)\]

\[\text{install-method} \text{(delegation-writer-method} n_i \text{)} W[l_n \ l_t \ \text{object}) l_t)]\]

delegation reader method
\[PT[\text{(delegation-reader-method} n_i)] \rightarrow\]
\[PT[\text{(eta} \ \text{anonymous}) \text{(read-delegation} \text{invoker) n_i)}]\]

delegation writer method
\[PT[\text{(delegation-writer-method} n_i)] \rightarrow PT[\text{(eta} \ \text{anonymous anonymous) (write-delegations} \text{invoker) n_i (read-slot (activation) 1))}]\]

Figure A.5.: Delegation accessor reductions.
Formal Syntax of Ambience

An Ambience expression is a value, a message or a method definition:

expression ::= value | message | method

The forthcoming sections explain the syntactic form of values (Section B.2), messages (Section B.3) and methods (Section B.4). Prior to the grammar, an explanation of the terminals is given next.

B.1. Scanner

The definitions of alpha-char and digit-char have been omitted expressly.

B.2. Values
B.3. Messages

message ::= nullary-message
    | unary-message
    | semi-binary-message
    | binary-message
    | keyword-message

nullary-message ::= basic-selector

unary-message ::= unary-argument basic-selector

unary-argument ::= value | unary-message

binary-message ::= binary-argument semi-binary-message

semi-binary-message ::= binary-selector unary-argument

binary-argument ::= unary-argument
    | binary-argument binary-selector unary-argument

keyword-message ::= keyword-argument message-remainder
    | message-remainder

message-remainder ::= keyword-selector keyword-argument
    | keyword-selector keyword-argument message-remainder

keyword-argument ::= binary-argument

B.4. Methods

method ::= nullary-method
    | unary-method
    | binary-method
    | keyword-method

nullary-method ::= basic-selector method-block

unary-method ::= method-parameter nullary-method

semi-binary-method ::= binary-selector method-parameter method-block

binary-method ::= method-parameter semi-binary-method

keyword-method ::= keyword-parameters method-block

keyword-parameters ::= method-parameter method-remainder
    | method-remainder

method-remainder ::= keyword method-parameter method-remainder
    | keyword method-parameter

method-parameter ::= unary-selector ( expression )
    | unary-selector
    | ( expression )

method-block ::= block
B.5. Auxiliary productions

keyword-selector ::= basic-selector :
symbol ::= # symbol-name
symbol-name ::= basic-selector | binary-selector | keyword-selector | string
No Rest for the Wicked

The following progress graph shows an approximate word count per hour on the vertical axis, as time goes by on the horizontal axis, for this dissertation:

Not only words, but also \LaTeX{} commands found in the .tex source files, and even discarded text which does not appear in this document, are counted.

Script The following Bash script shows the essential instructions that generate the graph:\footnote{With due thanks to my friend Alejandro Forero who suggested the idea of counting words, and provided the initial version of the script. It can be executed @hourly in a crontab(5).}

```
echo 'date "+%s" ; cat *tex | wc -w' >> progress.txt
{ cat <<EOF
    set terminal pdf
    set output "progress.pdf"
    set xdata time
    set format x "%d/%m"
    set timefmt "%s"
    plot "progress.txt" using 1:2 with lines
EOF
} | gnuplot 2>/dev/null
```
Bibliography

The references are sorted alphabetically by first author.


[40] Simon Dobson and Paddy Nixon. More principled design of pervasive computing systems. In Engineering for Human-Computer Interaction and Design, volume 3425 of Lecture Notes in Computer Science,
[41] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006. ISSN 0164-0925. DOI 10.1145/1119479.1119483.


Acronyms

AmI  Ambient Intelligence
AmOS Ambient Object System
http://ambience.info.ucl.ac.be
AOP  Aspect-Oriented Programming
API  Application Programming Interface
CLOS Common Lisp Object System
CODA Context-Oriented Domain Analysis
COP  Context-Oriented Programming
http://www.swa.hpi.uni-potsdam.de/cop
CPU  Central Processing Unit
DNS-SD DNS Service Discovery
http://www.dns-sd.org
DNS  Domain Name System
DSL  Domain-Specific Language
EBNF Extended Backus-Naur Form
EPG  Extended Precedence Graph
FIFO First In, First Out
GLR  Generalised LR
GNU GNU is Not Unix
http://www.gnu.org
GPS  Global Positioning System
GUI  Graphical User Interface
IDE Integrated Development Environment
IP   Internet Protocol
JVM Java Virtual Machine
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVMDI</td>
<td>JVM Debug Interface</td>
</tr>
<tr>
<td>JVMTI</td>
<td>JVM Tool Interface</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile Ad-hoc Network</td>
</tr>
<tr>
<td>MOP</td>
<td>Meta Object Protocol</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PMD</td>
<td>Prototypes with Multiple Dispatch</td>
</tr>
<tr>
<td>PNG</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>REPL</td>
<td>Read-Eval-Print Loop</td>
</tr>
<tr>
<td>RFC2462</td>
<td>IPv6 Stateless Address Autoconfiguration</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identification</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>SOP</td>
<td>Subject-Oriented Programming</td>
</tr>
<tr>
<td>STM</td>
<td>Software Transactional Memory</td>
</tr>
<tr>
<td>POLA</td>
<td>Principle of Least Authority</td>
</tr>
<tr>
<td>mDNS</td>
<td>Multicast DNS</td>
</tr>
</tbody>
</table>

[http://www.multicastdns.org](http://www.multicastdns.org)