Vrije Universiteit Brussel

Faculteit van de Wetenschappen
Vakgroep Computerwetenschappen
Laboratorium voor Programmeerkunde

Modularising Context Dependency and Group Behaviour in Ambient-oriented Programming

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

Jorge Vallejos

Promotoren: Prof. Dr. Theo D'Hondt, Prof. Dr. Wolfgang De Meuter,
Dr. Pascal Costanza

Jul 2011

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (INNOVIRIS), and by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State, Belgian Science Policy.

“Wie niet waagt, blijft maagd.”

– Popular Belgian saying.

Samenvatting

In de visie van *pervasive computing* zijn computers ingebed in alledaagse apparaten. Zulke apparaten vormen onderling dynamische communicatienetwerken waarvan de topologie verandert naarmate de gebruikers zich verplaatsen. Van software services wordt verwacht dat ze maximaal gebruik maken van de beschikbare apparaten en dat de vereiste coördinatie transparant verloopt. Deze services moeten ook kunnen reageren op dynamische veranderingen in de omgeving zodat ze hun gedrag automatisch kunnen aanpassen. Een populaire aanpak om aan deze vereisten te voldoen bestaat erin om veranderingen in de omgeving voor te stellen als *events*, zodat pervasive computing applicaties gemodelleerd worden als *event-gedreven* systemen. Deze strategie heeft echter wel een impact op de onderhoudbaarheid van pervasive computing programma's. Hoe meer verschillende events een applicatie moet ondersteunen, hoe complexer het wordt om de *control flow* van het programma te beheren. Door een gebrek aan geschikte taalconstructies om events te manipuleren, is het in het algemeen moeilijk om zulke programma's te onderhouden en uit te breiden.

In deze verhandeling bestuderen we de impact van event-gedreven distributie op de modulaire structuur van programma's. We bestuderen twee bekommernissen in detail: Enerzijds bestuderen we de mogelijkheid van programma's om zich aan te passen aan veranderingen in de omgeving—een eigenschap die bekend staat als *contextafhankelijk* gedrag—en anderzijds bestuderen we de capaciteit van programma's om met elkaar te coördineren—een eigenschap die bekend staat als *groepsafhankelijk* gedrag. We baseren ons onderzoek op het *ambient-georiënteerde* programmeerparadigma (AmOP). Dit paradigma definieert een event-gedreven programmeermodel dat specifiek ontworpen is voor het pervasive computing domein. We identificeren een aantal vereisten voor het unificeren van event-gestuurde distributie, contextafhankelijk gedrag, en groepsafhankelijk gedrag. In ons overzicht van de literatuur tonen we aan dat geen van de bestaande programmeermodellen aan al deze vereisten voldoet. Dit leidde ons ertoe een experimentele programmeertaal te ontwikkelen voor pervasive computing, genaamd *Lambic*.

Lambic is een uitbreiding van het op generische functies gebaseerde objectmodel van de programmeertaal *Common Lisp*. Ons model breidt de *multiple dispatch* semantiek van generische functies uit om een modulaire programmastructuur mogelijk te maken voor pervasive computing applicaties. Ter ondersteuning van event-gestuurde distributie, vervult Lambic de eigenschappen van het AmOP paradigma via een mechanisme dat we *futurised generic functions* noemen. Daarbovenop bieden we een nieuw mechanisme aan om de flexibele selectie en samenstelling van gedrag toe te laten. Dit mechanisme noemen we *predicated generic functions*. Het laat ons toe om taalconstructies te definiëren zodat *method dispatch* de dynamische context van het programma in acht neemt. Ter ondersteuning van groepsafhankelijk gedrag, stellen we een derde uitbreiding voor, met name *group generic functions*. Het idee achter dit mechanisme is om de coördinatie van interacties tussen groepen in te kapselen in de definitie van services. Tot slot hebben we een gemeenschappelijk uitvoermodel gedefinieerd zodat de drie mechanismen effectief in combinatie met elkaar gebruikt kunnen worden. We valideren ons onderzoek door het bespreken van een aantal case studies die aantonen hoe Lambic het mogelijk maakt om vlot integreerbare en dynamisch aanpasbare pervasive computing services te ontwikkelen.

Abstract

The pervasive computing field envisions users surrounded by computers embedded in everyday devices. Such devices form dynamic networks which change topology as users move about. Software services are expected to maximise available computational capacities by seamlessly coordinating with each other. The services should also react to dynamic changes in the environment and adapt their behaviour accordingly. An increasingly popular solution to cope with such requirements is to represent environmental changes as events. Thus, pervasive computing software is often modelled as event-driven systems. However, such support typically comes at a price in the evolvability of programs. The more events a service has to be aware of, the more cumbersome its control flow becomes. The lack of adequate language abstractions to handle such events results in programs that are difficult to maintain and extend.

In this work, we study the effects of event-driven distribution on the modularity of programs. We focus on the modularity of two concerns: the capacity of services to adapt to environmental changes —a property known as *context-dependent behaviour*— and their capacity to coordinate with each other —a property known as *group behaviour.* We use the ambient-oriented programming (AmOP) paradigm as the basis of our research. This paradigm proposes an event-driven programming model which has been designed specifically for pervasive computing. We identify a list of requirements for a unified model for event-driven distribution, context dependency and group behaviour. In the study of the state of the art we demonstrate that no single approach fulfils these requirements so far. This observation has led us to the definition of a proof-of-concept programming language model, called *Lambic.*

Lambic is an extension of the generic function-based object system of the Common Lisp programming language. Our model extends the multiple dispatch semantics of generic functions to allow for modularity in pervasive computing. For event-driven distribution, Lambic integrates the properties of the AmOP paradigm, in what we call *futurised generic functions.* In addition, we provide a novel mechanism to allow flexible selection and composition of behaviour, called *predicated generic functions.* This mechanism provides language abstractions to influence method dispatch based on the program's context. For group behaviour, we propose a third extension called *group generic functions.* The main idea of this feature is to encapsulate the coordination of group interactions in the definition of services. Finally, a common underlying execution process ensures that these three features can be effectively used in combination with one another. We validate our work by showing in a number of case studies how Lambic facilitates the natural integration and dynamic adaptation of pervasive computing services.

Acknowledgements

I would like to express my deepest gratitude to everyone who helped me to complete this dissertation.

I would like to start thanking Theo D'Hondt and Wolfgang De Meuter for the invaluable opportunity you gave me to do the PhD. Many thanks for your advice and encouragement throughout all these years. I would also like to thank Pascal Costanza for his guidance. Many thanks for always believing in my work, especially during the toughest periods. All my respect goes to you three!

I would like to thank the members of the jury: Prof. Dave Clarke, Prof. Didier Verna, Prof. Viviane Jonckers, Prof. Beat Signer, Prof. Bernard Manderick and Prof. Luc Steels. Thanks for the time you invested reading my dissertation and for the questions and feedback during the private and public defences.

Many thanks to all the members of the Software Languages Lab. I am especially indebted to my partners in crime: Peter Ebraert and Brecht Desmet for all the collaborations at the early stage of my PhD, Elisa Gonzalez Boix for the many works we did together and for reviewing my dissertation, Tom Van Cutsem for those discussions which significantly influenced my research, and Engineer Bainomugisha for the endless but extremely helpful brainstormings. Special thanks to Charlotte Herzeel for finding a name to my programming language model, and for translating the abstract of this dissertation to Dutch. Last but not least, thanks to Lydie Seghers, Brigitte Beyens and Simonne De Schrijver for their permanent support with the administrative matters.

I also thank all the colleagues from other universities with whom I had the opportunity to collaborate in the context of the Moves and Varibru projects. Special thanks to Boriss Mejías, Sebastián González, Éric Tanter, Alfredo Cádiz, Eddy Truyen, Frans Sanen, Andreas Classen, Arnaud Hubaux and Patrick Heymans.

For the record: The hardest moments of writing the dissertation were when I thought that this was such a busy matter that I could not spend time in other activities. **Totally nonsense**. The fewer social activities I had, the worse I felt and less productive I became. I am deeply grateful towards my friends who, despite my stubbornness, still invited me to have a drink, to do sports, etc.

Finally, I would like to thank my family. Many thanks to the great family Petitprez for the permanent support all these years. Thanks to my parents, Mary and Jorge, my sister Macarena and my sons Benjamín and Nataniel. You all helped me to give my work just the relevance it had. Lo hicimos, mierda!

I dedicate this dissertation to my wife, Sonia. Obtaining the PhD degree makes sense only because we are together. Many, many thanks for your patience and love.

Jorge Vallejos
7th July 2011

Contents

List of Figures

List of Tables

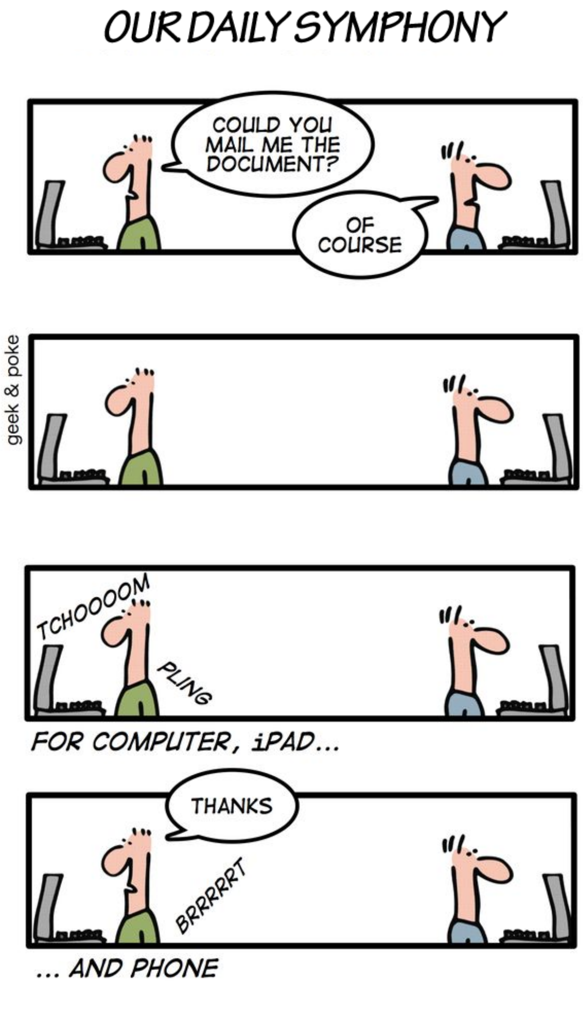


Figure 1: Our Daily Symphony [Oli11].

Chapter 1

Introduction

A central trope of pervasive computing is that software technology is moving "off the desktop", spread around computers embedded into everyday devices [BD07]. Pervasive computing services are expected to combine their functionality, maximising the use of devices available in living and working spaces. In doing so, the services have to deal with the highly dynamic nature of pervasive computing environments: devices with limited connectivity can appear and disappear from the users' surroundings as they move about. An increasingly popular way to deal with such dynamicity is to model the services in an *event-driven* fashion [GDL+04]. In this model, services can discover, communicate and react to changes in the environment, by means of asynchronous events. Then, the services can manifest their interest in particular events by defining handlers in the form of continuation or *callback* functions. This event-driven approach gives developers the means to build services that react to their dynamic environment. However, such support typically comes at a price in the evolvability of the programs. The more events a service has to be aware of, the more cumbersome its control flow becomes [HCN08]. The lack of adequate language abstractions to handle such events results in programs that are difficult to maintain and extend.

This dissertation presents the results of our study on software modularity for pervasive computing. In particular, we focus on modularisation techniques for event-driven distributed services. In our work, we identify two major concerns in the behaviour of such services: the capacity to adapt their functionality to environmental changes —a property known as *context-dependent behaviour* [GMH07]—, and the capacity to integrate their functionality with each other —property known as *group behaviour* [GFGM98]. The main issues concerning modularity is that both concerns require changes of behaviour that in most cases are scattered in the programs, and that can be determined only at runtime.

We ground our research in the object-oriented programming paradigm. At present, we can find an important number of proposals that extend this paradigm with language abstractions for event-driven distribution. Similarly, we find advanced object-oriented techniques to modularise dynamic behavioural adaptations. However, thus far there is no single approach that combine these two efforts.

Therefore, the central claim of this thesis is that in order to build evolvable pervasive computing software, we need a programming model that provides unified support for event-driven distribution and behavioural modularisation.

In the remainder of this chapter, we further introduce the context of our work and the problems to be tackled. We then outline our programming language-driven solution, giving an overview of its contributions. Finally, we provide the reader with a roadmap of the dissertation.

1.1 Research Context

The research context of this dissertation is the following:

Pervasive computing Originally envisioned by Mark Weiser in [Wei91], pervasive, or ubiquitous, computing is the result of hardware technology becoming increasingly powerful and affordable. As such, a wide variety of computing devices can be deployed throughout the users' environments. These devices are expected to coordinate with each other providing universal access to software services [Der99].

Mobile ad hoc networks We centre our research on a particular kind of pervasive computing environment, known as mobile ad hoc networks or MANETs [VME$^+$07]. MANETs consist of a set of wirelessly interconnected stationary and mobile devices, which can dynamically change topology as users move about. These networks are characterised mainly by the volatile connections that devices can sustain with each other, and their very little or inexistent fixed infrastructure.

Modularisation techniques The dynamicity of MANETs requires that devices *embrace* the environmental change and *adapt* their behaviour accordingly [GDL$^+$04]. Therefore, we focus on modularisation techniques to cleanly separate different behavioural variations, and to dynamically select and compose them. Thus far, we observe that although there is a considerable amount of research on behavioural modularisation, none of the existing approaches have been adapted to cope with the characteristics of MANETs.

Programming language design The work described in this dissertation concentrates on programming language design. We look into dedicated language abstractions to achieve modular definitions of pervasive computing services. This choice is motivated mainly by the expertise on this research domain at our laboratory.

Object-oriented programming We realise our research in the context of the object-oriented programming paradigm. We adhere to the premise that objects are good candidates for modelling units of distribution [GF99]. Therefore, this thesis investigates how the bastions of object-oriented programming (encapsulation, dynamic dispatch, inheritance, etc.), can be adapted to cope with the need for modularisation in pervasive computing.

1.2 Problem Statement

The dynamicity of computing fields such as mobile and pervasive computing have been the main reason for the uprising of event-driven distributed execution models [GDL$^+$04, VME$^+$07, Eug07]. Object-oriented programming languages integrating such models typically assimilate event-driven interactions into message-passing object communication. Special attention is paid to reflect the effects of distribution issues in remote interactions (e.g. network failures). For this, the languages provide dedicated abstractions (e.g. for remote message sending, message reception and result handling), which commonly differ from those used in standard local object communication. Such explicit language support eases the understanding of event-driven remote communication in programs. It also provides the adequate level of abstraction to deal with the distribution issues. Yet, this support also entails a number of negative effects for the program modularity. That is, for the ability of the languages to foster separation of the different concerns of a program [Dij82]. As we extensively discuss in Chapter 3, current language approaches with explicit support for event-driven distribution suffer from acknowledged issues such as *inversion of control* [HO06], *lost continuations* [FMM07], *asynchrony contagion* and *event interleavings* [SRRB10].

We analyse the impact of those issues on the modularity of two main concerns of pervasive computing services: context dependency and group behaviour. Context-dependent behaviour represents the influence of environmental conditions on the functionality of pervasive computing services [GMH07]. Group behaviour responds to the need for coordinating services that provide similar or complementary functionality [GFGM98]. Both concerns require a programming model that enables a *modular definition*, *dynamic selection* and *composition* of the different variations of behaviour.

The lack of an integrated support for event-driven distribution, context dependency and group behaviour, forces developers to manually deal with non-trivial interferences between these concerns:

- First, developers should ensure that the modularity boundaries of context-dependent behavioural variations do not conflict with those of group behaviour.
- Second, developers should ensure that the boundaries of behavioural variations do not conflict with those required to handle event-driven remote interactions.
- Last but not least, developers have to manually check that the issues of distribution (such as disconnections) do not hamper the dynamic selection and composition of behavioural variations.

To the best of our knowledge, no existing programming approach provides the means to deal with all these issues. We support this claim presenting an exhaustive study of the state of the art in object-oriented programming techniques for event-driven distribution and for modularity of context dependency and group behaviour (cf. Chapter 3).

1.2.1 Case Study: Pervasive Identities

To illustrate the problems tackled in this thesis, we introduce a particular, yet representative, scenario of pervasive computing. Chapter 2 discusses the way in which traditional "desktop" applications (e.g. drawing editors, communication applications, etc.) can be redesigned to exploit pervasive computing environments. We observe that in a setting with several computers available for a user, the need arises for such applications to be replicated on several of those computers. The applications abstract from specific locations and identities, and run in the environment as a single pervasive service. We refer to this characteristic as the need for *pervasive identities*. This requires that developers deal at the same time with the concerns of distribution, context dependency and group behaviour. An application with a pervasive identity should then be able to coordinate the tasks among the participating (remote) entities, and adapt this coordination to the dynamic changes in its context of use.

1.3 Research Goals

The primary goal of the research described in this dissertation is to enable the modular definition of pervasive computing services. This implies the following subgoals:

- To propose a new understanding of the effects of the pervasive computing paradigm in the identity and behaviour of software services.
- To review the interactions between programming language models for event-driven distribution, context dependency and group behaviour. We then propose a list of requirements for modularity in pervasive computing.
- To extend the object-oriented programming paradigm with a unified set of abstractions for distribution, context dependency and group behaviour. This way, we achieve our primary goal by means of a "proof by construction" that does not hinder the benefits of modularity of object-oriented programming.

We achieve our proof by construction by developing a novel set of language constructs in a programming model called *Lambic*.

Lambic. Lambic is a proof-of-concept extension to the Common Lisp programming language, used for our experiments on software development of pervasive computing services. Lambic achieves event-driven distribution by adopting a programming paradigm recently defined at our laboratory, called the *ambient-oriented programming* [DVCM$^+$06] (AmOP) paradigm. This paradigm provides dedicated properties to deal with distribution issues of pervasive computing.

Lambic extends the Generic function-based model of the Common Lisp object system [BDG$^+$88] with a combined language support for event-driven distribution, context dependency and group behaviour. In Lambic, pervasive computing services are modelled as classes and their behaviour as methods contained in generic functions. Generic functions can be accessed both locally and remotely. Remote invocations to generic functions are processed in an event-driven manner. Additionally, Methods can be specialised not only on the class of their arguments, but also on arbitrary *context predicates.* Finally, in Lambic we enable group behaviour to be part of generic function definitions, cleanly modularised in special abstractions, called *group methods.* All these extensions have been implemented in such a way that the original semantics of Common Lisp's generic functions are preserved.

1.4 Programming Language Approach

The dynamic nature of pervasive computing environments requires highly adaptable services. Yet this adaptability should not hamper the maintainability and extensibility of the programs. A primary goal of our research was to look for a programming language solution that ensures these three conditions.

1.4.1 Ambient-oriented Programming

To cope with the dynamicity of pervasive computing, we base our solution on an extension of the object-oriented programming paradigm, called *ambient-oriented programming* [DVCM+06] (AmOP). The distinctive characteristic of AmOP is that it is the only paradigm designed especially for mobile ad hoc networks. This paradigm identifies a number of requirements to keep programs from inconsistencies resulting from distribution issues such as volatile connectivity and limited infrastructure. As we extensively explain in Chapter 3, such requirements promote an event-driven execution model that allows for highly decoupled interactions between distributed services, with an underlying support for disconnections.

The AmOP paradigm has been realised in an object-oriented programming language, called *AmbientTalk* [VME+07]. AmbientTalk fulfils the requirements of AmOP by featuring a model for event-driven concurrency and distribution, known as the *event loops model.* In our solution, we used this model as the starting point of our research on modularity for pervasive computing.

1.4.2 Modularising Event-driven Distributed Programs

In the object-oriented programming research literature, the introduction of explicit support for concurrency and distribution has given rise to a number of long-lasting debates. Two examples that are relevant in the context of our research are the *myth of transparent distribution* [GF99] and the *inheritance anomaly problem* [MY93]. In what follows, we briefly introduce these problems and discuss their implications for our research.

The Myth of Transparent Distribution

In object-oriented distributed computing, there has been a long-standing discussion about the inconvenience of hiding distribution behind traditional object communication abstractions [WWWK96, GF99, EGS00]. Several works have pointed out that distributed interactions are inherently unreliable due to the effects of network failures, and thus can be hardly comparable to local interactions. Guerraoui and Fayad refer to this issue as the *myth of transparent distribution* [GF99], arguing for an alternative approach where developers need to be aware of distribution. As such, the complicated aspects of this concern can be exposed in a controlled manner by means of explicit language abstractions. Yet, as we previously explained in this chapter, such a special support can also become a major obstacle for the modularity promoted by object-oriented programming.

In this thesis, we adhere to this idea of making distribution explicit in the programs. Yet, to mitigate the conflict with modularity, we re-evaluate existing event-driven distribution approaches in the light of this concern. We then reflect the insights gained in this study in our model which allows for modular event-driven distributed programs.

The Inheritance Anomaly Problem

The problem of *inheritance anomaly* was originally coined by Matsuoka and Yonezawa in [MY93]. It focuses on the implications that concurrency models have on standard object techniques for behaviour encapsulation and reuse (classes and inheritance). In a nutshell, concurrency models require the specification of synchronisation constraints. Such constraints can become intertwined with the behaviour of the classes which can severely obfuscate the semantics of inheritance. During the 90s, the quirks arising from the coexistence of inheritance and concurrency were considered so critical as to suggest the removal of inheritance from concurrent object-oriented languages [MS04, Ame91].

As Van Cutsem states in [Van08], concurrency is a natural phenomenon of services deployed in distributed networks. Hence, the question about the conflicts with inheritance is also applicable to this domain. At present, we can find a number of approaches that tackle the inheritance anomaly problem by extracting concurrency specifications from the behaviour of programs [MS04, VME^{+}07, HO09]. However, we also observe that the explicit language support that such approaches provide for event-driven distribution, can raise additional conflicts with inheritance. For instance, when the effects of distribution cannot be fully encapsulated in a method definition, non-trivial interdependencies between a class and its subclasses are created. In this thesis, we refer to this issue as *asynchrony contagion*. We then propose a programming model for event-driven distribution which properly handles such contagion.

1.4.3 Extending Generic Functions-based Object-oriented Programming

In our work, we use Common Lisp as our implementation platform. This language provides advanced support to facilitate the definition and extension of object-oriented programming models, both in terms of syntax (e.g. through its macro system) and execution semantics (e.g. through its metaobject protocol [KRB91]).

Common Lisp features a *generic function*-based object model, known as the Common Lisp object system (acronym CLOS) [BDG+88]. In the context of event-driven distributed services, we observe that a proper integration between the semantics of event-driven execution and generic functions still remains an open issue. Event-driven interactions are typically represented as messages exchanged between objects. However, in a generic function-based approach programs are written in terms of function invocations. Several extensions to CLOS have acknowledged this problem by introducing message-passing semantics, e.g. a "send" remote operation [HT99, GSW+02, Ger05, Har08]. However, if the non-distributed part of a service is written using plain generic functions, the overall program is forced to combine the two different paradigms, which is far from trivial. Furthermore, such a combination seriously diminishes the benefits of generic functions, not in the least their multiple method dispatching semantics [BDG+88].

In this work, we present an object-oriented programming model that reconciles event-driven programming with generic functions for concurrency and distribution (cf. Section 4.2). The main idea of this model is to represent event notifications as asynchronous generic function invocations which are sequentially processed by the actors dispatching to the appropriate generic functions.

1.5 Contributions

In this section, we highlight the major contributions of this thesis.

Modularity in Ambient-oriented Programming. In the context of pervasive computing, we augment the ambient-oriented programming paradigm definition with a list of requirements for modularity. In particular, our model modularises context dependency and group behaviour in event-driven distributed programming. For this, we characterise the issues of existing event-driven distributed models. We then evaluate the impact of such issues on the modularity of context dependency and group behaviour.

A Generic Function-based Model for Event-driven Distribution. We propose an object-oriented programming language model that gracefully aligns generic functions with event-driven distributed programming. In this model, called *futurised generic functions*, we introduce the notion of asynchronous remote invocations of generic functions. That is, the possibility to invoke remote generic functions in an event-driven manner, without reverting to traditional message-passing semantics. As such, we preserve the multiple dispatching semantics of generic functions in a distributed setting.

Explicit and Uniform Language Support for Event-driven Distribution. We provide an in-depth study of the programming language support required for models of event-driven distribution. We investigate the extent to which such support can be aligned to standard object-oriented techniques. The result of this study is a flexible programming language syntax which allows for different levels of *explicitness* of distribution interactions in the programs. We have integrated this syntax into the futurised generic function model. Futurised generic functions provide explicit syntax for distribution, i.e. for asynchronous generic function invocations and for asynchronous result handling. Additionally, this model provides an internal process to handle asynchronous results. This model allows distributed computations to use the same syntax as local computation, while internally still executing them in an asynchronous manner. Providing both styles of syntax has enabled us to better understand their benefits and drawbacks for coping with the distribution issues.

Generic Functions with Context Predicate Dispatch. We provide a novel mechanism to allow flexible behaviour selection and composition, called *predicated generic functions*. This mechanism provides language abstractions to influence method dispatching semantics based on the programs' context. In our model, method definitions can be guarded by context predicates, which are used to decide on the applicability of the method for a list of actual arguments. Predicated generic functions enable users to establish a priority order between possibly logically unrelated predicates. If more than one predicated method is applicable, the order in which the predicates are declared in the corresponding generic function is used as tiebreaker. These main tools offer a fine-grained control of applicability and specificity of methods.

Generic Functions for Group Behaviour. We propose an innovative approach to deal with distributed service groups in object-oriented programming, called *group generic functions*. In this model, group identities are represented as group classes. A method invoked on an instance of such a class can be implicitly propagated to some or all the instances of the group class. This is defined in a group protocol which is cleanly modularised in group methods. Group protocols are defined on a per-method basis: for each method representing an operation of the class, there can be a group method defining the group protocol for such operation.

Pervasive Identities. We validate our work by realising a unique characteristic for software services in the pervasive computing paradigm, known as *pervasive identities*. Pervasive identities promote the natural integration of nearby services to provide ubiquitous access to their functionality. By implementing this condition, we illustrate how the three components of our solution can seamlessly work in coordination.

1.6 Dissertation Roadmap

The central scientific contribution of this work is to propose a model for software modularity in the Ambient-oriented Programming paradigm. Below we summarise the chapters of this dissertation.

Chapter 2: Modularity in Ambient-oriented Programming presents the non-trivial requirements for modularity of pervasive computing services. We illustrate these requirements by means of a number of scenarios, envisioning a novel way for software services to exploit their environment, called *pervasive identities*. Then, we introduce the ambient-oriented programming paradigm and the two properties of behaviour we aim to modularise: context dependency and group behaviour.

Chapter 3: Related Work gives a detailed view of the state of the art of object-oriented programming models for event-driven distribution, context dependency and group behaviour. We review each approach according to the requirements identified in Chapter 2. Special emphasis is given to analyse the interaction between the language abstractions proposed for the different concerns. We end the chapter with a revised version of the list of requirements for modularity in pervasive computing.

Chapter 4: Ambient-oriented Programming in Lambic presents the incarnation of the ambient-oriented programming paradigm in our programming language model, named Lambic. While the traditional AmOP paradigm is based on remote messaging, our incarnation will be based on so-called *futurised generic functions*. First, we explain how futurised generic functions gracefully align ambient-oriented programming with multiple dispatch semantics of generic functions. Then, we propose and compare two complementary approaches for explicit and uniform language syntax for communication, which guarantees the event-driven execution of remote interactions.

Chapter 5: Context Dependency in Lambic describes *predicated generic functions*, a novel mechanism to allow flexible behaviour definition and selection according to context. First, we explain how predicated generic functions allow the expression of context-dependent behaviour in a declarative and modular manner, providing fine-grained control of method applicability and method specificity. Then, we present the integration between futurised and predicated generic functions. As result of this integration, context-dependent behaviour can preserve its modularity despite the distributed interactions implied in the behaviour.

Chapter 6: Group Behaviour in Lambic introduces a novel approach to group behaviour, called *group generic functions*. We explain the rationale behind this approach by means of a metaphor on the social conduct, known as *empathy*. The main idea is to associate group behaviour with classes. This way we abstract group concern from client programs. Group behaviour is defined in dedicated group methods. Thus, it is explicitly separated from the base logic of the classes.

Chapter 7: Validation illustrates the benefit of Lambic by implementing a number of scenarios of pervasive computing services. We present the development of the cases of pervasive computing services introduced in Chapter 2. In these scenarios, we show a number of programming patterns that become possible thanks to the integrated support for event-distribution, context dependency and group behaviour.

Chapter 8: Conclusion summarises the contributions made in this dissertation. We provide a global evaluation to Lambic with respect to the requirements of modularity. We then discuss the open issues and potential directions for future research.

Appendix A: Lambic in Common Lisp explain the main details of the implementation of our proof-of-concept programming model in Common Lisp. We pay special attention to show the use of the meta-object protocol of Common Lisp to achieve the integration between the three models composing our solution.

Appendix B: Lambic Syntax and Libraries summarises the syntax and built-in operations provided by the Lambic programming language model.

Appendix C: Lambic Kriek presents the full implementation of our first validation case, a pervasive communication service called Kriek.

Appendix D: Lambic Geuze completes the dissertation presenting the implementation of our second validation case, a pervasive computing drawing editor called Geuze.

Chapter 2

Modularity in Ambient-oriented Programming

A central idea in Mark Weiser's original definition of pervasive computing is that devices co-located in a certain environment should *seamlessly* interoperate with each other. This would enable the devices to integrate their functionality in order to maximise the environment's available computational capacities. At present, however, we observe that achieving such interoperation is far from trivial. This is mainly caused by the highly dynamic nature of pervasive computing environments: devices with limited connectivity can appear and disappear from the environment. To cope with this dynamicity, a new programming paradigm has been defined, called *ambient-oriented programming* [DVCM$^+$06] (AmOP). This paradigm identifies a number of requirements for the software development, especially for distribution concerns such as service discovery, communication and network failure handling. Such requirements are the essential ground for the development of collaborative services in pervasive computing environments. Yet, the AmOP paradigm says very little about how to deal with the effects of dynamic environments on the behaviour of the collaborating devices themselves. As we explain in this chapter, pervasive computing devices are expected to dynamically adapt their behaviour in response to changes in their environment —a property known as *context-dependent behaviour* (cf. Section 2.3). In addition, devices should adapt their behaviour according to the coordination schemes required for the collaborative services —a property referred as *group behaviour* (cf. Section 2.4).

As we will see, the lack of adequate support to define such behavioural variations may lead to programs that are hard to maintain and extend. We augment the AmOP paradigm with a list of requirements for the modularisation of context dependency and group behaviour in pervasive computing services.

2.1 Motivating Scenario: Pervasive Identities

To illustrate pervasive computing and its effects on the software development, we introduce two well-known types of software services —a communication service and a drawing editor— and discuss their adaptation to a pervasive computing environment. While it is trivial to directly extend the case study to other "desktop" services (e.g. mail applications, music players, etc.), it is also worth noting that there are other and new kinds of services envisioned for pervasive computing that exhibit similar characteristics and issues. In Chapter 7 we further evaluate these scenarios.

2.1.1 *Kriek*: a Pervasive Communication Service

Consider the case of a software application for communication. The main property of such an application is to offer multiple communication services (based on text, audio and video) requiring only one identity for every user. Assume that such an application —called *Kriek* in this thesis— runs in a pervasive computing environment comprising a smart phone, a laptop and a smart TV, as shown in Figure 2.1. It should be possible that the user has a single account that he can simultaneously use on all devices. This plurality of devices is transparent to the user's contacts who should still be able to address the user employing only one location. Additionally, the user may have some preferences on how to bring into play the different communication services. For instance, he may want to receive notifications of incoming communications at all the devices (e.g. a pop-up message), but get the corresponding chat windows in only one location (e.g. the smart phone for audio and text chat and the TV for videoconference). The user may also prefer to store the conversation records by the same device or set of devices. Finally, all communication services are conditioned to the context in which the communication occurs. For example, if the smart phone is about to run out of power, Kriek could suggest its user to proceed with the audio and text chats on the laptop instead; or if somebody else is watching TV at the moment the user receives a videoconference call, he may also prefer to use this service on the laptop.

2.1.2 *Geuze*: a Collaborative Drawing Editor

Consider as a second example a drawing editor —called *Geuze*— shown in the left part (side *a*) of Figure 2.2. In a pervasive computing environment, Geuze enables its users to have drawing sessions for sharing and editing each other's shapes in a common canvas. In this case, the operations a user performs are propagated to the rest of the participants of the session. As with the previous scenario, collaboration in Geuze is also influenced by the context in which it occurs. For instance, to keep a consistent drawing in all the session's editors, Geuze provides means to constrain the editing of a shape to only one user at a time, and to react to the disconnection of the participants, e.g. by hiding the disconnected participant's shapes. Similarly, Geuze controls the network traffic that the collaborative edition entails by supporting different propagation strategies for the graphical operations. For example, an editor propagates its user's performed operations (e.g. moving a shape, as shown in the right side of Figure 2.2) to the rest of the participants of the session, by

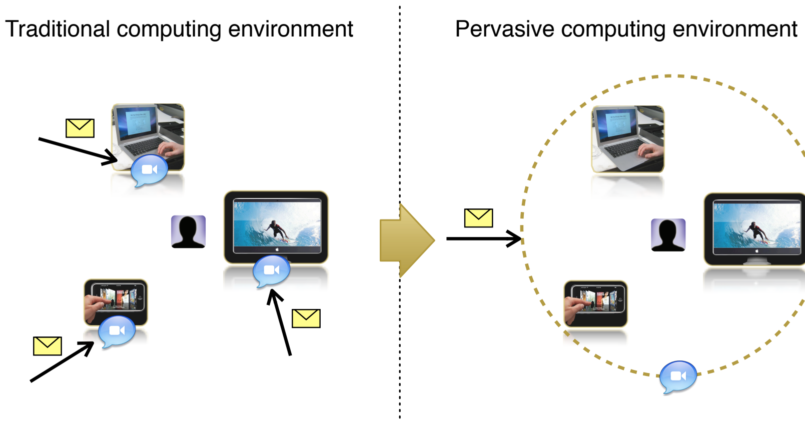

Traditional computing environment
Pervasive computing environment

Figure 2.1: The Kriek pervasive communication service.

communicating the shape's intermediate positions only to the editors that are currently displaying the part of the drawing where the shape appears,[1] and merely transmitting its final position otherwise.

2.1.3 Pervasive Identities

The scenarios above envision two orthogonal kinds of pervasive computing services. Kriek is a single-user service that has to dynamically distribute its functionality among the user's devices. Geuze, on the other hand, is a manifestation of realtime group collaboration. It integrates the work of several users ensuring consistency between their simultaneous interactions. Yet, the two scenarios share a common characteristic which stems from the very essence of the pervasive computing paradigm: In a setting where computers spread through the user's environment, software entities providing similar functionality should abstract from specific locations and identities, and run in the environment as a single pervasive service.

This service should be able to distribute the tasks among the participating entities, and adapt this distribution to the dynamic changes in its context of use. We call this property the *pervasive identity* of the services. Pervasive identities entail a number of issues which we roughly classify in three major concerns: *distribution*, *context-dependent behaviour*, and *group behaviour*. For the distribution concern we base our solution on the research results offered by the AmOP paradigm, described in the next section. We then present the requirements for context dependency and group behaviour. These concerns are not entirely mutually exclusive; in fact, both scenarios combine elements of the three concerns. As we discuss at the end of the chapter, this interdependency is actually more challenging than the issues raised by each concern individually.

[1] Assuming that there is a zoom functionality so that the canvas can have a different size at each editor's window.

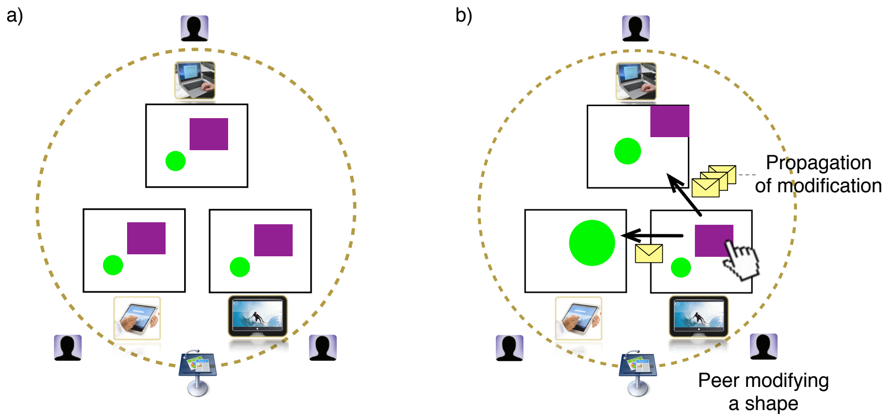


Figure 2.2: The Geuze collaborative drawing editor.

2.2 Ambient-oriented Programming

Pervasive computing services run in an inherently concurrent and distributed environment. This environment consists of a set of stationary and mobile devices which are wirelessly interconnected, forming a network that dynamically changes its topology as the users move about. Van Cutsem et al. [VME$^+$07] characterise such a mobile ad hoc network —or MANET— essentially by two hardware phenomena that clearly distinguish it from traditional fixed networks: *volatile connections* and *zero infrastructure*.

Volatile connections. The first phenomenon present in MANETs is related to the fact that connections between devices may be regularly interrupted as result of the user mobility and the limited capabilities of the devices in terms of network connectivity, transmission range and battery life. In several cases the disconnection is temporary which means that the devices may eventually meet again and require their connection to be re-established. As an example consider the case of Kriek whose user wants to always store this conversation records on the same device, e.g. on the laptop. For a situation in which the user is text chatting on his cell phone, it is not guaranteed that the connection with the laptop will remain always available (e.g. if the user moves away from the laptop with his cell phone). However, eventually it should not be a problem to transmit the chat transcripts from the cell phone to the laptop as presumably these devices will connect again at some moment later in time.

Zero infrastructure. The second phenomenon in MANETs is their non-existent shared infrastructure. In such networks, services become dynamically available according to the connection state of the devices that contain them. A service that relies on other services in its environment to perform its task needs to be aware of the availability of such devices. The lack of a shared infrastructure demands the devices (or more accurately their services) to be autonomous so that they can cope with necessary resources being unavailable for an extended period of time. This is a major issue for services such as Geuze, which cannot rely on a particular machine, i.e. a fixed server, to ensure consistent collaborative editing (as in web-based collaborative applications [Goo09]), as there is no certainty that the connection with such machine will be permanently available.

2.2.1 Requirements for Distribution

These phenomena entail a number of requirements for the development of pervasive computing services, which Dedecker et al. [DVCM+06] initially bundled as the *ambient-oriented programming* paradigm, and Van Cutsem et al. [VME+07] further refine as follows:

$\mathbf{R_{D.1}}$ Decentralised service discovery Any application that relies on other devices to perform its task should be able to discover its dynamic network environment. Yet this discovery should not be centralised in one machine. A decentralised service discovery protocol needs to be introduced to enable the services to autonomously act upon the availability and unavailability of nearby services.

$\mathbf{R_{D.2}}$ Decoupled communication In MANETs, the communication between services should be independent of the volatile connectivity of their devices. This means that the services do not necessarily need to be online at the same time to communicate with each other (a condition known as *time* decoupling). Similarly, the potentially extended periods of disconnection during a communication imply that synchronisation between different parties should be performed without blocking their control flow, i.e. without suspending their thread of control (*synchronisation* decoupling).

Services should also be able to communicate without knowing each other's address or location beforehand (*space* decoupling). This enables the services to better adapt to changes in their physical environment as the conceptually same service may be provided by several instances hosted by different devices. Such anonymity also enables a service to be represented as more than one instance (*arity* decoupling).

$\mathbf{R_{D.3}}$ Connection-independent failure handling Services should be able to perform failure handling independent of any network failure. The reason for this is that disconnections can be transient and as such services may want to resume computation upon reconnection. Treating disconnections as a normal mode of operation is an optimistic form of partial failure handling (and also higher level than physical network failures).

Remember that the common principle in all these requirements is that it is not possible to hide distribution completely from our programs [WWWK96]. Distribution has a major impact in the interaction of the services with their environment, and as such this concern should be properly supported at the program level.

2.3 Context Dependency

Context dependency is the ability of a software service to acquaint its execution environment and adapt its behaviour accordingly. As Hirschfeld et al. indicate in [HCN08], this ability is already an integral part of regular business applications, where the context is *any information computationally accessible upon which behavioural variations may depend.* Information such as the users' personal data and preferences have always been employed to customise services (noticeably in internet computing). Also, the state and historical information of the services have been used to change their functionality at runtime, as illustrated in behavioural design patterns such as the *state* and *strategy* patterns [GHJV95]. But context dependency is becoming even more critical because of the ever larger sensing technology embedded in today devices. Sensors reify the devices' physical environment making available a broad set of information such as location, light, temperature, movement, and proximity [BKZD04]. In an application domain such as pervasive computing, where users are permanently surrounded by such devices and sensing data, software services are expected to become highly context aware, adapting their behaviour to the users' environmental state. In doing so, the services have to cope with a number of issues related to the nature of such contextual information, and the way it is incorporated in the services' behaviour.

In the research literature, we find two main issues that characterise the context of a pervasive computing service and influence the development of context-dependent behaviour: *dynamicity* and *heterogeneity* [BC06, SDA99, DVC$^+$07].

Context dynamicity. In pervasive computing, the context of a service can dynamically change over time without any periodicity and predictability, depending on diverse factors ranging from user mobility to operations performed by the devices (automatically or in response to user interactions). Concerning the context-dependent behaviour of the service, the possibility exists that with such dynamic changes some context condition becomes invalid while the behaviour it has triggered is already being executed. In fact, the injudicious abortion or continuation of context-dependent behaviour could lead the service into an inconsistent program state [DVC$^+$07]. The problem increases if we take the inherent concurrency of pervasive computing environments into account. As concurrent changes of context may presumably require different behavioural adaptations, it is often difficult to ensure that services do not end up with adaptations that conflict with each other.

When dealing with context dynamicity it is important to notice that not all the changes in the services' context leads to adaptations in their behaviour. Similarly, not all the context changes that concern a service influence its behaviour in the same way. There are some context changes that can directly prompt an adaptation, e.g. an immediate change in the service's state. Others, instead, are taken into account only for later computations, e.g. to determine the appropriate adaptation to process an invocation to the service's functionality.

In the scenarios presented in this chapter, we find situations that require either kind of adaptation. Examples of prompt context-dependent adaptations are the Geuze editors that react to a peer joining or leaving the editing session by immediately adding or removing the peer's shapes from their canvas respectively. Examples of delayed context-dependent adaptations are the different configurations that the Kriek service can adopt to receive a communication call, i.e. the user's devices that should be involved in the communication. These may depend on the type of communication (video, audio or textual), the devices' internal conditions (e.g. battery level) and the number of people using a device (e.g. the smart TV). In this second case, adaptation is required only when the call occurs and not every time the devices' conditions change or a user logs into or out of a device.

Context heterogeneity. A context is a heterogeneous collection of information which varies from one device to another depending on the devices' accessible sensing data, its various user profiles, and services in which it is involved. In some cases, the same type of information can have dissimilar data formats tightly related to the sensing sources. A typical example of this is the location data which can come from global positioning systems, GeoLocation services (location deduced from the devices' IP address), and WiFi triangulation or active badges (used for indoor location) [Wik11a]. To provide uniform access to such data, sensing details are often abstracted using transformation techniques based on context ontologies [PVW$^+$04]. A major consequence of context heterogeneity, though, is that as the number of available data increases, developers have to deal with a combinatorial explosion of possible context-dependent adaptations [DVC$^+$07]. An adaptation may depend on a combination of context conditions (in some cases referred as *context aggregation*[2] [YB05]) and a context condition may trigger several adaptations, often affecting different parts of the overall service's functionality. Reasoning about such combinations inside of the program would lead to undesirable situations such as scattered conditional statements, resulting in cluttered code that is hard to maintain and evolve [CH05].

The issues raised by context heterogeneity can be perceived in the graphical operations of the Geuze editor. Each operation has its own interaction pattern defined in handlers for GUI events. A pattern can require a combination of GUI events (e.g. moving a shape is specified in handlers for the *mouse-down*, *mouse-move* and *mouse-up* events). Additionally, the same GUI event can be used in several patterns (e.g. the *mouse-move* event is used to move, paint and draw shapes). We further develop this develop this scenario in the Section 7.2.

2.3.1 Requirements for Context Dependency

Context dependency requires explicit programming technology support which we summarise in the following requirements derived from Hirschfeld et al.'s notion of *context-oriented programming* [HCN08]:

$R_{C.1}$ Modular behavioural variations Introducing context-dependent behaviour to a service does not imply that the code required to reason about the context should

[2]The term `context aggregation` is mostly used to combine and refine raw sensor data [YB05].

get entangled with the rest of the service's program. Context-dependent adaptations, as well as the reasoning process that they require, should be modularised to avoid their entanglement and scattering in the main functionality of the program.

$R_{C.2}$ Dynamic Selection Context-dependent adaptations should be selected dynamically at runtime, either in response to context changes or proactively at specific points in the program execution.

$R_{C.3}$ Consistent composition An important requirement for using partial adaptations is that the resulting behaviour of the application should be a consistent composition of its default behaviour and the adaptations.

$R_{C.4}$ Restricted Scope In most cases, context-dependent adaptations affect only parts of the program. Therefore, adaptations must be circumscribed to a restricted scope of action, which should be unambiguous even in the presence of concurrent interactions.

2.4 Group Behaviour

By encapsulating a set of cooperating services, group abstractions have proven to be very convenient for achieving continuous availability through replication or sharing computational load [GFGM98]. In pervasive computing, group behaviour also responds to the need for coordinating the functionality of software services in order to maximise the use of the resources available in the users' surroundings. Service groups therefore become a natural way to interact with their users and to cope with the dynamicity of pervasive computing environments.

Natural Grouping. In pervasive computing, the availability of the services is determined by the set of devices found in the users' environment. In some cases, the services may correspond to software applications originally conceived to run autonomously in one device, and for single users (e.g. "desktop" applications). However, the multiplicity of devices in the environment can bring about situations in which the users have several of such services at their disposal, possibly providing similar functionality (e.g. instances of the same desktop application) and similar properties (e.g. a common user). Instead of remaining autonomous, the services are expected to be grouped together in such cases, combining their functionality and interacting with their users as a single entity. Requests issued to such service groups are thus transparently propagated and processed by the group members.

We can observe the natural grouping of services in the scenarios of Kriek and Geuze. The group abstraction enables maximising the use of surrounding resources required for communication (text, audio and video inputs and outputs) in the case of Kriek, and to keep a consistent collaborative edition in the case of Geuze.

Group coordination. Coordinating service groups in pervasive computing corresponds to managing the dependencies between the participating services in order to achieve a consistent and resilient form of collaboration, even in the presence of network failures. As Guerraoui and Rodrigues explain in [GR06], group collaborations can be modelled as cases of distributed *agreement* problems. For instance, the services may need to agree on whether a certain event did (or did not) take place, the order by which a set of requests need to be processed, or a common sequence of actions to be performed. Additionally, more sophisticated agreements can emerge from solutions to simpler agreement problems. For instance, the services need to agree on a common representation of their identity and a reliable way to communicate with each other. In case of having several ways to perform a group task (e.g. the number of participants involved in the task), the services need to reach a consensus on the most appropriate option according to the context in which the task occurs. The services may even need to agree on each step corresponding to the execution of the task (agreement problem known as *atomic commitment*) and also on the order in which such steps should take place (agreement problem known as *total order broadcast*).[3]

Similar to the case of context dependency, the risk exists that the support required for dealing with all of the above coordination concerns gets tangled with the functional behaviour of the services as stand-alone applications, still used when there are no nearby peer services. This situation is particularly cumbersome for pervasive computing services which should be able to dynamically switch behaviour, from stand-alone to group member and vice versa, as they can join or leave a group (voluntarily or as consequence of network failures). The inadequate modularity of these two behaviours can lead to undesirable scattered conditional statements.

2.4.1 Requirements for Group Behaviour

To support the development of group behaviour in pervasive computing services, we identify the following requirements based on the works of Black [BI93] and Guerraoui et al. [GFGM98]:

$\mathbf{R_{G.1}}$ **Plurality encapsulation** In a pervasive computing environment, services sharing common functionality and properties should be able to naturally become a group. The number of services that compose the group should be hidden from clients who interact with the services as if they were a single entity. In the existing research literature, this requirement has been formulated as the need for *encapsulating plurality* [BI93] or *arity decoupling* [VDM07].

$\mathbf{R_{G.2}}$ **Group coordination protocols** Service groups require a coordination protocol to consistently manage the dynamics of the groups in terms of membership and communication, even in the presence of network failures and disconnections. Such a

[3]Atomic commitment is particularly relevant for processing distributed transactions, whereas total order broadcast is the basis of one of the most fundamental techniques to replicate computation in order to achieve fault tolerance [GFGM98].

coordination protocol should provide means to define agreements according to the conditions of each group collaboration.

$R_{G.3}$ Modular group behaviour The coordination protocol should not interfere with the functional logic of the service. It should be possible to transparently plug this protocol into services that were originally developed without group concerns in mind.

2.5 Entangled Concerns

The three concerns described in this chapter (i.e. distribution, context dependency and group behaviour) reveal a number of issues that characterise pervasive computing services. For each concern, we have presented a list of requirements to cope with such issues. Yet, these concerns do not occur in isolation. The interdependence between them is in our experience a major problem. This means that developers have to reason not only about how to introduce distribution, context dependency and group behaviour in the programs, but also about the interference that can occur between these concerns, which is far from trivial.

To illustrate this issue we outline a concrete use case of the Geuze service about the editing of a shape (further elaborated in Chapter 7). It consists mainly of two steps: selecting the shape (a user selects a shape from his editor; the editor acknowledges the selection e.g. by drawing a halo around the shape), and modifying the shape (e.g. the user moves the shape). The implementation of this use case for the Geuze editor running as a stand-alone service implies basically the handling of a number of events triggered by the editor's GUI. When modelling the editor as a pervasive computing service, however, this use case should also cope with the distribution, context dependency and group behaviour concerns as follows:

Group behaviour The edition of the shape should adapt to the context of collaborative drawing sessions. This implies working in accordance with coordination schemes required to keep consistent replicas of the drawing in the session's editors. Such a scheme may involve, for instance, the designation of a leader editor that regulates the access to the different shapes of the drawing, allowing each shape to be modified in only one peer editor at the time. In this setting, the step of selecting the shape should include a request to the leader editor, which acknowledges the selection only if the shape is not already in use. Then the leader should inform the selection to the session's editors so that all of them display the selection effect (the halo) on the shape. Also, in a collaborative session the modification (the move of the shape) should be propagated to all the session's editors.

Context dependency There are several ways in which context information influence the editing of a shape. Most of them can be detected only at runtime requiring the introduction of conditional expressions in the implementation of the use case. For instance, the interaction with the leader and the propagation of the selection and move of the shape, should be performed only if the editor is participating in a session. Also, the shape can be selected only if it is available (condition evaluated by the leader editor). Finally, propagating the move of the shape depends on the window's position at each editor, as explained in Section 2.1. The shape's intermediate positions during the move are propagated only to the editors displaying the shape.

Distribution Collaborative sessions as described in the previous points, make the editing of a shape prone to distribution issues. First, the two operations composing this use case require one or more interactions with a remote editor (with the leader editor in the case of the selection, and with all the editors in the case of the move), and as such they can be affected by network failures. The implementation of these operations should then provide some means to handle possible disconnections (transient or permanent) during the remote interactions. Second, editing a shape should have consistent effects in the collaborative session even if the leader editor is affected by a network failure. Because the leader is used as a synchronisation point to avoid simultaneous editing of the same shape, the session should deal with the leader's disconnections, e.g. by electing a new leader or by keeping replicated leaders. Thus, some issues that need to be tackled in the implementation of the use case are, for instance, how the editors can be aware of the changes of leader, or what to do in case the leader gets disconnected while a peer editor is editing a shape (e.g. after allowing the selection but before the move is finished).

In the above description of the use case, we observe that the three concerns are tightly interconnected. The group behaviour of the Geuze service is deeply influenced by the context (e.g. state of the editors and network failures). Context-dependent adaptations may depend on distributed context information which makes the adaptation process also vulnerable to network failures. None of the different sets of requirements identified for each concern can cope independently with this issue. This situation drives us to the definition of our last requirement:

$\mathbf{R_I}$ Interdependent support For situations in which distribution, context dependency and group behaviour cannot be handled in isolation, it should be possible to integrate the support provided for these concerns.

2.6 Summary

In this chapter, we identified three major concerns for the development of pervasive computing services —distribution, context dependency and group behaviour. We discussed the issues that each concern implies and derived a set of requirements for the development of pervasive computing services:

- The major issues of distribution are related to network failures and zero infrastructure of the devices. A model for this concern should then provide a decentralised service discovery mechanism, a decoupled communication scheme and connection-independent failure handling.

- Context dependency is characterised by the dynamicity and heterogeneity of the contextual information that can influence the behaviour of pervasive computing services. A model for this concern should therefore enable the definition of modular context-dependent adaptations, which can be dynamically deployed within a restricted scope of action, and can consistently interact with the default behaviour of the service.

- Group behaviour in pervasive computing responds to the need for coordinating software services that provide similar or complementary functionality. A model for this concern should encapsulate this plurality of services so that they can be addressed as a single entity, and enable the modular definition of coordination protocols.

- Last but not least, these concerns do not occur in isolation. This implies that the technological support for each of these concerns should also work in coordination with the technological support required for the others.

In the next chapter, we review the state of the art of software development models for pervasive computing in the light of these requirements.

Chapter 3

Related Work

We present the state of the art of software development models that deal with the concerns identified in the previous chapter: distribution, context dependency and group behaviour. As stated in the introduction of this dissertation, our target research domain is the *object-oriented programming* paradigm. Similarly, our starting point for distribution is *event-driven programming*, as required by the AmOP paradigm. We therefore focus the present literature study on solutions based on these programming models. Most of the existing work we describe in this chapter provides support for only one or two of the concerns of pervasive computing. Still, we include them as their understanding on they have significantly influenced the design of our contribution.

3.1 Event-driven Distributed Programming

Event-driven programming has increasingly gained popularity among emerging distributed computing paradigms such as mobile and pervasive computing [VME$^+$07]. In this programming model, software services discover, communicate and deal with network failures by means of events. Languages with an event-driven execution model are built around the concept of an *event loop*: a perpetual loop that processes events (usually, but not necessarily, from an event queue) and dispatches them to the appropriate event handlers, one at a time. The event handlers themselves are usually not allowed to suspend the thread of the event loop because doing so would make the entire event loop, and all other event handlers, unresponsive. Instead, an event handler posts follow-up event handlers (callbacks) that will be invoked by the event loop when subsequent events occur. Thus, waiting between concurrent activities is not performed by blocking but rather by *registering* the continuation of the computation with the event loop.

The actor model of concurrent programming [Agh86] is an archetypical example of such an event loop architecture. Actors are concurrent processes that share no state and interact strictly by means of asynchronous events in the form of messages. This means that actors cannot access each other's state synchronously which prevents race conditions on such data. Actors have also proven to be a clean abstraction for distributed systems where processes generally have no shared state and communicate via network messages [VA01, BBC$^+$06, DVCM$^+$06].

The actor model was originally conceived as a functional model, but extensions have been formulated that combine the actor model with object-oriented programming. In this dissertation, we use the object-oriented extension to the actor model featured by the AmbientTalk programming language [VME$^+$07], as the basis of our research. Such extension, known as *communicating event loops*, has been designed especially to fulfil the distribution requirements of ambient-oriented programming presented in Section 2.2. In this section, we describe the distinguishing properties of communicating event loops and refer to [Van08] for an exhaustive definition of their semantics and comparison with other event-driven distributed models. We then extend this comparison by discussing the *programming style* featured by the different approaches. We use this extra criterion to evaluate the integration of these models with existing language support for the context dependency and group behaviour.

3.1.1 Communicating Event Loops

In early actor languages (e.g. Lieberman's ACT-1 language [Lie87]), all values were represented as actors. This enabled a flexible and uniform programming model, but also made local sequential, non-distributed computing more complicated than necessary. Subsequent object-oriented extensions to the actor model represented actors as special *active* objects: objects with their own execution process and accessible via asynchronous message passing (for example, ABCL/1 [YBS86], NetCLOS [HT99], ProActive [BBC$^+$06], Salsa [VA01]). This enabled programs to explicitly distinguish remote from local computations, but also put an extra burden on the developers who had to determine for each part of the program whether it should be modelled as an active or standard object, which is far from trivial [Van08]. Recently, the E [MES05] and AmbientTalk [VME$^+$07] programming languages have introduced an execution model that allows objects and actors to gracefully co-exist. In these languages, actors are not represented as objects but rather as containers of objects. Both local and remote computations are modelled as message-passing interactions exclusively among regular objects (not actors).

Actors as Object Containers. Actors define boundaries of concurrent execution around groups of objects. Each object is *owned* by an actor. An actor can own multiple objects, but each object is owned by exactly one actor. Two objects owned by the same actor can communicate synchronously, by means of traditional message passing. However, objects may also refer to objects owned by other actors. Object references that span different actor boundaries are named *far references* and only allow asynchronous access to the referenced object. Any message sent to a receiver object via a far reference is enqueued in the mailbox of the actor that owns the receiver object and processed by the owner itself. Actors are event loops: they take messages one by one (i.e. sequentially) from their mailbox and dispatch them to the receiver object by invoking its appropriate method. Figure 3.1 depicts actors and objects in AmbientTalk.

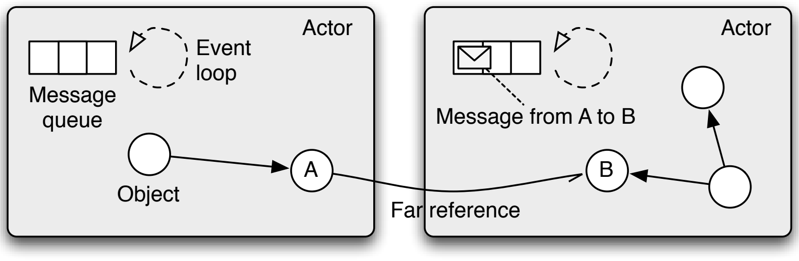


Figure 3.1: Actors in AmbientTalk

Asynchronous Message Passing. E and AmbientTalk combine the actor primitives to send and receive asynchronous messages with object-oriented message passing. Neither language features an explicit `receive` or `react` statement to receive messages from remote objects (as in the Erlang [Eri11] and Scala [Pro10] programming languages). Rather, message reception is represented simply as method invocation. An object can thus accept any messages for which it defines a corresponding method.

Non-blocking Futures. By default, asynchronous message sends do not return a result. E and AmbientTalk allow asynchronous message sends to return values by means of *futures* [Hal85]. A future is a placeholder for the return value of an asynchronous message send. Once the return value is computed, it replaces the future object; the future is then said to be resolved with the value.

To make the discussion more concrete, consider the following example. Assume `messenger` represents a far reference to a remote object that represents a communication service (as that explained in Section 2.1). The following AmbientTalk code shows how to query this service for its user's username:

```
def f := messenger<-getName();
when: f becomes: { |val| display(val) } catch: { |exception| ... }
```

The `<-` operator denotes an asynchronous send of the message `getName` to the remote `messenger` object. This operation returns a future `f`. In the communicating event loops model, an actor cannot suspend on an unresolved future. Actions that depend on the result of an asynchronous message are defined by registering an observer for its corresponding future, using the `when:becomes:catch:` construct. When the future is resolved, the observer (a closure passed as the `becomes:` argument) is called with the resolved value (`val`). To be able to respond to a `getName` message, it suffices for the `messenger` object to define a `getName` method as follows:

```
def createMessenger(name) {
  object: {
    def getName() { name }
  }
}
```

In the example above, the return value of the `getName` method is used to resolve the future that was created as a result of the `messenger<-getName()` message send. Exceptions raised during the asynchronous execution of the method, propagate up to the level of the asynchronous invocation. Since at this point the only available continuation is the future attached to the message, the exception is signalled by "ruining" the future. The exception of the ruined future can then be handled with the closure passed as the `catch:` argument. The execution of either the `becomes:` or `catch:` closures is always scheduled in the owning actor's message queue, such that their execution is serialised with respect to other messages processed by the actor.

Communicating Event Loops and Pervasive Computing

AmbientTalk adapts the communicating event loops model to cope with the hardware phenomena of pervasive computing environments, i.e. volatile connections and zero infrastructure (cf. Section 2.2).

Far References and Partial Failures. AmbientTalk's far references are by default resilient to network disconnections. When a network failure occurs, a far reference to a disconnected object starts buffering all messages sent to it. When the network partition is restored at a later point in time, the far reference flushes all accumulated messages to the remote object in the same order as they were originally sent. Hence, network failures have no immediate impact on the program's control flow, which is the desirable behaviour to cope with often temporary disconnections of pervasive computing environments. To cope with persistent failures, AmbientTalk uses a time-based failure handling mechanism founded on the notion of *leasing* (as in Jini [Wal99]). In its simplest expression, this mechanism enables an asynchronous message to be annotated with a timeout representing the time in which the message should be processed (including message send, remote method execution and resolution of the message's future).[1] If the timeout is reached the future is ruined with a `timeoutException`, as shown in the following example:

[1] A further explanation about AmbientTalk leases is beyond the scope of this thesis. We refer to [GBCV$^+$09] for a deep discussion on this subject.

```
when: messenger<-receiveText(aTextMessage)@Due(minutes(1)) becomes: { |ack|
  // message received successfully
} catch: TimeoutException using: { |e|
  // message timed out
}
```

Publish/subscribe Service Discovery. In most distributed systems, objects are exported to the network by means of a simple name, a universal unique identifier (UUID) in a name server or by a URL. In pervasive computing, though, name servers are impractical due to the limited infrastructure and the URL of a service may not be known to other actors [DVCM$^+$06]. Instead, AmbientTalk objects are exported and imported by means of *type tags* which are intensional descriptions denoting the services the objects provide. The following code shows how to export a service with the `MessengerType` type tag (using the `export:as:` construct):

```
deftype MessengerType;
def localMessenger := object: { ... };
export: localMessenger as: MessengerType;
```

AmbientTalk employs a decentralised service discovery protocol based on the publish/subscribe programming paradigm [EFGK03]. The type tag serves as a topic known to both publishers and subscribers [VME$^+$07]. A subscription takes the form of the registration of an event handler on a type tag (using the `whenever:discovered:` construct), which is triggered whenever an exported object under that tag has become available in the network:

```
deftype MessengerType;
whenever: MessengerType discovered: { |remoteMessenger| ... };
```

The `whenever:discovered:` function takes as arguments a type tag and an event handler represented as a closure which is executed whenever an exported object matching the type tag is encountered. The parameter of this closure is then bound to a far reference to the discovered object.

Discussion

AmbientTalk's communicating event loops comply with the requirements of distribution identified in Section 2.2 as follows:

Decentralised service discovery In AmbientTalk, objects are discovered using a decentralised publish/subscribe protocol based on type tags. Type tags are independent of any particular device address, catering for anonymous interactions among objects (space decoupling). Because each actor can both publish services and subscribe in order to be notified of services that become available in the network, no intermediary server is required.

Decoupled communication Event-driven communication between objects owned by different actors suits the volatile connections of pervasive computing environments. Asynchronous message passing and result handling decouple the communication in synchronisation, whereas far references and their means to buffer messages sent to a disconnected object (and to flush them when the object becomes available again), decouple communication in time.[2]

Connection-independent failure handling The time-based failure handling mechanism of Ambienttalk enables programs to abstract away from —possibly transient— network disconnections. A network failure during an asynchronous message execution is considered permanent (and signalled with an exception) only after a timeout is reached.

3.1.2 Event-driven Programming Styles

Distributed object-oriented programming languages featuring models for event-driven execution differ in the way they unify event-driven distribution and object-oriented primitives. In this section, we focus on the support that such languages provide for remote communication, i.e. their abstractions for remote message sending, message reception and result handling.[3] As we explain in Sections 3.2.3 and 3.3.3, these abstractions have a direct impact on the modularity of the systems. The differences between existing object communication models with event-driven semantics can be characterised as follows:

Uniform versus non-uniform message sending Programming languages can differ in the programming style they attribute to event-driven remote interactions. This style can be either *uniform* with respect to local object communication primitives, i.e. using the same operation for local and remote message sending, or *non-uniform*, introducing an explicit asynchronous *send* operator. The former typically implies an implicit transformation process to ensure remote interactions to still be sent and processed in an event-driven way.

Direct versus continuation-passing message and result handling In order to receive asynchronous messages and to asynchronously handle their results, event-

[2]AmbientTalk also achieves arity decoupling by means of a language feature known as *Ambient References*. We discuss this feature when reviewing the state of the art of group behaviour abstractions (cf. Section 3.3).

[3]A further study of the other aspects of distribution such as service discovery and parameter passing semantics is beyond the scope of this thesis, and for which we stick to the choices made by AmbientTalk [VME$^+$07].

driven programming language models can provide either a *direct* or *continuation-passing* programming style. A programming language with direct style aligns asynchronous message reception with method invocation (an asynchronous message is implicitly handled by invoking a method on the receiver object), and handles asynchronous return values using the standard request/reply pattern of object communication (the result of the asynchronous message execution is implicitly returned to the sender object). Conversely, we say that a programming language enforces a continuation-passing programming style if its communication primitives require the programmer to perform some sort of explicit continuation management. Typical manifestations of such continuations are callback functions, listener objects, token-passing continuations, `receive` and `when:becomes:catch:` statements to react to incoming messages.

In the object-oriented programming research literature we find a considerable number of approaches that advocate different combinations of the above alternatives of programming style, which we roughly classify as *implicit* versus *explicit* language support for event-driven distribution (uniform message sending and direct message and result handling for the former, non-uniform message sending and continuation-passing message and result handling for the latter). In what follows, we summarise the different positions about this language design concern and then discuss the proposed solutions of existing event-driven distributed programming languages.

Explicit Language Support

Explicit language support for distribution helps developers to better cope with the fundamental differences between local and remote object communication. As Eugster et al. state in [EGS00], distributed interactions are inherently unreliable due to the effects of network latency and failures, and thus can be hardly comparable to local interactions. In the same line, Waldo et al. [WWWK96] claim that network failures render unifying local and remote computing models impossible without making undesirable compromises. Treating remote interactions as local interactions may lead to a model which is essentially non-deterministic in the presence of *partial* network failures: When a component (machine, network link) fails it is usually expensive and hard, if not impossible to determine the state of the distributed system that caused the failure and the state of the system after the failure. On the contrary, treating all interactions as remote interactions, may introduce unnecessary semantics (to deal with network failures) for objects that are never used remotely, increasing the complexity of the overall system. Guerraoui and Fayad refer to this issue as the myth of distribution transparency [GF99], arguing for an alternative approach where developers can be aware of distribution and where the non essential aspects of this concern can be encapsulated inside language features with a well-defined interface.

For all that an object-oriented programming language can comply with the above requirement by featuring a dedicated event-driven execution model for distribution, explicit language abstractions are nevertheless needed to ease the understanding of where event-driven remote object communication occurs in the programs, and where control may yield back to the event loop.

Implicit Language Support

Recent distributed programming language models with implicit support for event-driven execution respond to the need to cope with a number of issues found in existing explicit programming approaches: *inversion of control* [HO06], *lost continuations* [FMM07] and *asynchrony contagion*.

Inversion of Control. Event-driven programming models often require the decomposition of a program into separated event handlers (e.g. the continuations for receiving messages and handling results). Because such handlers are often not invoked by the program itself but by an external source in the execution environment (e.g. the event loop), the control over the execution of the program's logic is said to be "inverted" [HO06]. Hohpe [Hoh06] relates this problem to the impossibility to use a call stack in the event-driven execution of programs: Event handlers are processed with no program-specific stack frames on the runtime stack. As such, developers have to manually encode the continuation and execution context among event handlers (a problem also referred as *stack ripping* [AHT$^+$02]).

Lost Continuations. In [FMM07], Fischer et al. explain that the lost continuation problem occurs when an event handler has an execution path in which its continuation is neither invoked nor passed along to the next continuation in the event chain. A lost continuation causes the intended sequential behaviour of the program to be broken, often producing errors that are difficult to trace to their source. Exceptions raised during the execution of an event handler can also be lost as they are not properly tackled by the subsequent continuation, potentially causing the program to crash or continue executing in undefined ways.

Asynchrony Contagion. Explicit distinctions between local and remote computations can have negative consequences on the evolvability of the programs. Any change of location or rebinding of an object (from local to remote or vice versa) has direct consequences on the clients of such object. Calls to the object's methods have to be manually adapted to conform to the programming model that corresponds to the service's current location (e.g. switching from synchronous to asynchronous message sending, and from direct to continuation-passing message and result handling). Often, a ripple effect occurs because the changes made to the immediate callers of the method themselves trigger cascading changes in their callers, and so on. We refer to this problem as *asynchrony contagion*.

The three problems above are consequences of the dedicated language abstractions for distribution and their interactions with standard object mechanisms such as message passing, method encapsulation and inheritance[4]. Hence, programming languages featur-

[4]The interaction between concurrent and distributed object models and inheritance have been thoroughly studied in an issue known as the *inheritance anomaly problem* [MY93]. We discuss details of such studies in Section 3.2.3.

ing implicit support for event-driven distribution aim to cope with such problems by enabling the definition of local and remote computations using the uniform and direct style of object-oriented programming, while internally still executing the remote computations in an event-driven fashion.

The Event Interleaving Hazard.

An additional concern we use to evaluate event-driven programming models is known as the *event interleaving hazard* [SRRB10]. In [AHT^{+}02], Adya et al. explain that event-driven execution models provide the developer with the ability to consider the handling of a single event as the unit of concurrent interleaving. This is a major benefit in comparison to other concurrency approaches such as multithreading systems, in which interleaving can occur even at the level of each basic instruction. Yet, event interleaving can still represent a hazard for the execution of the programs, especially for those cases that require mutual exclusion and cooperation among remote event loops [SRRB10]. For an event-driven program defined in terms of local and remote method invocations, the execution of a method can be interrupted as it asynchronously invokes one on a remote object. Because the continuation of the invoking method (i.e. an explicit callback continuation, or the rest of the method in models with implicit language support) is processed only after the result of the remote invocation is asynchronously returned, the possibility exists that the executing event loop processes other invocations in between. This may cause the method's continuation to be processed in a possibly modified execution environment, leading the complete program to an inconsistent state.

Note that the interleaving problem is not a consequence of the programming style of even-driven models but of their non-blocking execution semantics. Still, if those semantics are to be preserved because of its suitability to deal with network failures (cf. Section 3.1.1), the language should enable developers to at least identify the points in the methods where the interleaving can occur. In this sense, a programming language style with explicit abstractions for distribution seems more appropriate, although other alternatives have been proposed, as we describe in the next section.

Evaluation of Event-driven Programming Styles

We now evaluate existing programming languages with support for event-driven execution, focusing our attention mainly on the four issues identified in the previous sections: inversion of control, lost continuations, asynchrony contagion and the event interleaving hazard.

E and AmbientTalk Revisited. In Section 3.1.1 we described E's and AmbientTalk's explicit language support for distribution. Asynchronous messages are specified via the <- operator (different to the dot notation used for synchronous message passing). Message reception is represented as method invocation which enables objects to be completely unaware of whether their methods invoked synchronously or asynchronously. The futures representing the result of asynchronous messages are handled with explicit continuations registered via `when:becomes:catch:` forms. Regarding the problems of event-driven models with explicit language support, E and AmbientTalk behave as follows:

Inversion of control These languages alleviate the problem of inversion of control by modelling the continuations (to handle futures) as closures which can be defined "in line", i.e. using `when:becomes:catch:` forms inside the methods that issue the asynchronous invocations. This makes the flow of the programs easier to trace and also prevents developers from having to manually encode the execution environment of the continuations (as it is implicitly captured by the closures at the moment they are registered).

Lost continuations E and AmbientTalk avoid the problem of lost continuations by using futures as the implicit return address of asynchronous messages. A continuation defined for a future is bound to be processed, as the future is always resolved with a result or ruined with an exception.[5] In particular, in case that a network failure occurs that impedes the remote result (or exception) to be returned to the future, the time-based failure handling mechanism of AmbientTalk ensures that the future is ruined with a timeout exception. This is unless the future is explicitly configured to wait indefinitely for its resolution, despite any network failure [VME+07].

Asynchrony contagion E's and AmbientTalk's explicit language support for distribution poses a degree of difficulty for the evolvability of the programs. While local and remote message reception are uniformly handled through method invocations, local and remote message sending and result handling have completely different semantics. Although in these languages message sending can eventually be expressed using only the asynchronous `<-` operator, this would imply to also explicitly handling the resulting futures for each invocation. Explicit continuations can be avoided by an implicit resolution mechanism called *future or promise pipelining* [MES05]. However, this mechanism cannot be used in combination with standard control structures such as looping or conditional expressions.

Event interleaving hazard E and AmbientTalk deal with the problem of event interleaving by providing dedicated abstractions for asynchronous remote invocations and result handling. For all that these abstractions make clear where the interleaving can occur, developers have to manually ensure the consistent execution between the method that makes the asynchronous invocation, and its continuation defined in a `when:becomes:catch:` form.

Similar solutions to E and AmbientTalk can be found in other language models such as Twisted [Kin05] and Tweak Islands [Mil06], and thus the above analysis also applies to them.

Salsa. Salsa [VA01] is an actor-based programming language extension of Java for internet and mobile computing. Salsa differs from E and AmbientTalk in that it represents

[5] Of course, one can still forget to register a listener on the future.

actors as active objects which are defined akin to Java classes (but with a `behaviour` keyword instead). Salsa communication semantics are tightly coupled to the nature of the objects: Active objects can be accessed remotely and asynchronously, whereas passive (Java) objects are accessible only locally and via synchronous messages. Salsa provides a `<-` operator to denote asynchronous message sending, which can be associated with *continuation messages* (by means of the `@` symbol) to handle the messages' return values. This return value is passed to the continuation message in the form of a `token` variable, as shown in the following example:

```
remoteMessenger<-getUsername() @
  localMessenger<-addContact(remoteMessenger,token) @
    standardOutput<-println("contact added");
```

As with E and AmbientTalk, Salsa does not require extra language support to receive asynchronous messages; an asynchronous message is implicitly handled by invoking a method on the actor's behaviour.

Inversion of control Salsa enables writing distributed programs as sequences of continuation messages parameterised by a token containing the return value of the previous continuation. As such, programs can have an unambiguous control flow even in the presence of remote interactions. However, continuation messages are still processed in different execution environments (possibly, also in different hosts). Only the token and the arguments of the continuation message (including the receiver actor) are implicitly passed to the environment where the continuation message is executed. Moreover, the scope of the token is restricted exclusively to one continuation message. Reusing a token in another context such as another message is possible only by introducing callback functions (as shown by Dedecker in [Ded06]), which breaks the sequential semantics of the continuation chain.

Lost continuations Salsa avoids the lost continuation problem by enforcing asynchronous messages and their corresponding continuation messages to be explicitly connected via the `@` operator. Standard Java exception handlers can be defined for such continuation chains, although this language does not provide support for coping with the effects of network failures on remote interactions.

Asynchrony contagion Salsa's explicit support for remote message sending and result handling (different from the one used for object interactions) exhibits similar evolvability issues as E's and AmbientTalk's solution. However, Salsa's dual passive/active object model also entails a number of additional changes, e.g. for a passive object to be reachable remotely it needs to be redefined as an actor, enforcing the adaptation of existing local interactions with the object (from synchronous to asynchronous).

Event interleaving hazard Salsa does not provide any means to prevent inconsistencies due to event interleaving hazards, other than making explicit remote interactions via the `<-` asynchronous operator and the use of continuation messages.

A further limitation of Salsa is the use of centrally managed UUIDs, considered impractical in our context due to the zero infrastructure of pervasive computing networks (cf. Section 2.2).

ProActive. ProActive [BBC+06] is another extension to Java for distributed programming which combines the dual passive/active object model with futures. As in Salsa, only active objects are remotely accessible. ProActive uses Java's communication model for both passive and active objects. Yet, this approach ensures that messages sent to active objects are asynchronously processed. Synchronisation on the futures' value is handled by a mechanism known as *wait-by-necessity* [BBC+06]: Asynchronously invoked methods return a future which, when used in a subsequent computation, suspends the active object's execution process until the future is resolved with the result of the remote method invocation.

Inversion of control and lost continuations ProActive does not provide explicit continuation management which means that it does not suffer from the problems of inversion of control and lost continuations. Similar to AmbientTalk, network failures during the execution of a method invocation are notified by updating the corresponding future with a runtime exception. This exception can be handled with the standard Java `try-catch` forms.

Asynchrony contagion ProActive's uniform programming style for local and remote interactions provided by this model also facilitate the evolvability of programs. Yet that language suffers from the same problems as Salsa due to the distinction between active and passive objects (for a passive object to become remotely accessible it has to be redefined as active object).

Event interleaving hazard The blocking semantics of the wait-by-necessity future handling mechanism prevents the programs from event interleaving hazards; the body of a method is executed completely without releasing the active object's process, even in the presence of remote interactions. However, these blocking semantics are also the main drawback when considering the distribution requirement of decoupling communication (cf. Section 2.2).

Scala. Scala [HO09] is a programming language that provides both functional and object-oriented programming styles. Scala supports event-driven program execution by featuring an actor library based on the concurrency model of the Erlang [Eri11] functional language. Asynchronous message sends use a dedicated `!` operator while asynchronous message reception. Result handling require explicit `react` form. These forms contain a number of `case` statements which are selected to handle the messages through a pattern matching mechanism.

Inversion of control Scala copes with the problem of inversion of control by making actors "thread-less". Actors waiting for messages are not represented by a blocked thread but by a closure (using the `react` statement) that captures the rest of the actor's computation. The execution of the closure can be executed by the same thread that sends the asynchronous message to the actor, although it is not specified how this scheme extends to remote interactions. The control is returned when the closure terminates or blocks in a nested `react` form. As in the case of E's and AmbientTalk's `when:becomes:catch:` expressions, a `react` form can be defined "in line" and captures its execution environment.

Lost continuations Scala's actor library enforces developers to explicitly encode the continuations of asynchronous messages. The sender actor has to pass itself as an argument in the message and then define a `receive` form to handle the message's result. The actor receiving the message has to explicitly send the result to the sender actor. Hence, avoiding lost continuations during the communication between actors is completely the responsibility of the developer. A request/reply pattern has been proposed in [HO09] but it still requires developers to manually return the result (using a `reply` message), and to handle it at the sender actor in a `match` statement. Furthermore, in Scala developers have to manually encode support for network failures to avoid lost continuations, as in Salsa and ProActive.

Asynchrony contagion Scala supports local and remote computations in a different way, to send messages, receive messages and handle the results. In addition, it explicitly distinguishes between objects and actors, exhibiting the same problems as Salsa and ProActive.

Event interleaving hazard Scala actors avoids the event interleaving hazard by means of a `receive` form. Unlike the `react` form, the `receive` form is represented as a blocked thread. An actor suspended in a `receive` form waiting for the result of an asynchronous message, can resume only to process messages matching a pattern defined in such a `receive` form.

Kilim. Kilim [SM08] extends Java with an event-driven execution model for concurrent programming. Kilim features a dual passive/active object model in which asynchronous interactions are possible only among active objects (referred as actors), and can be written using the standard Java programming style (message sending, reception and result handling). Methods that may block waiting for the results of asynchronous method invocations in their body, have to be explicitly annotated with a `@pausable` qualifier. This allows Kilim to ensure an event-driven execution for such methods (transforming the method definition into continuation passing style). The `@pausable` qualifier is similar in spirit to checked exceptions in that both are "viral": all callers and overriding methods of the *pausable* method, must be marked `@pausable` as well.

Inversion of control and lost continuations Kilim's implicit event-driven execution semantics enables developers to preserve the direct and imperative programming style of Java. At the same time, the `@pausable` annotation makes the parts of the program (the methods) explicit for developers that are processed asynchronously.

Asynchrony contagion Kilim exhibits the evolvability problems of programming languages with a dual object model described so far in this section. However, its internal transformation process allows actor method invocations to look the same as object method invocations. Only the `@pausable` annotation is used to indicate the asynchronous execution of the method. The main benefit of this approach is that the developer can use a uniform model for passive and active object communication, while still ensuring event-driven execution for active objects.

Event interleaving hazard Kilim's event-driven execution semantics are non blocking which means that that approach may still be affected by event interleaving issues. Yet, the `@pausable` annotation helps developers to spot those places. If a method is not marked as `@pausable`, then developers know that there can be no event interleaving.

Thus far Kilim does not support distributed programming, although the authors argue that Kilim's communication model lends naturally to a seamless view of local and distributed message-passing object interactions [SM08].

TaskJava. TaskJava [FMM07] is another extension of Java that provides event-driven program execution by means of a cooperative multitasking mechanism based on coroutines, called *tasks*. A task is a class that encapsulates an independent unit of work which can be defined using the standard Java programming style, while internally being executed in an event-driven manner (also using a form of transformation to continuation-passing style, as in Kilim). TaskJava features an `async` qualifier to annotate the methods that should be executed asynchronously. Additionally, that model also provides a `wait` primitive that programs use to register their interest for one or more events. Conceptually, a `wait` call blocks until one of the requested events has occurred, therefore avoiding the use of explicit continuation callbacks. Internally, the execution of a task blocked in a `wait` call yields the control thread enabling the execution of other tasks.

Inversion of control and lost continuations As in Kilim, TaskJava avoids the problems of inversion of control and lost continuations by implicitly transforming methods annotated as `async`, so that they can be processed in an event-driven manner. Yet, thus far we observe that the model has only limited support for distribution concerns, such as network failures: Tasks can deal with issues produced by network failures while sending remote messages (expressed as I/O exceptions and handled with standard Java exception handlers), but they do not have any means to handle (transient or permanent) disconnections while awaiting events.

Asynchrony contagion TaskJava's uniform communication semantics avoids evolvability issues, although their explicit distinction between classes and tasks entail comparable problems to those of models with active and passive objects (in order for a method to be processed asynchronously it must be defined as part of a class that extends the `Task` class).

Event interleaving hazard The "conceptual" blocking execution semantics of tasks in TaskJava (by means of the `wait` primitive), together with the fact that there is no data sharing among tasks, prevent programs from problems caused by event interleaving.

Lua. Lua [Rob10] is a scripting language that features both explicit and implicit support for event-driven distributed programming, using a RPC-like communication model (called *ALua* [SRRB10]). The explicit support relies on a `rpc.async` primitive which associates remote procedure calls with their corresponding result handlers (to handle both the effective return value of the procedure and eventual exceptions). Both the remote call and the result handler are executed asynchronously by the Lua event loop. As in E and AmbientTalk, Lua represents a result handler as a closure which has access to the execution environment in which the remote call is issued. Lua's implicit programming support, on the other hand, features a `rpc.sync` primitive which allows the definition of direct remote calls and result handling, while internally keeping the same asynchronous execution described above. This is achieved by using a coroutine-based mechanism with implicit control transfer.

Lua separates the definition of a remote procedure call from its invocations in the program. The `rpc.sync` and `rpc.async` primitives define a remote procedure call indicating the process and name of the procedure (and the result handler in the case of `rpc.async`). These primitives return a function which can then be invoked with the necessary arguments. The main benefit of this separation is that programs preserve the same style of invocation for local and remote invocations.

Inversion of control Lua's explicit support alleviates the problems of inversion of control by enforcing developers to declare the continuation of remote procedure call (the function to handle the result) as part of the call. Also, this function can be defined in line, capturing the lexical environment the function that makes the remote call. Lua's implicit support provides a direct programming style that does not exhibit inversion of control.

Lost continuation Lua's explicit support alleviates the problems of lost continuations by enforcing developers to declare the continuation of a remote procedure call as part of the call. This means that asynchronously invoked procedures do not have to explicitly call the continuation. Again, Lua's implicit handling of continuations (using `rpc.sync`) also avoids lost continuations. However, both approaches have the same limitations to deal with network failures as in TaskJava (while waiting for the result of the remote procedure call).

Asynchrony contagion Lua's explicit support presents evolvability problems similar to those of E's and AmbientTalk's models (due to the explicit handling of asynchronous results). On the contrary, Lua's implicit support enables the definition of computations using a direct programming style.

Event interleaving hazard Lua's asynchronous execution of remote procedure calls does not prevent event interleaving. As their authors explain in [SRRB10], while a coroutine is blocked, other invocations may arrive and modify shared variables, leaving them in a state different from that which the blocked coroutine expects upon being resumed. To solve this problem Lua provides an additional library with synchronisation constraints (originally proposed by Frølund and Agha [Frø92, AFK+93]), which allows for the definition of conditional method executions.

JCoBox. JCoBox [SPH10] is an extension of Java that combines E's and AmbientTalk's communicating event loops with the standard direct and sequential programming style of Java, by means of a coroutine-like mechanism, also called *tasks* (as in TaskJava). However, unlike TaskJava, a task in JCoBox is implicitly created as a method and is asynchronously invoked on an object.

JCoBox actors (called *coboxes*) encapsulates objects, as in E and AmbientTalk, but also tasks. A task processing a method invocation has exclusive access to its containing actor's state, and has to release control explicitly to allow other tasks to become active (as in standard coroutines). Asynchronous method calls are expressed by using the `!` operation which returns a future as an immediate result. In that approach, the value of a future has to be explicitly claimed in two different ways: a blocking one (using a `get` method) which prevents other tasks from being executed in the same actor, and a cooperative one (using an `await` method) which gives up the control allowing the interleaved execution of other tasks. In the latter case, the future is resolved asynchronously. Finally, it is also possible to yield control using the `JCoBox.yield()` method.

Inversion of control and lost continuations By preserving the direct programming style of Java, JCoBox does not exhibit the problems of inversion of control and lost continuations. Further, exceptions occurred during the execution of asynchronous method calls are thrown when the future of the call is claimed (as in E and AmbientTalk) but which in addition enables the use of standard exception handlers. Yet, JCoBox has only rudimentary support for distribution (mostly based on Java RMI), and its documentation does not mention whether the support for network failures is aligned with the same exception mechanism.

Asynchrony contagion JCoBox features dedicated programming language abstractions for synchronous and asynchronous message sending but still more sensitive is the explicit claim of futures: For a local object to become remote the communication should be changed to explicitly claim the future, which also implies deciding for each future whether it is claimed blockingly or cooperatively.

Event interleaving hazard JCoBox's two ways to claim futures enable the developers to selectively deal with event interleaving. For preventing computations from compromising state changes, futures can be claimed in a blocking manner.

Model	Event-driven programming issues									
	Inversion of control	Lost continuations	Asynchrony contagion					Event interleaving	Synchronisation decoupling	Failure handling
			obj. def.	meth. def.	msg send	msg rec.	res. hand.			
E	✓ in-line closures	✓ implicit future resolution/ruin	✓	✓	✗	✓	✗	✗ non-blocking futures	✓ non-blocking event loops	✓ notified in futures
AmbientTalk	✓ in-line closures	✓ implicit future resolution/ruin	✓	✓	✗	✓	✗	✗ non-blocking futures	✓ non-blocking event loops	✓ based on time
Salsa	✗ loss of context in cont. msgs	✓ implicit call to cont. msg	✗	✓	✗	✓	✗	✗ non-blocking cont. msgs	✓ non-blocking event loops	✗ only via callbacks
ProActive	✓ direct style	✓ implicit future resolution/ruin	✗	✓	✓	✓	✓	✓ blocking futures	✗ blocking event loops	✓ notified in futures
Scala actors	✓ in-line `receive` forms	✗ manual cont. management	✗	✓	✗	✗	✗	✓ blocking `receive`	✓ thread-less actors	✗
Kilim	✓ `@pausable` methods	✓ internal CPS transformation	✗	✗	✓	✓	✓	✗ non-blocking continuations	✓ non-blocking event loops	✗
TaskJava	✓ `async` methods	✓ internal CPS transformation	✗	✗	✓	✓	✗	✓ blocking event subscription	✓ thread-less tasks	✗ only for msg send
Lua `rpc.async`	✓ in-line closures	✓ implicit call to cont. proced.	✓	✓	✗	✓	✗	✓ synch. constraints	✓ thread-less tasks	✗
Lua `rpc.sync`	✓ direct style	✓ internal CPS transformation	✓	✓	✗	✓	✓	✓ synch. constraints	✓ thread-less tasks	✗
JCoBox	✓ direct style	✓ implicit future resolution/ruin	✓	✓	✗	✓	✗	✓ blocking fut. claims	✓ cooperative fut. claims	✗

Table 3.1: Programming style for event-driven distributed communication

3.1.3 Event-driven Distributed Programming: Synthesis and Discussion

Table 3.1 summarises the evaluation of the programming styles of the models presented in the previous section:

Inversion of Control. To cope with the problem of inversion of control, i.e. to avoid fragmented control flows and manual stack ripping, existing programming models have proposed solution such as: *closures* (as in E, AmbientTalk, Scala actors and Lua `rpc.async`) which enable continuations to be defined in the same context where the remote invocation is done; *token-based continuation messages* (as in Salsa), which can also be associated with the remote invocations, but with some problems of loss of context (e.g. limited scope of tokens); and *implicit asynchronous continuation management* (as in ProActive, Kilim, TaskJava, Lua `rpc.sync` and JCoBox) which enables remote interactions to be written in a *direct style* (as in standard local object interactions). In the cases of Kilim and TaskJava, this implicit management process can be performed only on methods annotated with the `@pausable` and `async` qualifiers respectively.

Lost Continuations. To cope with the problem of lost continuations, i.e. to cover all the possible ways in which an asynchronous request may terminate, programming models have proposed the following solutions: To use *futures* as the implicit return address for remote calls (as in E, AmbientTalk, ProActive, JCoBox and optionally in Lua[6]); to encode such return addresses in the internal CPS transformation of the methods (as in Kilim, TaskJava and Lua `rpc.sync`); and to implicitly send the continuation message (as in Salsa) or call the continuation procedure (as in Lua `rpc.async`). Scala actors, on the contrary, require that the developers manually manage the continuations.

To deal with exceptions that occur during remote executions, future-based continuation models *ruin* the corresponding futures with the exceptions, which are then handled with the `when:becomes:catch` form in E and AmbientTalk, and with standard Java exception handlers in ProActive and JCoBox. Standard exception handlers of the languages are used also in the rest of the approaches. However, thus far we observe that only E, AmbientTalk and ProActive properly deal with exceptional conditions caused by network failures, as we explain at the end of this discussion.

Asynchrony Contagion. The asynchrony contagion is related to the degree of uniformity in the language support for local and remote interactions. Table 3.1 shows the following operations where such support can be observed:

Object definition Approaches such as Salsa, ProActive, Scala, TaskJava and Kilim explicitly distinguish objects that are locally and synchronously addressable (passive standard objects), from those that can be remotely and asynchronously addressable (active objects or tasks). The main drawback of this design is that it enforces developers to distinguish upfront the passive and active parts of the programs. Those

[6]Although only when using the `rpc.future` form.

approaches also make evolvability more difficult than necessary. E.g. to make the functionality of an existing object available for the network, developers have to convert the object into an actor or task and then adapt the existing local interactions with the object.

Method definition All the models included in this evaluation enable methods to contain asynchronous invocations in their body. In particular, Kilim and TaskJava require that developers annotate such methods so that they can be internally executed asynchronously without requiring explicit continuation management.

Message sending Language models such as E, AmbientTalk, Salsa, Scala, and JCoBox provide an explicit operator for remote message sending. Alternatively, ProActive's, Kilim's and TaskJava's implicit handling of remote invocations enable them to use the same syntax of uniform invocations. Lua obtains a similar result with the separation of the specification of remote procedure calls from their invocations in the programs: After a remote procedure call has been specified, it can be invoked using the same syntax for local procedure calls.

Message reception Most approaches handle local and remote message reception uniformly, i.e. by invoking a method on the receiver object. The only exception is Scala which requires the use of a `react` form.

Result handling ProActive, Kilim and Lua `rpc.sync` provide a direct programming style for handling asynchronous results. E, AmbientTalk, Salsa, Scala and Lua `rpc.async` require some sort of explicit continuation management (closures, token-based continuation). JCoBox and TaskJava also enable the direct handling of asynchronous results but only after having explicitly claimed the futures (as in JCoBox) or registered for an result reception event (as in TaskJava).

The Event Interleaving Hazard. To cope with event interleaving hazards, programming languages (re)introduce blocking semantics, to claim futures as in ProActive and in JCoBox (using `get`), or to wait for events as in Scala and TaskJava. Alternatively, Lua has proposed the use of higher-level language patterns such as dynamic monitors and synchronisers which allow for a selective reception of events.

Note that all the approaches, with the exception of ProActive, comply with the synchronisation decoupling requirement for communication, described in Section 2.2. While E, AmbientTalk, Salsa and Kilim feature non-blocking execution semantics of the actors' event loop, Scala makes actors thread-less which is in spirit similar to the coroutine-based solutions of TaskJava, Lua and JCoBox (the three featuring thread-less cooperative tasks). Notably, in the case of JCoBox the non-blocking execution semantics are optional, obtained only by cooperatively claiming futures (using the `await` method).

Finally, regarding the support that the different language models provide to cope with network failures during the communication, we observe that thus far only E, AmbientTalk and ProActive have dedicated means to deal with network disconnections. Additionally, AmbientTalk includes a time-based failure handling support which makes remote invocations resilient to transient disconnections.

Other Approaches of Event-driven Distributed Programming

Several other programming paradigms have been proposed that deal with distribution concerns, but the evaluation criteria for object-oriented programming style we focus on this thesis can hardly apply to their solutions:

Publish/subscribe and Tuple spaces There is an important amount of research on publish/subscribe [Eug07] and tuple spaces [MPR06] middleware. In this middleware, communication occurs in the form of services subscribing to the events they are interested in, and services anonymously and asynchronously publishing events. These events are notified to clients with matching subscriptions by sending them the data associated with these events. Hence, remote communication is highly decoupled in terms of space, time and arity, and also in synchronisation in some more recent variations of these models [EGS00]. At the programming level, these models imply the use of generic interfaces to publish and register for events enforcing distributed programs to be expressed essentially as sequences of callbacks. Other than that, in this chapter we focus on models that preserve interactions based on standard object-specific interfaces, and with some built-in support for correlating events, as in bidirectional reply/request schemes afforded by message passing. Yet, those approaches have significantly influenced several aspects of our work, e.g. as the publish/subscribe discovery mechanism which we inherit from AmbientTalk (cf. Section 3.1.1), and their space and arity decoupling property which allows for group communication, discussed later in Section 3.3.

Reactive programming Reactive programming is a functional paradigm that emphasises on the data dependencies and change propagation of the programs [CK06]. That approach provides explicit means to represent values as continuously changing over time, and functions depending on such values that automatically re-evaluate every time the values change. This is unlike conventional imperative programming where changing values can be captured only indirectly and discreetly, through state and mutations. Reactive programming enables developers to write event-driven systems using a direct programming style (without explicit callbacks). Event streams can be directly defined as changing values (called *reactive values* [MGB$^+$09]) while actions that depend on the events can be modelled as function calls with the reactive values as arguments. Nevertheless, that solution involves an important switch of mindset in the developers that set this paradigm apart from the object-oriented programming models we include in this chapter. Reactive programs are defined not in terms of imperative message-passing communication, but in terms of reactions that are automatically recomputed when the other reactions they depend on have been modified.

Aspect-oriented programming Several extensions to aspect-oriented programming models have been proposed to deal with distribution [NCT04, PSDF01, NSV$^+$06]. The central idea is to enable developers to define *distributed* aspects, in which specific points in a program executed at certain host (e.g. method invocations), can implicitly trigger some action in another host.[7] This support enables the transparent handling of method invocations on remote objects. However, most of these models make developers to deal with RMI-like code at the aspect level (to forward invocations and to handle results and network failures), which is far from trivial [NSV+06]. Furthermore, most of such approaches provide only synchronous execution semantics. Benavides et al. [NSV+06] define a model for asynchronous aspect execution with blocking futures as results (as in ProActive). Yet, such futures must be handled exclusively at the aspect level (to preserve aspects' obliviousness property). It still implies non-trivial definitions and combinations of aspects. We further evaluate this solution when discussing about programming models for distributed group behaviour (cf. Section 3.3).

3.2 Modelling Context Dependency

The lack of linguistic support for encoding context dependency forces programmers to scatter these dependencies throughout application code in the form of conditional statements. In object-oriented programming, ad hoc polymorphism alleviates this problem by means of dynamic method dispatch. That mechanism enables behavioural variations based on a *receiver* argument, or potentially multiple arguments, in languages based on *generic functions* (also known as *multimethods*). Methods and their overriding relationships are defined along inheritance chains of classes or objects. Although an improvement, object-oriented dispatch has been found to be limiting in many situations. Contemporary computing paradigms such as ambient-oriented programming require means to make the behaviour of programs depend on arbitrary information available in their context. Several programming models have acknowledged this issue, extending OOP to model the programs' context-dependent behavioural variations.

In [GMH07], Gonzalez presents an extensive list of diverse programming paradigms that can be used for modelling context dependency in the programs. In this dissertation, we focus on the solutions of context-oriented programming language models, i.e. models with dedicated language abstractions to express context-dependent behaviour. We evaluate such approaches according to the requirements of context dependency presented in Section 2.3. In this evaluation we use the mapping of the requirements for context-dependent behaviour to object-oriented features, introduced by Hirschfeld et al. [HCN08] and Gonzalez [GMH07]:

Modularity. A context-oriented programming model should enable the definition of more than one behavioural variation for a given event. In object orientation, such events typically correspond to method invocations. The behavioural variations are defined in terms of modules provided by the underlying programming model, such as methods or classes. Each variation should be specialised on a particular context situation.

[7]Or in several hosts, as we discuss in Section 3.3.

Dynamic Selection. The selection of the behavioural variations must be *late bound*: Context-dependent adaptations should be decided at runtime [GMH07]. Dynamic behaviour selection is one of the flagships of object orientation, usually referred to as dynamic dispatch. A context-oriented programming model should extend that mechanism (based on inheritance hierarchies) to enable the behaviour selection according to the current context.

Consistent Composition. A context-oriented programming model should enable the consistent composition between the behavioural variations required for a certain context, and the base (context-independent) system behaviour. In object orientation, behaviour composition is usually realised through various forms of inheritance. However, other composition techniques have been proposed to complement or replace that mechanism.

Restricted Scope. Context-dependent behavioural variations should have an unambiguous scope of action. This scope can de restricted *lexically* or *dynamically*. Lexically-scoped behavioural variations affect a specific part of the program, e.g. a method definition. Dynamically-scoped variations affect a dynamic extent of the program execution, e.g. the execution of a method, including all the code that the method calls directly or indirectly.

3.2.1 Evaluation of Context-oriented Programming Models

This section presents the evaluation of context-oriented programming language models with respect to the above requirements.

ContextL. ContextL [CH05] was the first object-oriented language model that explicitly supports context-oriented programming. It extends CLOS with the notion of *layers* which are first-class entities encapsulating partial class definitions. The layers represent the different contexts in which a program execution can occur. The partial definitions correspond to the behaviour expected from the classes in those different contexts. Layers must be explicitly *activated*, i.e. selected for execution. A `with-active-layers` language construct is provided to enable the layers to be activated anywhere in the program, and at any point in the program execution. The `with-active-layers` construct delimits a *dynamic scope* for the activation of layers. It receives as arguments a list of layers and a body containing the computations that should be executed using those layers. The layers are active only during the execution of the body, including all the code that the body calls directly or indirectly. Likewise, layers can be explicitly deactivated for a dynamic execution extent, using the `with-inactive-layers` construct.

Modularity ContextL provides two levels of modularity. First, this model allows context-dependent behaviour to be cleanly separated into partial class definitions. Second, that model can group the partial definitions (of different classes) that are required for the same context in layers.

Dynamic selection In ContextL, layers can be activated dynamically at runtime. Yet, no explicit abstractions are provided to encode the context-dependent selection of layers.

Consistent composition In ContextL, layers can be dynamically composed. The composition order between layers depends on the order in which they are activated. A `call-next-layered-method` construct enables layers to reuse behaviour from less specific layers. Additionally, ContextL supports a number of composition rules [CD08] to represent advanced relationships between layers (e.g. exclusion, conditional dependency, etc.).

Restricted scope The scope within which layers are activated can be controlled explicitly in the programs by means of `with-active-layers` forms. This form delimits the dynamic scope for layer activation to the execution of its body. Layer activations can be further constrained by nesting the `with-inactive-layers` forms.

Several other implementations of ContextL's layer-based model for context dependency have been defined for other languages (ContextJ for Java, ContextS for SmallTalk, ContextR for Ruby, to name a few). A similar evaluation can be done for those variations, as described in [AHH$^+$09].

PyContext. PyContext [vLDN07] is a framework for context-oriented programming in Python [Fou11]. PyContext is a variation of the layer-based model of ContextL. It supports implicit layer activations and dynamic variables as means to access context-dependent state. PyContext alleviates ContextL's modularity issue of explicit activations: If a context condition that should trigger a layer activation can become true at any time in the program (and if it is necessary that the program reacts to the context change quickly), the check for the change of condition may need to be added in many places. Eventually, the code to activate the layers becomes scattered and larger than the actual contextual behaviour. To solve this problem, PyContext layers provide an `active` method which contains a condition that is implicitly evaluated when a layered method is called. The partial definition of a layer is used only if its `active` method evaluates to true. Yet, that implicit activation mechanism also prevents developers from defining the priority order between layers (no details about activation order is provided in the documentation of that model).

PyContext's dynamic variables have been defined to avoid passing context information as parameters in the method invocations. Such variables are represented as globally accessible objects, whose values are dynamically determined within the dynamic extent of a layer activation. Previous values are shadowed and available again after leaving the current extent.

Modularity PyContext keeps the two levels of modularity provided by ContextL: layers and layered methods. In addition, PyContext also abstracts the layer activation code away from the programs. It is contained in each layer in the form of an `active` method, which describes the context condition for layer activation. The main disadvantage of this approach is that it prevents developers from explicitly defining the activation order between the layers.

Dynamic selection Layer activation occurs dynamically in response to layered method invocations. This is an implicit process. A layer is taken into account for execution if its activation condition is accomplished (if the layer's `active` method returns true).

Consistent composition PyContext follows the same semantics of layer composition as ContextL. The partial definitions for a layered method can be aggregated, and a `proceed` construct is provided to reuse behaviour.

Restricted scope In PyContext, the scope of an implicit layer activation is restricted by the execution of a layered method. No support is provided to deal with concurrent invocations.

Ambience. Ambience [GMH07] is a programming language with dedicated means for explicit context representation and context-dependent behaviour definition. That model proposes a *subjective* dispatching mechanism in which the execution of a message depends not only on its arguments but also on the context in which it occurs. The context is represented by one or more objects structured in a delegation hierarchy, which is implicitly passed as the first argument of every message. Correspondingly, an implicit parameter is added to every method definition, using the current context as specialiser. Methods can thus be specialised on their context of definition. Alternatively, developers can explicitly specify the context for which a method should be specialised. For this purpose Ambience provides an `in-context` construct which receives as arguments a context object and the code (e.g. a method definition) to be evaluated using that context.

Ambience's subjective dispatch ensures that there is always only one multimethod which is the most specific according to the current context. Such a mechanism is driven by the delegation hierarchies in which their arguments are involved, including the hierarchy of the (implicit or explicit) context argument. Access to less specific methods are possible by means of `resend` calls.

Modularity In Ambience, context-dependent behavioural variation is modelled as multimethods. Each multimethod is specialised on a context situation represented as a list of objects. Such a context can be specified both explicitly by developers or implicitly by the Ambience interpreter. Unlike ContextL and PyContext, Ambience does not have an explicit construct to group context-dependent behaviour. Note, however, that such groups can still be implicitly created by specialising different multimethods on the same context object.

Dynamic selection Ambience's subjective dispatch enables the method selection and combination to depend not only on the received arguments but also on the message sender's context.

Consistent composition In Ambience, the composition of multimethods is driven by the delegation hierarchies of the arguments received in the invocation. Complementary techniques are used to resolve ambiguities in the process of selecting the most specific multimethod according to the current context.[8]

Restricted scope Ambience enables programs to be aware of the dynamic context changes and to react to them adapting their behaviour accordingly. To ensure this, the scope of action of context-dependent behavioural variations is strictly limited to the execution of one multimethod. Ambience (re)evaluates the context for each method invocation, and upon each super call inside a multimethod's body.

The CDR Model. The context-dependent role (CDR) model [VED+07] is an extension of the AmbientTalk programming language for the development of context-dependent behaviour in mobile distributed systems.[9] In this model, services are represented as actors provided with a set of context-dependent behavioural variations organised in a delegation hierarchy. Each variation is represented as a role that the actor can adopt to respond to a message. The hierarchical delegation structure, originally presented in [Lie86], enables the adaptations to extend the default behaviour of the actor —placed at the root of the hierarchy— or any other more general adaptation situated higher up the delegation tree. The delegation semantics ensure a consistent interaction between the role objects [Lie86]. These objects cannot receive messages directly. Instead, actors receive messages and respond to them by first selecting the appropriate role and then executing the corresponding method in the adaptation object of that role.

The selection of which role an actor has to adopt to respond to a message is a decision made autonomously by the actor receiving the message. This decision is based on the context of both the message sender and receiver. This means that the sender cannot indicate the role required for the message execution (as in traditional role-based models [BD96]). Instead, it should pass part of its own context information along with the message. The role selection process is supported by a dedicated logic reasoning engine within the actor, called the *context-dependent role selector*.

Modularity Behavioural adaptations in the CDR model are encapsulated in role objects. Interaction between adaptations are regulated by the semantics of the delegation mechanism. The context reasoning is also concentrated in a single entity called the context-dependent role selector.

Dynamic selection Dynamic adaptations of behaviour occur transparently for the programmer as a result of the selection of a context-dependent role. This role indicates the behavioural adaptation (object) to which the actor has to address the message.

Consistent composition The composition of behavioural adaptations is defined by the delegation hierarchy. This hierarchy is a flexible structure in which the adaptations can specialise and consistently share behaviour.

[8] As the ordering algorithm C3 [BCH+96] to select the current most specific context for an invocation.

[9] The CDR model was the first result of our research and the direct precursor of the model presented in this dissertation (cf. Chapter 5).

Restricted scope Behavioural adaptations are active only within the scope of a message execution. This means that an actor adopts a certain role only to process a single message. Because actors can process only one message at the time, there are no interaction issues between adaptations required for different messages.

Context-aware Aspects. Tanter et al. [TGDB06] propose a reflective framework for Context-aware Aspects, i.e. aspects whose behaviour depends on the context. That approach provides language constructs to scope aspects to specific execution contexts, and to allow an aspect advice to be parameterised by context. That framework extends the traditional notion of context in aspect-oriented programming, which is limited to information directly associated to jointpoints, e.g. arguments of messages or control flow. In that approach, developers can reify any information about the state of a program, in dedicated context objects (in a similar way to Ambience). Special attention is put on enabling aspects to depend on "past" contexts (e.g. the context in which a certain object was created). For this, context information can be *snapshot* so that it is accessible at any point in the program execution.

Modularity The framework expresses behavioural variations in the form of aspect advice. Context reasoning is also encapsulated in context objects and pointcut definitions.

Dynamic selection The applicability of an aspect is determined by the evaluation of its pointcut, which in this framework can be constrained to a specific context situation.

Consistent composition The framework enables the composition between the context-aware aspect and the base behaviour. However, it is not specified how to ensure the consistent composition between several applicable context-aware aspects. There is also no explicit support to define context-dependent priorities between the aspects.

Restricted scope Context-aware aspects are lexically scoped. They are applied only to the parts of the programs, the join points, that adhere to the pointcut definitions. These definitions are evaluated each time a joint point is executed. The framework does not provide any support for concurrency.

JCop. JCop [AHM+10] is an extension of ContextL's layer-based model of context-oriented programming for the definition of dynamic behavioural variations in event-driven systems. The approach uses aspect-oriented programming to enable the specification of layer composition (activations) to span several event handlers. This prevents the programs from having composition statements scattered over several locations. JCop introduces the notion of *context types* which contain a declarative composition statement similar to a pointcut-advice construct. Context types can explicitly specify all the methods that should be included in the scope of a layer activation. Alternatively, context types can define guards for some properties in the system, and make activations depend on the changes of state of such properties. Context types are re-evaluated for every execution of the methods they are bound to.

Modularity JCop inherits ContextL's two levels of modularity of context-dependent behaviour, partial method definitions and layers. Additionally, JCop encapsulates layer composition statements (conditions and layers to be activated) in context types.

Dynamic selection Layer selection is a dynamic process based on the evaluation of predicates included in the context types. These predicates are evaluated either when a layered method is invoked, or upon a change in the state of the system.

Consistent composition JCop's context types allow for declarative composition of layers. Using context types, developers can specify the layers that should be activated (and deactivated), and the order of activation. Furthermore, any expression returning a list of layers can be used, so layer compositions can be dynamically computed. Yet, no language support is provided to determine the consistent composition between the layers.

Restricted scope The scope of a layer activation is determined by the context type. In addition to the original dynamic scope of layers defined in ContextL, in JCop a layer can also be applied to several independent control flows.

Filtered Dispatch. Filtered Dispatch [CHVD08] is a generalisation of the generic function-based method dispatch mechanism of CLOS. It draws inspiration from a programming model called specialisation-oriented programming [NR08]. That model enables methods to specialise their arguments on *filter* expressions. Such filters map arguments to representatives of equivalence classes. Filtered arguments are then used in place of the original arguments for method selection and combination. Still, the chosen methods are invoked using the original arguments. This explicit separation between method selection and execution resembles the *lookup* ∘ *apply* decomposition of method dispatch investigated by Malenfant et al. [MDC96]. In Filtered Dispatch, *lookup* receives the filtered arguments, and *apply* receives the unfiltered (original) ones.

The priorities between the filters are specified in a per-generic-function basis: *Filtered* generic functions contain not only the methods with a common name and argument structure (as in standard generic function models [BDG+88, CLCM00]), but also the predicates on which such methods can be specialised. To determine method specificity, methods using different filters are compared using the order in which the respective filter specifications appear in the generic function definition.

Filtered Dispatch has been originally conceived for the implementation of a Lisp interpreter. An alternative use case, though, which concerns our goal on the modelling of context-dependent behaviour, is to use filters as predicates on the runtime state of the arguments.

Modularity Filtered Dispatch allows behavioural variations to be modelled as methods with the filters representing the context conditions for their applicability. As in the case of Ambience, "layers" of adaptations can be implicitly defined by using the same filter in several generic functions.

Dynamic selection The selection of context-dependent behaviour is dynamically performed as part of the method dispatch. An extra step is added to the dispatch mechanism in order to evaluate the filters on the received arguments and thus determine the applicability of the methods. The selected methods are sorted according to the order in which the filters are specified in the generic function.

Consistent composition A consistent composition between the selected methods is manually ensured by developers through the order of definition of the filters in generic functions. Code reuse is achieved by enabling "super calls" between filtered methods (using a `call-next-method` construct, as in CLOS).

Restricted scope The filter-based selection and ordering of methods is calculated for each method invocation. Changes in the state of the arguments while processing a method invocation are taken into account only for future invocations (unlike Ambience). This approach does not specify any semantics for dealing with concurrent invocations.

Predicate Dispatch. The use of predicates for method dispatch has been explored before in a model called Predicate Dispatch [EKC98, MFRW09], which in turn drew inspiration from predicate classes [Cha93] and mode classes [Tai93]. However, in that model the method overriding relationship is based on the logical implications between predicates. Thus, the set of method predicates must be restricted to a well-chosen subset that can be statically analysed. This leads to a viable approach, but can be limiting in some circumstances, as users cannot extend Predicate Dispatch with their own arbitrary predicates (as those required for context-dependent behaviour).[10] Furthermore, Predicate Dispatch does not allow the composition of the selected methods (only the most specific is applied to the arguments).

Filtered Dispatch alleviates these restrictions by enabling users to establish a priority order between logically unrelated predicates (in a per-generic-function basis), and to potentially access all the selected methods (by means of super calls). Yet, there are still some properties of predicates in Predicate Dispatch that are not present in filters. Firstly, even though many filters can be defined for a given generic function in Filtered Dispatch, corresponding methods can use only one of those filters at a time. As a consequence, each possible combination of the filters that could prove useful needs to be anticipated and encoded as an additional filter in the generic function. Secondly, filtered expressions are parameterised exclusively on the argument they filter; they cannot depend on the value of other arguments of the method. This restriction renders Filtered Dispatch less amenable to express context adaptations, because the conditions for applicability (the predicates) cannot harness all available contextual information.

[10]Predicates without logical implications can still be added but they are treated as black boxes and the overriding relationship between two syntactically different expressions is considered ambiguous [EKC98].

Model	Modularity			Dynamic selection	Consistent composition	Restricted scope
	partial behaviour definition	groups of partial definitions	behaviour selection			
ContextL	✔ layered methods	✔ layers	✘ scattered layer activation	✔ based on explicit layer activation	✔ driven by activation order and reflective composition rules	✔ dynamic scope of method execution
PyContext	✔ layered methods	✔ layers	✔ active methods	✔ based on implicit layer activation	✔ driven by activation order	✔ method execution
CDR model	✔ role objects	✘	✔ role selector	✔ logic reasoning using sender's and receiver's ctx	✔ driven by delegation semantics of role hierarchy	✔ method execution
Ambience	✔ multimethods	✔ using same context object as specialiser	✘ no support to select context objects	✔ based on context objects	✔ driven by context hierarchy	✔ one multimethod execution
Context-aware aspects	✔ aspect advice	✘	✔ aspect pointcuts	✔ based on evaluation of pointcuts	✘ no composition rules specified	✔ jointpoints execution
JCop	✔ layered methods	✔ layers	✔ aspect pointcuts	✔ based on evaluation of pointcuts	✔ driven by activation order	✔ dynamic scope of jointpoints execution
Filtered dispatch	✔ multimethods	✔ using same filter as specialiser	✔ filters	✔ based on evaluation of filters	✔ priority order of filters per generic function	✔ multimethod execution
Predicated dispatch	✔ multimethods	✔ using same predicate as specialiser	✔ predicates	✘ based on logical implication (statically checked)	✘ no composition allowed	✔ multimethod execution

Table 3.2: Programming style for context-dependent behaviour

3.2.2 Context Dependency: Synthesis and Discussion

Table 3.2 summarises our study of the language features provided by today's models for explicitly supporting context-dependent behaviour:

Modularity. The solutions provided for modularity of context-dependent behaviour can be classified according to three different concerns: modular partial behaviour definitions, modular groups of partial behaviour definitions, and modular behaviour selection.

Modular partial behaviour definitions In the approaches included in this evaluation, context-dependent behaviour is modelled in the form of partial method definitions. These definitions can correspond to layered methods, multimethods, aspect's advice or role objects.

Modular groups of partial definitions Additionally, some models provide explicit or implicit means to group partial behaviour definitions of different classes, that are required for the same context. Approaches such as ContextL, PyContext and JCop provide a dedicated *layer* abstraction for this purpose. Alternatively, methods can be implicitly grouped by defining them with the same context-dependent specialiser (e.g. the same context object, filter and predicate in Ambience, Filtered Dispatch, and Predicate Dispatch respectively). An important drawback of such implicit approaches, however, is that it requires that the groups of behaviour are stateless. Because a group of behaviours does not have a first-class representation, it cannot encapsulate data. This leads to entanglements between the program's default state and the state required for context-specific groups of methods. Furthermore, it makes it more difficult to model variables that may have different values according to the context [CH05, Tan08].

Modular behaviour selection Finally, the code required to select the behaviour according to the context, can also be modularised. This can be observed in the *active* methods of PyContext which encapsulate the context conditions under which a layer should be activated. A similar role is played by pointcuts in Context-aware Aspects and JCop, filters in Filtered Dispatch, and predicates in Predicate Dispatch. In the CDR model, the rules required to determine the roles are also contained in one entity, the role selector. Ambience bases its support for behaviour selection in context objects which are used as implicit arguments of the multimethods. However, that approach does not provide any support to select the context objects that should be used at a particular situation. ContextL exhibits a similar issue. That approach enables relating methods to layers but it does not provide any dedicated support to define the context conditions under which a layer should be activated.

Dynamic Selection. We can observe that almost all the models have means to dynamically select the behaviour for a method invocation according to the context. This selection process is closely related to the method dispatch mechanism of the languages. They provide diverse means to enable developers to define the dependency between method definitions and context information. Some approaches enable dispatching methods on entities representing particular context situations (layers or context objects). Others enable the method dispatch to be based on filters or predicates on the received arguments' states, and possibly on any other information accessible in the execution environment. This is the case for Filtered Dispatch and Predicate Dispatch. However, this also applies for aspect-based and role-based solutions. The applicability of Context-aware Aspects depends on the evaluation of their pointcuts which can contain arbitrary conditions about the program's context. Similarly, the CDR model selects the role object based on the context of the message's sender and receiver. Finally, the selection of context-dependent behaviour can also be explicit in the programs. This is the case of ContextL and its constructs for layer activation and deactivation. The main drawback of this explicit mechanism, though, is that it can potentially lead to scattered code.

Consistent Composition. In this evaluation, we observe that consistently composing context-dependent behaviour corresponds mainly to ensuring an unambiguous combination between the different partial definitions required to process a method execution. For this, most of context-oriented programming models provide means to define priorities between the partial definitions. There is always a definition that is the most specific. Less specific definitions can be accessed via *super calls*. In layer-based approaches, the priority between partial definitions is given by the order of activation of the layers. This activation order can be dynamically computed (as in JCop), or assisted with composition rules representing advanced relationships between layers (as in ContextL). In the CDR model, the composition is driven by the delegation semantics of the role hierarchy. A similar situation occurs in Ambience where the combination of multimethods is driven by the delegation hierarchy of the multimethods' implicit context argument (and possibly also by the delegation hierarchies of the other arguments). In Filtered Dispatch, method combination is based on the priorities between the filters defined in a per-generic function basis. Context-aware Aspects do not specify any support for composition of context-dependent behaviour. In Predicate Dispatch, logical implication between predicates defines the overriding relationship between corresponding methods. However, that approach has limited facilities to resolve method combination ambiguities since the logical implications between predicates cannot be decided in the general case. Thus, the set of method predicates must be restricted to a well-chosen subset and thus can be statically analysed. This leads to a viable approach, but can be limiting in some circumstances, since users cannot extend Predicate Dispatch with their own arbitrary predicates in a straightforward way.

Restricted Scope. In most of the above models the scope of the activation of context-dependent behaviour is limited to the execution of a method invocation. A special case is ContextL which keeps the layer for a dynamic extent restricted by the `with-active-layer` form (and possibly by nested `without-active-layer` forms). Another special case are the aspect-based models which extend the scope of the behaviour activation to all join points in the program captured by the aspect poincuts. In particular, JCop combines the layers with aspects enabling the activations to have a dynamic extent at all join points. At the other extreme, there is Ambience which scopes the behaviour activation to only one multimethod execution. For all the other approaches, an activation is kept until the end of the processing of the method invocation. This might include the execution of a chain of methods accessible via super calls. Ambience, however, will re-evaluate the context at each super call to select the next applicable method. That approach enables the behaviour to keep consistent with the context changes. Yet, in some cases changing the composition of applicable methods during a method execution might lead to inconsistencies.

Finally, thus far we observe that with the exception of the CDR model, none of those approaches has been developed to deal with distribution issues. As we explain in the next section, these issues affect the definition and selection of context-dependent behaviour significantly.

Other Approaches for Context Dependency

There is a huge amount of research on frameworks that support the development and deployment of context-aware systems like WildCAT [DL05], ContextToolkit [SDA99] or Java Context Awareness Framework [Bar05]. The aim of these frameworks is to provide a generic programming infrastructure that deals with common functionalities like uniform interfaces to access sensor data, event-based system to signal context changes, and reasoning mechanism to aggregate context information. Context-aware frameworks are useful for both pro-active and reactive systems. In the former case, callback methods are used, as part of an event-driven system, to automatically invoke some behaviour in response to relevant context changes. Additionally, framework solutions also provide the ability to query for actual context information such that reactive systems can adopt their behaviour accordingly. Developers have to rely on traditional dispatching constructs like conditional statements or polymorphism to establish the behavioural adaptation. As soon as context-dependent behaviour appears to be the rule rather than the exception, these language constructs become unmanageable. We therefore argue that context-aware frameworks and a programming model for context-oriented programming are actually complementary. Whereas the framework solutions provide the required functionalities to develop context-aware systems, a context-oriented programming model focuses on how context-dependent adaptations can be decently modelled inside of software systems. The synergy between both proposals supports the development of pervasive systems.

3.2.3 Context Dependency in Ambient-oriented Programming

We now discuss the issues that show up when combining programming models for context-dependent behaviour and for distribution. In this discussion, we use the ambient-oriented programming paradigm and its manifestation in AmbientTalk's communicating event loops, as the basis of distribution. Yet, the present analysis can also apply to other event-driven programming approaches (cf. Section 3.1.2). In this chapter, we have presented context dependency essentially as the need for adapting the program's behaviour to respond to communication events (i.e. method invocations) according to the context. Thus, we focus the following discussion on the interactions between AmbientTalk's event-driven communication model, and the language support required for context-dependent behaviour.[11] We distinguish three aspects in this combination: the propagation of handling asynchronous computations, the propagation of network failure handling, and the activation scope of event-driven behaviour.

Propagation of Asynchrony

The interactions between synchronisation and behaviour specifications have long been studied in the object-oriented programming field. Briot et al. [BGL98] observe that synchronisation is particularly difficult to combine with standard object techniques for behaviour encapsulation and reuse (e.g. classes and inheritance). This is due to the high interdependency among synchronisation conditions required for different methods and classes, and the various uses of inheritance (to inherit variables, methods and synchronisations) which may conflict with each other. This limitation is known as the *inheritance anomaly problem* [MY93].

AmbientTalk's communicating event loop model avoids this limitation by explicitly separating synchronisation from behaviour definition. Synchronisation is handled strictly at the actor level whereas behaviour is specified exclusively in terms of objects (cf. Section 3.1.1). Therefore, techniques such as object inheritance have no influence on the synchronisation semantics of the programs. Still, some issues can occur due to the use of dedicated syntax for handling asynchronous remote method invocations. In particular, this is due to the cascading effect that such special handling causes in the interacting objects and their clients, the problem previously identified as asynchrony contagion (cf. Section 3.1.2). In this section, we discuss the consequences of this problem for the dynamic method dispatch and inheritance-based composition mechanisms, required for context-dependent behaviour. We refer to this concern as the *propagation of asynchrony.*

Event-driven Selection of Behaviour. In Section 2.5, we have discussed that in a distributed setting, the selection of behaviour may require the evaluation of remote context conditions. For this, the selection process should include remote interactions which AmbientTalk (and all the other event-driven programming models) represents as asynchronous remote method invocations (cf. Section 3.1.1). The result of asynchronous invocations is also asynchronously handled, in most cases, by means of explicit continuations. The problem occurs when trying to combine such explicit handling of asynchronous interactions with the implicit selection process promoted by models for context-dependent behaviour. Most of these models modularise the context conditions (in activation functions, filters, predicate functions or pointcuts) which are evaluated as part of the method dispatch mechanism (cf. Section 3.2.1). However, current implementations of this mechanism will fail if the evaluation of context conditions does not return an immediate result. Context conditions leading to asynchronous results force the method dispatch mechanism to be modelled as part of the continuations defined to handle such results. Yet, method dispatch is hidden from the programs; developers do not have any means to introduce handlers for the asynchronous results of the conditions.

[11] A similar analysis for AmOP's support for discovery is beyond the scope of this dissertation.

Consider as example the following AmbientTalk-like code in which a `receiveCall` method of a `messenger` object, uses an `isUrgentCall` function to determine the applicability:

```
def messenger := object: {
  def receiveCall(caller,callee) when: isUrgentCall(caller,callee) {
        ... }
}

def isUrgentCall (caller,callee) {
  caller<-isCurrentLocation(“hospital”);
}
```

Listing 3.1: AmbientTalk-like predicated method.

In this example, we assume an extension of AmbientTalk in which method definitions can be associated with a predicate function, indicated in a `when` argument (as in Predicate Dispatch [EKC98][12]). The `isUrgentCall` function determines the urgency of an incoming call according to whether the caller's current location is "hospital". For this, the `isCurrentLocation` method is asynchronously invoked on the `caller` object. The problem is that this invocation returns a future which can be handled only via a `when:becomes:catch:` form. Developers are therefore forced to move any computation depending on the predicate's future (e.g. the content of the method's body), inside the `when:becomes:catch:` form.

Composition of Event-driven Behaviour. Event-driven distribution can also hinder the composition of context-dependent behaviour. Especially for the composition approaches based on some sort of dynamic inheritance (as in all the models reviewed in Section 3.2.1, except for Context-aware Aspects and Predicate Dispatch). Upon a method invocation, those approaches select and order the available behaviour definitions (role objects, methods, aspects), according to their specificity to the invocation's context. Less specific definitions can be accessed by means of *super* calls. The problem occurs when the super calls access behaviour definitions whose results depend on asynchronous remote invocations, i.e. the remote invocation is contained in the body of the context-dependent behaviour. This means that the return value of the super call also has to be asynchronously handled. And this problem propagates all down the composition chain (i.e. there is asynchrony contagion through super calls). To cope with this situation, developers are forced to always handle super calls asynchronously, affecting the readability of the program's control flow.

[12]A similar extension has been defined by JPred [MFRW09] which introduces predicate dispatching semantics to Java.

Propagation of Network Failures

Another aspect of including remote interactions in context-dependent behaviour, is that the processes of behaviour selection and composition become vulnerable to network failures. Adequate language support should therefore be provided to handle the failures while respecting the semantics of method dispatch and inheritance. As in the case of the propagation of asynchrony, this requirement makes less suitable the approaches supporting failures as part of asynchronous result handlers (as in E and AmbientTalk). For instance, it would not be possible to use such handlers in predicate functions, as the handlers' outcome cannot be easily used as the result of the predicate functions. Network failures (and any other kind of exceptions) cannot be propagated to the method dispatch mechanism either. This would imply polluting such mechanism with application-specific logic to handle the exceptions (or defining awkwardly generic handlers for all kinds of exceptions). Similar problems appear when propagating exceptions through super calls.

Activation Scope of Event-driven Behaviour

In event-driven distributed programming models, computations are often expressed in terms of multiple event handlers. The question regarding context-dependent behaviour is how to restrict the scope of behavioural adaptations for such event-driven computations. Two strategies have been proposed for this which Desmet et al. [DVC^{+}07] defined as *loyal* and *prompt* adaptations. A loyal adaptation can span over the several handlers that compose the computation. An example of this strategy is found in JCop which uses a pointcut language to define the scope of adaptations. Models based on prompt adaptations, on the other hand, can change the adaptation for an ongoing computation as immediate reaction to context events. This strategy is implemented by Ambience which reevaluates the adaptation for a method invocation, even when making super calls.

Another concern related to the scope of the adaptations in event-driven distributed programming, is that there can be several computations occurring at the same time. In addition, those computations are not processed atomically (due to their event-driven nature) but they can be interleaved. In such a case, developers have to take care not only of the interactions between different adaptations for a computation, but also of those required by different computations. Since presumably these computations require different adaptations, the possibility exists that the programs end up with adaptations that conflict with each other. Part of this problem is directly related to the natural concurrency of distributed systems. Therefore, adaptations must be circumscribed to a restricted scope of action that is unambiguous even in the presence of concurrent and interleaved interactions.

3.3 Modelling Group Behaviour

Group data structures and group behaviour have been a natural extension to the object-oriented programming paradigm in the past. Several models with group abstractions have been defined to coordinate the work of distributed services. We review such models according to the requirements presented in Section 2.4. We map these requirements to object-oriented features based on the works of Guerraoui [GFGM98], Black [BI93] and Van Cutsem [Van08], as follows:

Plurality encapsulation A group language abstraction should make it easy to encapsulate objects that implement the same service. Group abstractions should treat a plurality of objects as if they were a single object. This way, interactions can be expressed without explicit reference to the identities and total number of the receiver objects. This also implies that interactions can use the standard request/reply scheme of object-oriented communication.

Group coordination protocols Coordination protocols should ensure consistency and continuous availability of the object group, despite concurrency and network failures. The protocols should enable developers to express group membership and communication mechanisms. Group membership mechanisms should include support for dynamic reconfigurations due to objects joining and leaving the group. Communication mechanisms should include means to propagate method invocations and to handle their results. This also means handling network failures during the group execution of methods.

Modular group behaviour Group protocols should comply with the modularity promoted by object-oriented programming. First, it should be possible to decouple the definition of group behaviour (i.e. coordination protocols) from the definition of the functional behaviour of the object members. At the same time, it should be possible that the group behaviour does not affect the interface provided by the object members.

3.3.1 Evaluation of Models for Group Behaviour

The approaches we present in this section have rather different ways to address object-oriented group behaviour. Although they share general concerns about membership and communication, most of them emphasise different aspects of the modelling of object groups. This makes the comparison less simple, but at the same time it helps us to have a broader picture about the language features required for group behaviour. Section 3.3.2 presents a summary of such language features.

Gaggles. Gaggles [BI93] are a mechanism for grouping and naming objects in a distributed system. It has been implemented as an extension to the Emerald [HRB$^+$91] object-oriented programming language. Using Gaggles programs can interact with groups without regard for the number of objects that a gaggle encapsulates. A gaggle behaves like an object: It can be named and addressed in the same way as an object. If a gaggle is invoked, one of its members is randomly chosen to receive the invocation. The caller programs are oblivious to this choice. Gaggles do not provide abstractions for group coordination and failure handling. They neither handle an explicit membership list. Instead, gaggles enable developers to construct their own protocols in whatever way is appropriate, and to encapsulate the result. Such protocols have to be defined inside the behaviour of the object members of the gaggle. In that model, each gaggle (as a whole) is represented by an object. However, such an object can be used only to add members and to get references to the gaggle. It cannot be extended to allow other invocation mechanisms.

Gaggles can be members of other gaggles. In case that the member gaggle is chosen to process a method invocation, the invocation will be recursively forwarded to one of its members.

Plurality encapsulation Gaggles encapsulate plurality in that they can represent one or more objects (or even several gaggles). A gaggle can be named and invoked exactly like an object. When a gaggle is invoked, one of the objects that it represents is randomly chosen to receive the invocation.

Group coordination Gaggles do not provide a built-in support for group coordination protocols. They just ensure that each invocation is handled by one of their members.

Modularity Gaggles provide restricted support to separate the group concern from the base functionality of the programs. Advanced invocation protocols have to be built within the methods of the objects that compose the gaggles.

The Mailer/Encapsulator Model. Mailer/encapsulator [GFGM98] is a reflective programming model that provides support for group behaviour using standard object-oriented mechanisms. The purpose of this model is two-fold: to provide an extensible built-in library of group coordination protocols, and to enable developers to transparently plug a protocol underneath object interactions (property known as group transparency). For this, group coordination protocols are explicitly separated from the application functional code, contained in two kinds of meta-objects: encapsulators and mailers. Encapsulators wrap the object members of a group, controlling the way they treat incoming and outgoing requests. Mailers are proxies to the object group. These proxies control the requests made by the object group's clients. Mailers define the propagation of the requests and the way to handle results and network failures during the communication. The Mailer/Encapsulator model has been implemented as extensions of SmallTalk and CORBA. Each implementation provides a library of encapsulator and mailer classes defining different memberships and communication strategies. New classes can be created although this implies dealing directly with lower-level protocols for asynchronous communication, intra-group concurrency control, persistence and replication. Group coordination protocols are organised with the aid of the strategy pattern, which is used in a recursive manner [GG97]. While such organisation allows for a flexible composition of protocols, it also forces the developers of the new classes to be aware of the non-trivial dependencies between the protocols.

Plurality encapsulation The Mailer/Encapsulator model achieves plurality encapsulation. Group coordination protocols are reflectively plugged into object invocations, within a program originally defined without group behaviour in mind.

Group protocols Group coordination protocols are provided by means of a complete library of encapsulator and mailer classes. Such classes provide support for important aspects of group membership and communication. Furthermore, new protocols can be added to the library, although this requires considerable knowledge on the definition and organisation of the existing protocols.

Modularity In the model, there is a strict separation between the group concerns and the base logic of the programs (the group transparency property). Group interactions are defined exclusively in encapsulator and mailer classes.

Distributed Asynchronous Collections. Distributed Asynchronous Collections or DACs [EGS00] are a language model to manipulate groups of distributed objects in Java, using a publish/subscribe programming style (discussed in Section 3.1.3). A DAC associates objects according to a topic. Developers interact with DACs by means of standard operations for collections. Adding an element to the collection corresponds to publishing an event, while invoking an operation on the collection corresponds to a subscription to notifications of elements added to the collection. Notifications are asynchronous and thus subscriptions have to be modelled in callback objects. These callbacks can contain conditions representing subscription filters.

DACs enable the unification of different publish/subscribe styles in a single framework. Each type of DAC supports different qualities of service (in a similar way as the library of coordination protocols in the Mailer/Encapsulator model). Examples of these qualities of service are reliable delivery semantics, handling of duplicate elements, delivery and insertion order, etc.

Regarding our concern on modelling group behaviour, a DAC defines two kinds of collections: the explicit collection of objects it encapsulates (which commonly correspond to event and callback objects), and the implicit collections of clients (publishers and subscribers) interacting through the DAC. In this evaluation we focus on the latter collection as the former only represents a means of communication between the clients. Group membership is defined by the topic of the DAC. Each process containing a reference to the DAC can communicate with all the clients registered to the DAC. Yet this communication is indirect. It occurs only through the DAC, by publishing or subscribing to events.

Plurality encapsulation A DAC effectively encapsulates a group of objects. Publishers and subscribers can interact through the DAC without being concerned about each other's identities. Yet, this model does not use the request/reply object communication model. Communication is possible only through the generic interface of the DAC. Developers have to explicitly create and publish message objects, and register callbacks for notifications about new message arrivals.

Group protocols That model provides support for different aspects of group communication. A library of DAC classes is provided with different styles of publish/subscribe communication. New classes can eventually be implemented, although no special semantics for extensibility are mentioned by the authors.

Modularity There is a clear separation between the definition of the client's behaviour and the group behaviour contained in DAC classes. Still, there are different communication semantics for addressing the clients as individual objects and as part of the registered members of the DAC.

Ambient References. Ambient References [VCDDM07] in AmbientTalk are references to a volatile set of objects. This set often denotes all objects of a certain type which are in communication range. Ambient references designate remote objects anonymously, by means of an intensional description. Any object that adheres to the description becomes part of the group addressed by the ambient reference. AmbientTalk provides several delivery policies to carry messages through ambient references. These policies are expressed in terms of annotations for the messages, indicating the way to propagate the message (to one or to all), to receive its answer, and to handle disconnections during the communication.

AmbientTalk's decoupled communication model (cf. Section 3.1.1) can also be observed in ambient references. Communication through this abstraction is strictly asynchronous. The sender of a message is unaware of the identities of the group of objects designated by the ambient references. Ambient references can be used to interact with services which are not yet available (buffering the messages until one or more matching services become available). Remote objects are implicitly grouped by matching the same description. All these properties are shared by publish/subscribe-based solutions for group behaviour such as Distributed Asynchronous Collections. However, ambient references do maintain the specific communication interface of object-oriented message passing. Messages sent to ambient references are handled by invoking a method provided by the interface of the service objects. Furthermore, messages can include annotations indicating how they should be propagated to the group of remote objects.

The result of a message sent to an ambient reference is a *multifuture*, a future that is resolved with the result returned by each service object the ambient reference invokes. Several extensions to the `when:becomes:` construct of AmbientTalk (cf. Section 3.1.1) are provided to handle one or all the resolutions of the multifuture.

Plurality encapsulation Ambient references achieve plurality encapsulation by implicitly grouping a volatile set of proximate service objects. This abstraction obeys the request/reply communication semantics of AmbientTalk. Ambient references do not provide generic interfaces. Messages can directly refer to methods specified in the grouped services' interface. The messages are asynchronously handled and return a future as immediate result of the messages.

Group protocols Group coordination protocols (known as delivery policies) are specified in a per-message basis, by means of annotations in the messages and different multifuture handlers.

Modularity The delivery policies are cleanly encapsulated by the ambient references. Developers have to explicitly specify the policy for each message and the handler for its multiple results. This scheme gives more control to the clients of the object group. However, it also implies scattering such specifications over the programs, at each interaction with the ambient references.

Distributed MapReduce. Distributed MapReduce is a programming pattern to support distributed computations on large data sets [DG08]. Computations are expressed in terms of user-defined `map` and `reduce` operations (based on the homologous primitives of functional languages). The `map` operation defines a function which is applied to each element of a data set received as an argument. This results in another set of *intermediate* values. The `reduce` operation receives such an intermediate set and merges its elements to form a possibly smaller set of values, typically zero or one output value.

This pattern relies on the assumption that the function defined in the `map` operation can be *independently* applied to each element of a data set. As such, rather than invoking `map` with a big set, it is possible to partition the set and make several calls to `map` with the different subsets. Such calls can then be processed by different machines in parallel. The `reduce` operation cannot be partitioned, although several `reduce` calls for different data sets can still occur simultaneously. For each group of machines involved in the MapReduce pattern, there is a *master* node which receives the invocation to the pattern's main function, `mapreduce`. That function receives as arguments the data set together with the `map` and `reduce` operations (encapsulated in an object). The master distributes the tasks among the *worker* nodes and returns the final result. Special support is provided to efficiently parallelise the work and to handle network and machine failures.

The focus of the Distributed MapReduce pattern on data parallelisation significantly differs from our focus on improving the language support for modelling group behaviour. While that pattern groups machines to distribute mapping and reduction operations, we aim to interact with a group of objects providing similar functionality. The `map` and `reduce` operations could still be used to propagate method invocations to the group members and to handle their results. However, this would contradict the pattern's original intention to execute each invocation to those operations independently, on one machine. All the guarantees of distribution provided by that pattern (failure handling and efficiency) are conceived only to coordinate several invocations to `map` and `reduce`.

Plurality encapsulation The Distributed MapReduce pattern does not provide plurality encapsulation. Developers have to directly deal with each object of the group. The request/reply communication scheme is replaced by invocations to the pattern's main functions.

Group protocols The pattern provides distribution protocols only to coordinate several `map` and `reduce` invocations on the same data set.

Modularity The pattern hides the distribution details from the programs. Computations are defined in objects implementing the `map` and `reduce` operations.

The Typed Group Model. The Typed Group Model [BBC02] is an extension to the ProActive programming language (cf. Section 3.1.2) for group behaviour in grid computing. This model enables developers to group remote objects providing a similar functionality (i.e. with a common super class). The model preserves ProActive's principle of uniform language syntax. Typed groups can be invoked using the classical dot notation, and addressing the public class interface of the members' common class. A method invocation is asynchronously propagated to all the group members. It returns a future as a result which is itself a typed group that automatically gathers all the results of the propagated invocation (which are also futures). As any standard ProActive future, a group future can be immediately used for further invocations. In that case, the *wait-by-necessity* semantics are used. An invocation on the group future blocks until one of the future receives the result. The execution of the invocation ends when all futures have been resolved.

Typed groups handle failures in a per-member-invocation way. Failures are not directly propagated but transparently caught and stored for later handling (as in AmbientTalk). If a member of a group communication raises an exception, it is stored in the result group at the exact place where the result is expected. Invocations on failed group members (exceptions) are omitted.

This model provides special semantics for passing typed groups as parameters. In case that an invocation on a group receives another group as argument, that model allows scattering the elements of the argument group in the different invocations to the members of the invoked group. This scattering is done in a one-to-one correspondence between n^{th} member of the argument group and the n^{th} member of the invoked group (where n indicates the order in which the members joined the groups). Several strategies are adopted if the two groups have dissimilar cardinality.

Finally, typed groups share two properties with Gaggles. First, typed groups can have other groups as members. The same transitive propagation scheme is applied in such a case. Second, a typed group is represented as an object which can be used to manage the group as a collection, and to get references to the group.

Plurality encapsulation Typed groups encapsulate plurality in that they can gather a set of remote objects of a common class. Method invocations preserve the syntax and implicit execution semantics of ProActive asynchronous remote invocations. The results of a group invocation are implicitly collected and do not require special event handlers.

Group protocols The model's support for group coordination is restricted to one communication protocol (full propagation of method invocations, implicit collecting of futures or exceptions). Further schemes would involve applying standard operations for collections to the typed group.

Modularity Using the Typed Group model, the group communication is abstracted from the functional aspects of the programs. Group concerns are defined at the meta level, where invocations are managed as first class entities and evaluated with the required semantics.

AWED. AWED [NSV+06] is a programming language and execution platform for aspects with explicit features for distribution. It allows developers to modularise coordination concerns in distributed programs. AWED uses as main abstractions remote pointcuts, remote advice and an explicit notion of host groups. Remote pointcuts are declarative definitions that can predicate over different points in the execution of a distributed program (called distributed join points). The pointcuts can contain conditions about groups of hosts where certain join points (e.g. a method invocations) occur. They can also indicate the hosts where the advice (the behaviour defined by the aspect) should be executed. Conditions on advice execution may further specify host selection strategies, e.g. to decide the order of application of the advice in the group of hosts. Advice is also used to manage group membership. Finally, remote pointcuts include means to manage how parameters of a given joint point (e.g. caller and target objects) are distributed. This model allows those variables to be passed by value or by reference.

Special support is provided to model group communication patterns by means of a finite-state automaton integrated into the pointcut mechanism. This support enables the definition of crosscutting (distributed) data accesses. An aspect is applied to two groups of hosts, called *source* and *target* hosts. When the aspect's pointcut matches execution events on the source hosts, its remote advice is executed on the target hosts. This adds a deterministic manipulation of distributed messages to the model by means of causally ordered protocols. In particular, the propagation of a method invocation corresponds to a *farm* pattern, where the group of sources is restricted to a single host.[13] Also, because pattern-defining aspects can be composed, nested group propagations (as proposed by Gaggles and the Typed Group model) can be straightforwardly defined as sequences of aspects. Yet, this propagation occurs implicitly, as result of the execution of the advice in the target hosts. This means that in case that the invocation expects a return value (as when using *around* aspects), the developers cannot handle the results of the propagation in one place. No single part in the program expects the results of the propagated invocation. Furthermore, developers do not have a way to deal with disconnections during the propagation.

Plurality encapsulation AWED encapsulates plurality in that it hides group communication concerns from the programs. Standard method invocations can be implicitly propagated to remote objects (providing the same interface). However, the model does not enable developers to handle the results of the propagation as well as network failures occurring during this process.

Group protocols Developers can declaratively specify the hosts that should be involved in the execution of a method invocation. They have fine control over the definition of remote pointcuts and the application of the remote advice (in terms of order of execution, parameter passing and synchronisation semantics). Furthermore, AWED provides a number of advanced communication patterns. Yet, very little or no support is provided to handle remote results and network failures.

Modularity AWED succeeds in modularising distribution and group concerns in aspects.

[13]Other patterns also restrict the number of target hosts or even of both groups.

Model	Plurality encapsulation		Group protocols						Modularity
			membership		communication				
	communication style	communication interface	group manag.	failure handl.	method propag.	param. passing	results handl.	failure handl.	
Gaggles	✓ standard request/reply	✓ member class	✓	✗	✓	✗	✓	✗	✗ group manager, modifiable only in member class
Mailer/Encapsulator model	✓ standard request/reply	✓ member class	✓	✓	✓	✗	✓	✓	✓ mailer and encapsulator meta objects
Distributed asynchronous collections	✗ standard request, callback-based reply	✗ DAC class	✓	✗	✓	✗	✗	✗	✓ DAC class
Ambient References (AmbientTalk)	✗ group request, multifuture-based reply	✓ member object	✓	✓	✓	✗	✓	✓	✓ ambient reference
Distributed MapReduce	✗ group request, standard reply	✗ MapReduce operation	✗	✗	✓	✗	✗	✗	✓ MapReduce class
Typed Group model (ProActive)	✗ standard request, multifuture-based reply	✓ member class	✓	✗	✓	✓	✓	✓	✗ group manager, modifiable only in member class
AWED	✓ standard request/reply	✓ member class	✓	✗	✓	✓	✗	✗	✓ aspect

Table 3.3: Programming style for group behaviour

3.3.2 Group Behaviour: Synthesis and Discussion

Table 3.3 summarises the language support provided by the models for group behaviour presented in this section:

Plurality Encapsulation. The main purpose of plurality encapsulation is to access a remote service without regard for the number of objects that provide the service. We can evaluate this property according to two aspects of the communication between the object groups and their clients:

Communication style There are a number of models that preserve the standard request/reply object communication model (Gaggles, the Mailer/Encapsulator model and AWED). Requests correspond to ordinary method invocations and the replies are implicitly returned as results of the execution of the invocations. The rest of the approaches define dedicated support for group requests (the group annotations in invocations on Ambient References, or the `MapReduce` object in the Distributed MapReduce pattern), and for group replies (multifutures in AmbientTalk and ProActive, and callback objects in DACs). Note that this explicit support is not necessarily related to the asynchronous execution semantics featured by some of the approaches. The Mailer/Encapsulator model provides asynchronous execution yet it does not require special abstractions for group requests or replies.

Communication interface Most of the models enable developers to invoke methods that correspond to the interface provided by the class of the group members. The only exceptions are DACs and the Distributed MapReduce pattern which use generic operations.

Group Protocols. In Table 3.3, we focus on the protocols for group membership and communication. Further protocols are provided, remarkably in the Mailer/Encapsulator model and DACs (for data replication management, transactional operations, etc.). Yet, group membership and communication are the basis of the other protocols. Regarding the protocol for group membership, two properties can be distinguished:

Group management All the models provide some sort of support for group management, i.e. to add and remove group members. In most cases, such operations are provided by the entity that represents the group as a whole, e.g. a group manager, a group reference or a collection. In all other approaches, group management is an implicit process. In DACs, for instance, members are added and removed by registering to, or unregistering from a topic. Similarly, Ambient References use a publish/subscribe discovery mechanism based on intensional descriptions.

Failure handling An important aspect of the management of a group is to provide means to react to failures affecting its members' availability. In this sense, we observe that only the Mailer/Encapsulator model and Ambient References provide support for handling such failures. In the former model, this support is included as part of the behaviour of the `encapsulator` meta object. In the latter, member failures are supported at the program level, by means of connectivity handlers.

With respect to the protocols for group communication, we distinguish four concerns:

Method propagation Several mechanisms have been proposed to propagate a method invocation to the group. Gaggles and Typed Groups feature a unique and implicit propagation strategy (to one member and to all members, respectively). Models such as DACs, Ambient References and AWED allow the invoking program to declaratively specify the propagation (in the conditions of callback objects, group annotations of messages, and aspects' pointcut, respectively). The Mailer/Encapsulator model provides different propagation strategies contained in the mailer meta classes.

Parameter passing Few of the approaches provide special parameter passing semantics for group communication. By default, the parameters of a method invocation on a group are broadcast to all the group members. Each parameter is then passed according to the semantics provided by the language, e.g. by copy, by reference, by move, etc., typically specified in the class definition of the parameter object. The only two exceptions are AWED and Typed Groups. AWED enables developers to specify the propagation of the parameters (by copy or by reference) at the aspects' pointcuts. Typed Groups define extra semantics to cope with groups used as parameters (giving the possibility to scatter the members of the group parameter over the invocations on the members of the invoked group).

Results handling Several strategies are proposed to handle the results of group method invocations. In Gaggles, the results are handled implicitly; the result produced by the member executing the group invocation is transparently sent to the invoking object. Ambient References and Typed Groups, on the other hand, provide a set of synchronisation abstractions, to wait for one or all the results (modelled as futures). In Ambient References, the synchronisation is specified as a property of the method invocation (as part of the group annotations). In Typed Groups, the synchronisation can be indicated directly in the result future. The Mailer/Encapsulator model also provides several synchronisation schemes which are contained in the mailer meta classes. The other models allow the developers to manually synchronise the results of the group invocation: inside the callback objects in DACs, and in the *reduce* operation in the Distributed MapReduce model. AWED does not allow the handling of the results of group invocations.

Failure handling Most of the schemes proposed to deal with failures during group communication, use the failure handling semantics of the underlying programming model. Ambient References and Typed Groups notify failures during the invocation on a group member, by ruining the future created by the invocation. In the Mailer/Encapsulator model, failures are implicitly handled at the meta level, as part of the behaviour of the mailer meta classes. Similarly, the DAC classes contain different strategies to handle failures.

Modularity. In terms of modularity, we can observe that all the approaches in the survey provide specific entities to contain the group protocols. Most of these entities can be adapted without losing their modularity. The only exceptions are the group managers of Gaggles and Typed Groups which can be modified only from within the class definition of their members. For the rest, different levels of adaptability are provided. Using Ambient References, the membership protocols can be modified by means of the references' intensional description. Communication protocols can be selected by means of the annotations attached to the method invocations. In the cases of DACs, Distributed MapReduce and AWED, the group protocols can be adapted by directly redefining the classes that contain the protocols. Similarly, the Mailer/Encapsulator model allows the definition of new `mailer` and `encapsulator` meta classes (or redefinition of existing ones). That model is also the one that offers the highest degree of flexibility for adapting the protocols, due to its reflective capabilities. Yet, such flexibility also entails more complexity.

Another point of comparison that can be derived from this evaluation is the basis on which a group behaviour is selected, i.e. the *scope* of use of the group communication protocols (in a way similar to the scope of context-dependent behaviour, cf. Section 3.2). Models such as Gaggles, the Mailer/Encapsulator model, DACs and Typed Groups, specify such protocols on a *per-group* basis. This means that the same communication strategy is used for handling all invocations to the group. Alternatively, Ambient References and the Distributed MapReduce model specify the group communication strategy on a *per-invocation* basis. Finally, AWED's scope of the group communication protocol is *per aspect*, i.e. a set of methods matched by the aspects' pointcut definitions.

3.3.3 Group Behaviour in Ambient-oriented Programming

We have focused on programming models for group behaviour on distributed systems. For this reason, some of the models already include support for coping with distribution issues, e.g. for network failures, as we summarise in the previous subsection. Regarding the distribution requirements of the ambient-oriented programming paradigm, we observe the following consequences for modelling group behaviour.

Decentralised Group Behaviour Management

AmOP's decentralised discovery property, as realised in AmbientTalk for example, requires that the members of a group can autonomously act upon the dynamic changes in the availability of their peers. The execution of group protocols, therefore, cannot rely on a specific host (i.e. a fixed group leader) as there is no certainty that the connection with that host will remain accessible. Approaches such as the Mailer/Encapsulator model, DACs, Ambient References and AWED have acknowledged this issue enabling group protocols to work in a peer-to-peer fashion.

Decoupled Group Communication

To deal with volatile connections, the AmOP paradigm fosters a communication model which is decoupled in time, space, synchronisation and arity (cf. Section 2.2). An approach for group behaviour in this case should support the same decoupling in the interactions between the groups and their clients. This is accomplished by the Mailer/Encapsulator model, DACs, Ambient References, and to some extent, by Gaggles, Typed Groups and AWED. Furthermore, it should also be possible that the decoupled communication is ensured for intra-group interactions, as those required in the definition of the group communication protocols. However, this is not the case for the models that enable redefinitions of the communication protocol. Developers have to interact with the group as a collection (no arity decoupling), using explicit references to the object members (no space decoupling), only with the members that are online (no time decoupling), and in some cases, only via synchronous messages (no synchronisation decoupling).

Group's External and Internal Connection-independent Failure Handling

To properly deal with connection volatility, interactions with object groups should support for connection-independent failure handling, similar to the one provided by Ambient References. This support should also be provided for interactions between the members of the group (inside the definition of the communication protocols).

3.3.4 Context dependency and Group Behaviour

A final matter we have to discuss is the combination between context dependency and group behaviour. The main challenge here is that the language abstractions defined for both concerns should work without interfering with each other. This implies coping with the following issues.

Context-dependent Selection of Group Behaviour

Dynamic changes in the context of an object group can influence its behaviour. Such changes can cause requests to the group, possibly for the same functionality, to be executed using a different communication protocol. Similarly, different context conditions may lead to different ways to react to member disconnections. Consider as a simple example a communication application running simultaneously on several devices (for a common user, as the case presented in Section 2.1). For such a service, the default group communication protocol can be to signal incoming messages in all the devices but to store them in only one of them. Yet, if the device that stores the messages becomes unavailable, a new group strategy should be adopted. Chapter 7 presents further examples of context-dependent group behaviour. We observe that in all these cases the selection of the adaptation of the group behaviour according to the context, is application dependent. This makes approaches with generic group protocols (defined in a per-group basis) less suitable, especially those that are more difficult to adapt (as the reflective Mailer/Encapsulator model).

Activation Scope of Group Behaviour

Allowing context-dependent variations of group behaviour also implies dealing with the activation scope of such variations. The issue is how to react to context changes occurring during the execution of an invocation to the group (i.e. during the propagation of the invocation, distributed execution and handling of results). This brings back the discussion about loyal and prompt behavioural adaptations (cf. Section 3.2.3), which can be included as part of the definition of the group protocols. Similarly, additional support should be provided to deal with concurrent invocations to the group, eventually leading to the selection of different group protocols.

Preserving Modularity and Composability

Last but not least, when modelling context dependency and group behaviour, the possibility exists that the organisation of a program required for one of these concerns hinders the modularity of the other. The challenge in this case is how to ensure the modularity of both concerns while preserving the program readability. Additionally, a flexible composition mechanism is required to allow a consistent combination of the different variations of context-dependent group behaviour.

3.4 Summary: Modularity in AmOP

In this chapter, we have reviewed the state of the art of three fundamental concerns in the software development for pervasive computing: distribution, context dependency and group behaviour. We have used as the basis for our study the ambient-oriented programming paradigm and its event-driven execution semantics. We have described the way in which such semantics have been combined with object-oriented programming. We presented a number of language models to support context dependency and group behaviour, and discussed the conditions for their integration with event-driven execution semantics of AmOP. From these conditions we derived a list of requirements which further refine the requirement for *interdependent support* proposed in Section 2.5. Figure 3.2 shows the requirements for AmOP, context dependency and group behaviour, and the requirements for interdependent support.

In summary, in object-oriented programming languages with an event-driven execution model, objects discover, communicate and deal with network failures, by means of events in the form of asynchronous messages. Such languages support event-driven distribution either *explicitly*, by providing dedicated language syntax for remote communication, or *implicitly*, by using a uniform syntax (for local and remote communication). Explicit language support makes clear the effects of distribution in the programs, but hinders their evolvability (in what we call the *asynchrony contagion* problem). Furthermore, some models with explicit event-driven abstractions suffer from the problems of *inversion of control* and *lost continuations*. Implicit language support, on the other hand, eases the programs' evolvability but hampers the understandability on the effects of distribution. Finally, event-driven program execution models suffer from the *event interleaving*

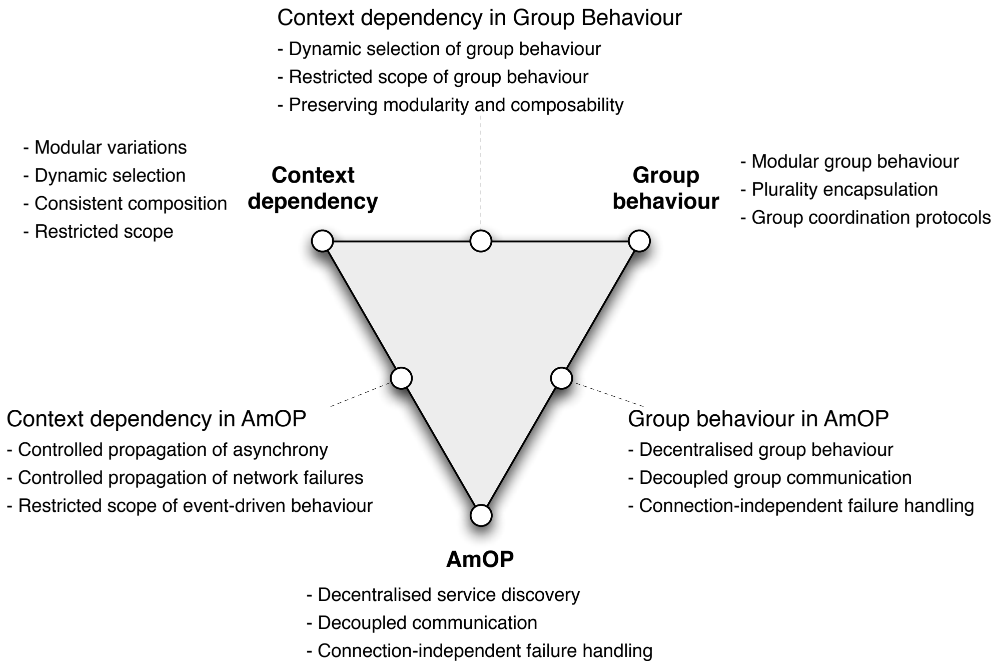

Context dependency in Group Behaviour
- Dynamic selection of group behaviour
- Restricted scope of group behaviour
- Preserving modularity and composability
- Modular variations
- Dynamic selection
- Consistent composition
- Restricted scope
Context dependency
Group behaviour
- Modular group behaviour
- Plurality encapsulation
- Group coordination protocols
Context dependency in AmOP
- Controlled propagation of asynchrony
- Controlled propagation of network failures
- Restricted scope of event-driven behaviour
Group behaviour in AmOP
- Decentralised group behaviour
- Decoupled group communication
- Connection-independent failure handling
AmOP
- Decentralised service discovery
- Decoupled communication
- Connection-independent failure handling

Figure 3.2: Requirements for modularity of context dependency and group behaviour in AmOP.

problem. Although this is not a problem of programming syntax, languages with implicit support for distribution make this problem more difficult to identify.

In the programming language models reviewed in Section 3.1.2, we observe that to cope with the above issues the models make specific decisions on the use of implicit and explicit syntax for each aspect of communication (for invoking methods, receiving the invocations, and handling their results).

In what follows, we recapitulate the requirements for modelling context dependency and group behaviour in AmOP.

3.4.1 Modelling Context Dependency in AmOP

Object-oriented programming languages with support for context dependency enable behavioural variations to be modularised and specialised on particular context conditions. Context-dependent variations are dynamically selected, activated and composed for a specific scope of the program execution. The modularity required for context-dependent behaviour is decomposed into three aspects: modular partial behaviour definition (mostly expressed as method definition), modular groups of partial definitions (e.g. in layers or aspects), and modular behaviour selection (e.g. in predicate functions or context objects). The selection and composition of context-dependent behaviour are modelled as part of the method dispatch mechanism of the programming languages. Special support is provided to select and order behavioural variations according to the context (based on hierarchies of context, roles or predicates). The scope of context-dependent behavioural variations is mostly defined by the lexical or dynamic extent of the execution of a method invocation.

Regarding the combination between the event-driven communication model of AmOP, and the models for context-dependent behaviour, we derive the following requirements:

$R_{I.1}$ Controlled propagation of asynchrony The selection of behaviour can depend on remote context conditions. Because the selection process is commonly modelled as part of the languages' method dispatch mechanism, language support required for asynchronously requesting remote context conditions should not interfere with the semantics of method dispatch. Similarly, in case that the selection process leads to a composition of context-dependent behaviour, the possibility exists that the asynchrony contagion occurs through super calls. Programming language models should enable developers to handle such results without hampering the readability of the programs (e.g. without forcing the methods to be defined in a continuation-passing style).

$R_{I.2}$ Controlled propagation of network failures Dynamic dispatch and composition mechanisms based on inheritance can be affected by network failures. Programming language models should therefore provide explicit support to deal with failures while respecting the semantics of such mechanisms.

$R_{I.3}$ Restricted scope of event-driven behaviour An event-driven computation is fragmented into a number of event handlers. In case that such a computation requires context-dependent behavioural adaptations, programming language models should ensure a consistent scope for the adaptations. This scope should be unambiguous even in the presence of concurrent and interleaved computations.

3.4.2 Modelling Group Behaviour in AmOP

Object-oriented programming languages with support for group behaviour provide different ways to achieve plurality encapsulation, group protocols and modularity. Plurality encapsulation aims to enable programs to interact with a remote service without regard for the number of objects that provide the service. This property can be evaluated with regard to two different aspects: the group communication style (in comparison to the standard request/reply object communication model), and the group communication interface (whether invocations to the group are handled using the interface of the member objects or a generic group interface). A model for group behaviour should define a number of protocols for group coordination. At the basis of these protocols there is the support for group membership and communication. For both cases, the models should define the handling of network failures. Group protocols should be modularised so that group behaviour is separated from the base functionality of the programs. Programming language models achieve this by encapsulating the group behaviour in dedicated entities (e.g. group objects, meta objects, references or aspects). Each of these entities define a particular scope of use of the group protocols, which can be per group, per invocation, and per aspect.

Regarding the combination of the event-driven communication model of AmOP, and models for group behaviour, we derive the following requirements:

$R_{I.4}$ Decentralised group behaviour management The decentralised discovery promoted by the AmOP paradigm requires that the member of a group can autonomously act upon the dynamic changes in the availability of their peers. Group protocols should be able to work in a peer-to-peer fashion.

$R_{I.5}$ Decoupled group communication AmOP's decoupled communication should be ensured for both the communication between the group and their clients, and for the intra-group communication (the latter required inside the definition of the group protocols).

$R_{I.6}$ Connection-independent failure handling Language support should be provided for handling network failures during both the interactions with the group and those between the group members. This support should be connection-independent to properly deal with transient disconnections.

Finally, to build an integrated object-oriented model that deals with context-dependent and group behaviour, the following requirements should be accomplished:

$R_{I.7}$ Dynamic selection of group behaviour Programming models should enable the definition and dynamic selection of group protocols according to context.

$R_{I.8}$ Restricted scope of group behaviour Context-dependent variations of group behaviour should have a consistent scope during the complete execution of a method invocation (i.e. during the propagation of the invocation, distributed execution and handling of results). This also means to keep consistency even in the presence of possible network failures affecting the intra-group interactions.

$R_{I.9}$ Preserving modularity and composability It should be possible that the modularity required for context-dependent behaviour does not interfere with the modularity required for group behaviour. The same applies to the dynamic composition of either kind of behaviour.

Chapter 4

Ambient-oriented Programming in Lambic

This chapter introduces *Lambic*, a new incarnation of the ambient-oriented programming paradigm [VME^{+}07]. However, instead of starting with the single-receiver OOP variant, Lambic is an extension of the generic function-based object system of Common Lisp (acronym CLOS). Our model extends the multiple dispatch semantics of generic functions to cope with the concerns of pervasive computing, identified in Chapter 2. For concurrency and distribution, Lambic combines the properties of the AmOP paradigm with generic functions, in what we call *futurised generic functions*. For context dependency and group behaviour, Lambic provides two other extensions, called *predicated generic functions* and *group generic functions* respectively. Additionally, a common underlying execution process ensures that these three features can be effectively used in combination with one another.

An in-depth discussion of CLOS is beyond the scope of this thesis. Instead, in the next section we briefly explain the essentials of its object model required to understand Lambic —especially the part concerning generic functions. We introduce further details as we go along. The remainder of the chapter presents futurised generic functions, and Chapters 5 and 6 introduce predicated and group generic functions respectively. In the course of these chapters we also describe the execution process that enables the integration of the three features.

4.1 Generic Function-based Object Orientation in Lambic

Lambic adheres to the mainstream idea of object-oriented programming (established by Simula) to organise a program around classes and then associate operations with those classes, in the form of methods. However, in CLOS methods do not belong to classes but to *generic functions*. A generic function defines an abstract operation, specifying a

Class definition ::=
(`defclass` *class-name* (*parent-class*)({*field*})) $\rightarrow$ *new-class*

field ::= (*field-name field-option*)
field-option ::= [`:initarg` *initarg-name*]
[`:initform` *form*]
[`:reader` *reader-function-name*] ...

Class instantiation ::=
(`make-instance` *class-name* [*init-values*]) $\rightarrow$ *new-object*

Table 4.1: Class definition and instantiation in Lambic

name and a parameter list that its methods can specialise on. Methods are accessible only by invoking the corresponding generic function. This means that programs are written in terms of generic function invocations rather than messages exchanged between objects. Thus, generic functions preserve the procedural programming style of Common Lisp. However, in CLOS, this does not imply that we lose the virtues of OOP. Generic functions allow the methods to specialise on the class of all their parameters, establishing the basis for multiple dispatch semantics (also known as multimethods).

4.1.1 Class Definition and Instantiation

Table 4.1 shows Lambic's syntax for the definition and instantiation of classes. For this, we use an adaptation of the EBNF notation.[1] Classes are defined by means of the `defclass` form which receives as arguments a name, a list of parent classes[2] and a list of fields. A basic field definition consists of a symbol denoting the name of the field. Alternatively, Lambic enables complementing such a definition with a number of options (inherited from CLOS) in the form of keyword arguments. For instance, the `:initarg` option specifies a name that can be used as a keyword parameter when instantiating the class (using the `make-instance` function), and whose argument will be stored in the field. The `:initform` option is used to specify a default value for the field if no `:initarg` is passed to the `make-instance` function. Finally, the `:reader` option specifies the name of the generic function used to access the field (similar to a *getter* method in other languages). As an example, consider the definition and instantiation of a `chat` class shown in Listing 4.1. The `chat` class has two fields representing the name of the user (`username`) and the list of the user's contacts (`address-book`). The `username` field can be initialised when instantiating the class, using the `:username` keyword (specified in the `:initarg` option). The `address-book`

[1]Due to the central role of parentheses in Lambic's syntax, we have substituted this symbol in the EBNF specification (used for grouping terms), with angle brackets. The expressions after the arrows indicate the return values.

[2]Although Lambic preserves the multiple inheritance semantics of CLOS, a complete analysis of the interaction between these semantics and our extension remains part of future work.

Generic function definition ::=	
(`defgeneric` *function-name parameter-list*)	→ *new-generic*
parameter-list ::= ({*parameter*})	
Method definition ::=	
(`defmethod` *method-name specialised-parameter-list body*)	→ *new-method*
specialised-parameter-list ::= ({*parameter* | (*parameter specialiser*)})	
Generic function invocation ::=	
(*function-name* {*argument*})	→ *result*

Table 4.2: Generic functions and methods in Lambic (as in CLOS)

field is bound to a hash table by default (indicated in the `:initform` option). The second expression in the listing creates a new instance of the `chat` class with the string `"Bob"` as the initial value for the `username` field. This instance is then bound to the `local-chat` variable.

```
(defclass chat ()
  ((username :initarg :username
             :reader get-username)
   (address-book :initform (make-hash-table)
                 :reader get-address-book)))

(defvar local-chat (make-instance 'chat :username "Bob"))
```

Listing 4.1: Class definition in Lambic.

4.1.2 Generic Function and Method Definitions

Table 4.2 shows the language constructs for the definition of generic functions and methods in Lambic. Just as in plain CLOS, generic functions are defined using the `defgeneric` form which receives as parameters a name and a parameter list. This list consists of one or more symbols denoting the name of each parameter. The generic function's individual methods are defined independently by means of the `defmethod` form. This form receives the name of a generic function and a specialised parameter list. A specialised parameter is a parameter associated with a symbol denoting the name of a class (the *specialiser*). All the method's parameters can have a specialiser. The specialised parameter list should have the same arity as the parameter list of the method's generic function.

```
(defgeneric receive-text (receiver sender text))

(defmethod receive-text ((receiver chat) sender text)
  (display “Message from ” sender “: ” text))
```

Listing 4.2: Generic function and method definitions in Lambic.

The above listing shows the definition of the `receive-text` generic function. This generic function specifies the behaviour of the `chat` class to receive a text message. A method is defined for this generic function with the `receiver` parameter specialised in the `chat` class. The generic function is invoked as follows:

```
> (receive-text local-chat “Alice” “Hi there”)
Message from Alice: Hi there
```

Listing 4.3: Invoking a generic function in Lambic.

We defer the explanation on how Lambic processes generic function invocations to Chapter 5. For now, it suffices to assume the traditional CLOS semantics.

4.2 Futurised Generic Functions

The existing incarnations of the AmOP paradigm use the popular single-receiver OOP variant. In Lambic, we adhere to the paradigm by means of a model, called *futurised generic functions*. It is a variation of the communicating event loops model of the AmbientTalk [VME+07] programming language, which combines the non-blocking execution process of event loops with the multiple dispatching semantics of generic functions. In this model, software services can discover, communicate and deal with network failures by means of events, which are represented as generic function invocations asynchronously processed by actors.

Futurised generic functions comply with the three properties of AmOP: decentralised discovery, decoupled communication and connection-independent failure handling. We put special emphasis on the language support for communication which, as explained in Chapter 3, has a significant influence on the modularity of the behaviour of pervasive computing services. For this reason, futurised generic functions extend the execution process of communicating event loops, for handling the results of non-blocking remote invocations. As such, Lambic enables the definition of distributed computations using *explicit* syntax (as originally proposed in the event loops model, cf. Section 3.1.1), and also *uniform* syntax (the same syntax of local invocations).

The contributions of futurised generic functions are two-fold:

1. To reconcile multiple dispatch semantics with the communicating event loops model. Lambic gracefully aligns generic functions with event-driven programming for service discovery, communication and failure handling.

2. To provide explicit and uniform syntax for communication, while guaranteeing the event-driven execution of remote interactions.

In the remainder of this chapter, we present Lambic's generic function-based event loops model. Next, we introduce Lambic's event-driven programming style.

4.3 Generic Function-based Event Loops

Futurised generic functions support concurrency and distribution by featuring an extension to *communicating event loops*, the actor-based model originally proposed by E and further refined for pervasive computing by AmbientTalk. In Lambic, software services can discover, communicate and deal with network failures by means of events, which are represented as asynchronous generic function invocations. Generic function-based event loops in Lambic are built around three main components: *actors*, *asynchronous generic function invocations* and *non-blocking futures.*

4.3.1 Actors

In Lambic, an actor consists of an event loop (a thread of execution), an event queue, and an internal state represented by a collection of objects and generic functions. Events correspond to asynchronous generic function invocations which are received in the actor's queue and sequentially processed by its event loop, dispatching to the appropriate generic function. Figure 4.1 shows the communicating event loops model in Lambic.

Actors define boundaries of concurrent execution for objects and generic functions: Each object and generic function are contained in exactly one actor and as such they can be accessed exclusively by their containing actor's event loop. This means that two or more actors never share mutable state. Mutating another actor's state has to be performed indirectly, by means of asynchronous generic function invocations. These invocations are processed in a strictly sequential order, one at a time, in what is known as an event loop *turn* [Mil06]. Turns are Lambic's unit of atomicity and interleaving. The handling of a single invocation happens in mutual exclusion with respect to other invocations. All communication between actors is non-blocking: An actor never suspends its event loop to wait for the result of an asynchronous generic function invocation processed by another actor.

Intra- and Inter-actor Object References. In Lambic, standard (local) references to objects can be used only inside the objects' actor. Object references spanning several actors are called *remote references.* When a local reference to an object crosses the

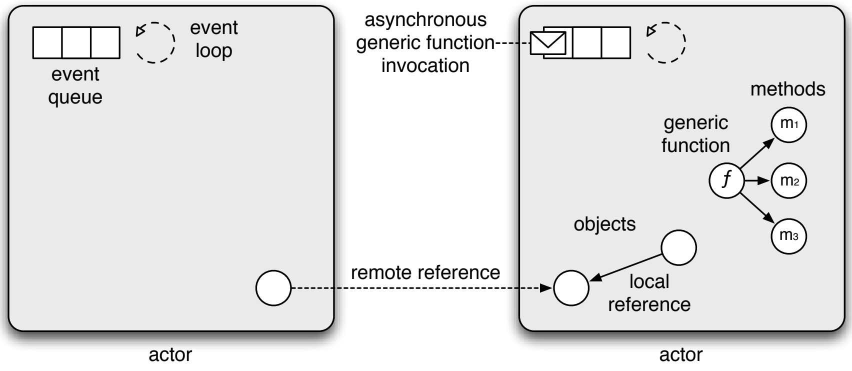


Figure 4.1: The communicating event loops model in Lambic

boundaries of the object's actor, e.g. when passed as argument or return value of a generic function invocation, it is automatically converted into a remote reference. Conversely, a remote reference received in the actor that contains the corresponding referenced object, is converted into a local reference.

4.3.2 Asynchronous Generic Function Invocations

In Lambic, inter-actor computations are possible exclusively by means of asynchronous generic function invocations. A generic function is asynchronously invoked by designating an actor as the responsible for its execution. In our model neither actors nor objects can be addressed directly, i.e. sending messages to them. Instead, an actor is indirectly selected by associating with the invocation a reference to an object contained in such actor. We name this reference the *actor designator*. An asynchronous generic function invocation consists of the specification of a standard generic function invocation (name of a generic function and a set of arguments) and an actor designator. The standard generic function invocation is processed in the actor of the object indicated as actor designator.

In existing actor models with single-dispatch message-passing semantics, the actor that executes an invocation always corresponds to the actor of the receiver argument (which can be the actor itself as in Salsa, or an object hosted in the actor, as in AmbientTalk). In our model, however, no argument plays such a role in an invocation. Thus, the object reference used as actor designator is independent of the references passed as arguments to the generic function (although it will often coincide with one of them). This means that developers can arbitrarily choose an actor to process any invocation.[3] Also, the actor designator of an asynchronous generic function invocation can correspond to a local or remote reference. In either case the invocation is scheduled in the actor's event queue of the referenced object. This means that using a local reference as actor designator does

[3]As we illustrate in Chapter 7, this is beneficial for cases of several actors providing the same generic function.

not result in a synchronous execution of the invocation, but in scheduling the invocation in the current actor's event queue.

Implicit Actor's Message-passing Semantics. Note that an asynchronous generic function invocation corresponds to a "send" operation at the actor level which, in Section 1.4.3, we claimed incompatible with multiple dispatch semantics of generic functions. However, in our case this operation does not conflict with the generic function model as methods are only specialised on objects, not on actors. The only role of actors is to prevent concurrent invocations on the same set of objects. The message-passing actor communication is explicitly separated from the programming level which enables the developers to write programs in a generic function style.

4.3.3 Asynchronous Return Values

Lambic handles the return values of asynchronous generic function invocations by means of *non-blocking futures*, as in AmbientTalk. By default, an asynchronous generic function invocation returns a future as immediate result. A future is an object created at the actor doing the invocation, acting as a placeholder for its result. Once the return value is computed, it is communicated to the future's actor in the form of an event. The future is then said to be *resolved* with the value. An actor in Lambic cannot suspend on an unresolved future. Actions that depend on the result of an asynchronous generic function invocation are defined by registering an observer for its corresponding future.

The observer of the future corresponds to a closure that will be applied to the future's resolved value. The registration of the observer is a non-blocking process itself. The closure is executed immediately only if the future is already resolved. Otherwise, it is processed asynchronously after the future's resolution. A future and its observer are always contained in the same actor, which is also the actor that initiated the asynchronous generic function invocation that returned that future. As such, only the actor of the future can notify its observer, preventing concurrent interactions with the internal state of the actor. This also ensures that the bindings of the closures used as observers are available when handling the future's result. Yet, the possibility exists that the values of those bindings might have changed as a result of other events processed by the actor, between the asynchronous generic function invocation and the resolution of its future [VME$^+$07]. This situation, that we name *message interleaving*, is an important issue of non-blocking execution models, and as such, we analyse it in depth in Section 4.5.1.

Future-based Exception Handling. When an exception is raised during the execution of an asynchronously invoked generic function, the exception propagates back to the actor performing the invocation. At this point, the only available continuations are the observers registered to the future returned by the asynchronous invocation. Hence, the exception is signalled by *ruining* such a future [VME$^+$07]. This action also occurs in the form of an event sent to the future's actor. To deal with a ruined future, developers can attach one or more exception handlers to the future's observer. These handlers are asynchronously called after the future is ruined.

4.3.4 Summary

In summary, Lambic's futurised generic functions align the communicating event loops model with the multiple dispatch semantics of generic functions, as follows:

- Actors define boundaries of concurrent execution of generic functions. Methods are specialised on objects and their classes, not on actors. Hence, actors play no role in the method dispatch semantics.
- Inter-actor computations are realised by means of asynchronous generic function invocations. This kind of invocation differs from standard generic function invocations in that it has to specify the actor that should evaluate the function.
- By default, the result of an asynchronous generic function invocation (if any) is also asynchronously returned to the actor from which the function is invoked (the future's location).

4.4 Lambic's Event-driven Programming Style

We now present Lambic's programming language support for the AmOP paradigm. This support is built on top of the generic function-based event loop model. We explain how Lambic realises the three properties of AmOP: decentralised discovery, decoupled communication and connection-independent failure handling. In this section, we show that in our model all these features are cleanly aligned with the multiple-dispatch semantics of generic functions. The message-based interaction of actors is explicitly separated from the programming level, abstracted in a number of generic functions. Thus programs do not send messages to objects but only invoke generic functions, indicating the actor that should process them.

We gradually introduce the syntax and semantics of Lambic as necessary in this section. For a complete definition of the Lambic language, we refer the reader to Appendix B.

4.4.1 Decentralised Discovery

Table 4.3 shows Lambic's language support for service discovery. Our model deals with the lack of fixed network infrastructure of mobile ad hoc networks (cf. Section 2.2) by adopting the publish/subscribe service discovery protocol of AmbientTalk. This protocol is fully decentralised, no servers or other infrastructure are required. Objects are published by providing a description of the kind of service they represent. A subscription takes the form of the registration of a discovery observer on a service description. This observer is notified whenever an object exported under that description becomes available in the network. As a result of the discovery, Lambic yields a remote reference to the exported service. The discovery is supported by means of the functions `export-service` and `import-service`, and the event handler `whenever-discovered`.

Export service ::=	
(`export-service` *object service-description*)	→ *nil*
Import service ::=	
(`import-service` *service-description*)	→ *unbound-reference*
Discovery event handler ::=	
(`whenever-discovered` *service-description lambda*)	→ *nil*

Table 4.3: Lambic's syntax for decentralised discovery

Exporting Remote Services

The `export-service` form receives as parameters the object to be exported and a description of the service it provides. By default, the service description corresponds to a text tag denoting the class of the exported object. For the case of class tags, we assume that all hosts agree on the meaning of the classes.

The following code snippets show how to publish a `chat` object on the wireless network:

```
(defvar chat-tag (make-class-tag 'chat))
(defvar local-chat (make-instance 'chat))
(export-service local-chat chat-tag)
```

Listing 4.4: Actor exporting service.

The actor executing the code above creates an instance of the `chat` class and exports it using the class tag corresponding to the chat, stored in the `chat-tag` variable. For the sake of simplicity, we have encapsulated this definition in the `make-class-tag` function.

Importing Remote Services

Objects are imported using the `import-service` construct. This form specifies a service description and returns a possibly *unbound* remote reference as a result. This reference acts as a proxy to the requested service which waits for an object providing the service to become available. Once the remote object is discovered, it is "bound" to the remote reference. The behaviour of the remote object can be accessed only by means of asynchronous generic function invocations using the remote reference as actor designator. The remote reference can be used in asynchronous invocations even if the service object it represents is not yet discovered. In such a case, the unbound remote reference buffers the invocations until the object is discovered. After discovery, the remote reference flushes all the stored invocations to the corresponding actor.

```
(defvar chat-tag (make-class-tag 'chat))
(defvar remote-chat (import-service chat-tag))
```

Listing 4.5: Actor importing service.

The actor executing this code imports a service providing the same `chat-tag`. This actor binds the remote reference returned by the `import-service` form to the `remote-chat` variable.

Discovery Event Handlers

In Lambic, the discovery of a service is also modelled as an event which can be handled by means of a `whenever-discovered` form. This form installs an observer for discovery events, allowing the programs to perform actions that are processed whenever an object matching a service description is discovered. This observer receives as argument a remote reference to the discovered remote object. As in the case of asynchronous generic function invocations, discovery events are asynchronously received and processed by the actor that installs the corresponding observers. Thus, discovery event handlers can never be executed concurrently with other activities of the same actor. The following example illustrates the use of the `whenever-discovered` form in the definition of a function that creates a chat service:

```
(defun create-chat (username)
  (let ((local-chat (make-instance 'chat :username username))
        (chat-tag (make-class-tag 'chat)))
    (export-service local-chat chat-tag)
    (whenever-discovered chat-tag
      (lambda (remote-chat)
        (add-contact local-chat remote-chat)
        (display-contact local-chat remote-chat)))
    local-chat))
```

Listing 4.6: Creation of a chat service.

The function above creates a local instance of the `chat` class which is published using the name of its class as a service description (`chat-tag`). Then, the function imports other instances in the network that were published with the same tag, adding them to the local instance's address book and graphic user interface. Finally the function returns a reference to the local chat as a result (`local-chat`).

Actor definition ::=	
(spawn-actor *name* {*object*})	→ *actor*
Remote generic function invocation ::=	
(in-actor-of *actor-designator function-invocation*	
[:with-future ⟨*future* | nil⟩]	
[:due-in *seconds*])	→ *future* | *nil*
Remote result handling ::=	
(when-resolved *future function*	
[:catch *exception-handler*])	→ *nil*

Table 4.4: Lambic's explicit syntax for communication.

4.4.2 Decoupled Communication

As in existing manifestations of the communicating event loops model, Lambic provides a dedicated language syntax for remote inter-actor communication. However, in addition to this explicit support, the execution process of our model can implicitly handle the futures of asynchronous invocations. This means that remote computations can also be defined using standard generic function invocations (as in local, intra-actor computations). These invocations are internally converted into asynchronous invocations and processed accordingly. Yet, Lambic's aim is not to advocate the use of a particular kind of syntax for object-oriented distributed programming —*explicit* or *uniform*. Providing both kinds of syntax has enabled us to better understand their benefits and drawbacks for coping with distribution issues. Furthermore, our experiences have shown that both syntaxes can complement each other, which is especially beneficial for the introduction of distribution to existing programs. In this section, we first explain the syntax proposed in both cases —for the definition of actors, generic functions, and remote invocations, and for handling remote results and exceptions. We then present the results of the integration of both syntaxes.

Explicit Syntax for Communication

Table 4.4 shows Lambic's explicit syntax for asynchronous remote communication. Lambic provides language support for defining actors, asynchronously invoking remote generic functions and handling results and exceptions (which in our case corresponds to handling the resolution or ruining of non-blocking futures).

Actor Definition. An actor is defined using the `spawn-actor` construct which receives as arguments the name of the actor and a set of objects representing its state. The classes of such objects, as well as the generic functions with methods specialised on such classes, also become part of the actor. Yet, because in Lambic's event loop model actors cannot share state, the classes and generic functions are copied to the new actor (so that there is no sharing between the creating and created actors). The `spawn-actor` expression returns a reference to the actor as a result.[4]

By default, Lambic provides an actor that represents the hosting device. Objects that are not explicitly included in any user-defined actor (using the `spawn-actor` construct) are owned by such a default actor.

The example below shows the definition of an actor for the chat service, named `“chat-actor”`, with the `local-chat` object as its state:

```
(spawn-actor “chat-actor” local-chat)
```

Listing 4.7: Definition of an actor.

Explicit Remote Generic Function Invocations. Lambic provides a dedicated form for asynchronously invoking remote generic functions, called `in-actor-of`. This form receives as arguments a remote reference to an object, and a generic function invocation. The remote reference is the actor designator of the remote invocation. It indicates the actor that should process the generic function invocation passed as the second argument. This generic function invocation follows the standard syntax of generic function calls presented in Table 4.2. The actor designator is independent of the arguments of the generic function invocation. Still, the same object reference can be used both as actor designator and as argument. Such independence enables developers to freely choose the actor that will process the invocation.

The following listing illustrates an asynchronous generic function invocation using the `in-actor-of` form:

```
(in-actor-of remote-chat (get-username remote-chat))
```

Listing 4.8: Asynchronous invocation of a remote generic function.

This expression can be read as “process the invocation of the `get-username` generic function in the actor of the `remote-chat` object”. In this invocation, `remote-chat` is a reference to a remote instance of the `chat` class. By passing this remote reference as an actor designator, we ensure that the invocation of the `get-username` generic function is processed by the remote `chat` object’s actor.

By default, the `in-actor-of` form immediately returns a future. Alternatively, this form enables a finer manipulation of the future by means of the optional argument `:with-future`, explained below. Similarly, the `in-actor-of` form enables the specification of a

[4]The actor is itself an object also contained in the actor, and implicitly created by Lambic.

timeout for the reception of the result of the remote invocation (using the `:due-in` optional argument).

Explicit Remote Result Handling. Actions that depend on the result of an asynchronous generic function invocation are defined by registering an observer for the future returned by the invocation. This is done by means of the `when-resolved` form. This form receives as arguments a future and a function representing the observer. The lambda's body is evaluated after the resolution of the future. The result of the resolved future is then bound to the parameter of the lambda. The return value of the `when-resolved` form itself is `nil`.

The following listing illustrates the use of the `when-resolved` form:

```
(defvar name-future (in-actor-of remote-chat (get-username remote-chat)))
(when-resolved name-future
  (lambda (contact-name)
    (display “Contact’s name: ” contact-name)))
```

Listing 4.9: Defining an observer for a future.

In this example, the invocation of the `display` generic function is evaluated after the resolution of the future returned by the asynchronous invocation of the `get-username` generic function. The result of the resolved future (a string in this case) is then bound to the `contact-name` parameter of the passed function.

Futures in Lambic are instances of a `future` class. As such they can be directly created, passed as arguments to function invocations, resolved with a value and ruined with exceptions. As we explain in Section 4.3.3, this enables developers to use futures for a special kind of synchronisation between processes, called *conditional synchronisation* [VME+07]. This also enables developers to indicate the future that an asynchronous generic function invocation should resolve. This is done by passing a `future` object in the `:with-future` optional argument of the `in-actor-of` form. Conversely, invocations that do not expect any result, known as *one-way* invocations [VME+07], can be expressed by passing `nil` to the `:with-future` argument. One-way invocations do not create any future, returning `nil` as their result.

Explicit Remote Exception Handling. As explained in Section 4.3.3, an exception that occurs during an asynchronous generic function invocation ruins the future that corresponds to the invocation. In Lambic, a ruined future can be handled by defining one or more exception handlers[5] in an optional `:catch` argument of the `when-resolved` form. For instance, the following code illustrates the handling of a `division-by-zero` exception (adapted from [Van08]):

[5]Standard Common Lisp exception handlers [BDG+88].

```
(when-resolved (in-actor-of remote-calculator (/ x y))
  (lambda (quotient) (display "result :" quotient))
  :catch (division-by-zero ()
           (display "Error: divided " x " by zero.")))
```

Listing 4.10: Handling exceptions of remote generic function invocations.

Generic Function Definitions. None of the language abstractions above require changes in the signatures of generic functions and methods. The semantics of Lambic's event loop model ensures that a generic function (and its methods) can be defined as presented in Section 4.1.2, regardless of whether it is invoked synchronously or asynchronously.

Uniform Syntax for Communication

As an alternative to the explicit language syntax described above, Lambic enables the definition of remote calls to generic functions, and handling of their results, using the same syntax as standard (local) invocations. To achieve this, we extend the execution process of the event loop model to support *implicit* actor designation and implicit future handling. This extension ensures that remote invocations are internally converted and executed in an asynchronous manner. Yet, methods containing remote invocations have to be explicitly annotated as *interruptible*. This way, developers can still be aware of the methods that are executed asynchronously. Alternatively, Lambic enables the methods to be annotated as *uninterruptible* to prevent its asynchronous execution to be interleaved with the execution of further invocations. Table 4.5 shows Lambic's uniform syntax for communication.

Actor Definition. Lambic's uniform syntax still requires new actors (other than the default actor) to be defined by means of the `spawn-actor` form.

Uniform Generic Function Invocations. Using Lambic's uniform syntax, a generic function invocation is executed in the actor of its *first* argument, called the *implicit actor designator*. Passing a local object reference as this argument leads to a standard (synchronous) generic function execution, whereas passing a remote object reference leads to an asynchronous invocation, executed at the object's remote actor. This also means that a standard invocation can now return a future as result. Invocations to remote generic functions cannot control the future object they return (as in the case of the `:with-future` argument of the `in-actor-of` form). Neither can they include timeouts for the resolution of their future. Such a timeout should be indicated outside the invocation, using the `*response-timeout*` dynamic variable. By default, this variable defines a timeout of 10 seconds.

Actor definition ::=	
(`spawn-actor` *name* {*object*})	→ *actor*
Method definition ::=	
(`defmethod` *method-name* [*execution-qualifier*] *specialised-parameter-list body*)	→ *new-method*
execution-qualifier ::= `:interruptible` | `:uninterruptible`	
specialised-parameter-list ::= ({*parameter* | (*parameter specialiser*)})	
Generic function invocation ::=	
(*function-name implicit-actor-designator* {*other-argument*})	→ *result* | *future*
Exception handler ::=	
(`try-catch` *expression* {*error-clause*})	→ *result* | *future*
error-clause ::= (*exception-name* ({*argument*}) *body*)	

Table 4.5: Lambic's uniform syntax for communication.

Using Lambic's uniform syntax, the expression introduced in Listing 4.8:

```
(in-actor-of remote-chat (get-username remote-chat))
```

can also be expressed as:

```
(get-username remote-chat)
```

Lambic processes the invocation of the `get-username` generic function in the actor of its first argument (and only argument in this case), `remote-chat`. Because this argument is a remote reference, the invocation gets internally converted into an asynchronous invocation, similar to the `in-actor-of` form.

Uniform Result Handling. In Lambic, generic function invocations can receive futures as arguments and return futures as results. This allows remote computations to be written sequentially, constructing a chain of futures which effectively encodes a data-flow graph. No special callbacks are required to receive the results of remote method invocations. Thus, the example of the `when-resolved` form of Listing 4.9 can be replaced by the nested expression of Listing 4.11.

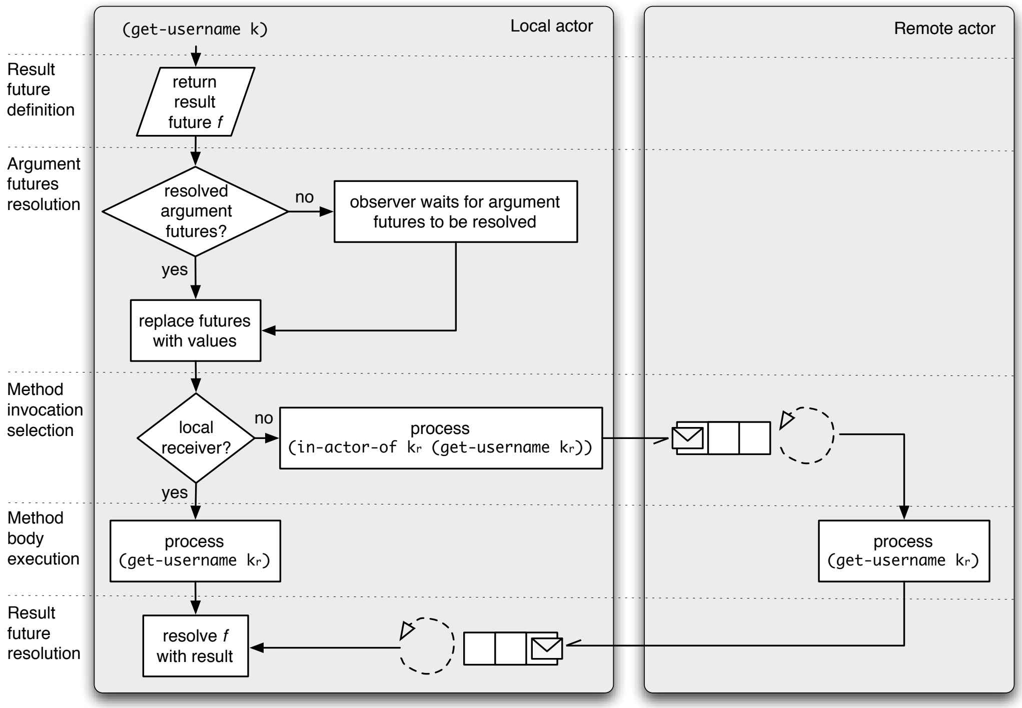

(get-username k)
Local actor
Remote actor
Result future definition
return result future f
Argument futures resolution
resolved argument futures?
no
observer waits for argument futures to be resolved
yes
replace futures with values
Method invocation selection
local receiver?
no
process (in-actor-of kr (get-username kr))
yes
Method body execution
process (get-username kr)
process (get-username kr)
Result future resolution
resolve f with result

Figure 4.2: The Lambic futurised method execution process

```
(display “Contact’s name: ” (get-username remote-chat))
```

Listing 4.11: Nesting local and remote generic function invocations.

Lambic preserves the sequential programming style using the semantics depicted in Figure 4.2. This process consists of the following five steps:

1. *Result future definition.* Lambic's semantics create a future that will be returned as the result of the generic function invocation. This future is implicitly generated; the generic function and method definitions are oblivious to it.

2. *Argument futures resolution.* In a second step, Lambic checks if futures have been passed as arguments to the invocation. In such a case, the execution is deferred until the resolution of the futures. This way we preserve the execution semantics of CLOS in which the arguments are evaluated before applying the generic function. We model this situation by defining an observer for the unresolved futures (a function), which captures the continuation of the generic function execution.

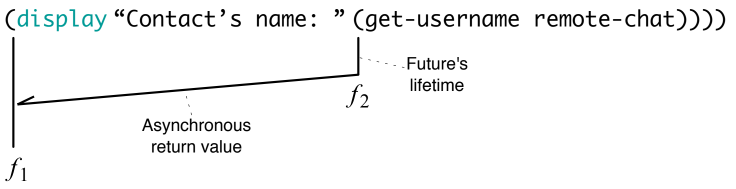

(display "Contact's name: " (get-username remote-chat))))
Future's lifetime
f_2
Asynchronous return value
f_1

Figure 4.3: Implicit futures handling in Lambic.

Then the observer is asynchronously invoked after all the futures become resolved. The futures are replaced with their result values (in the figure this is illustrated by replacing the argument k with k_r).

3. *Method invocation selection.* The third step consists of checking the location of the actor designator of the generic function invocation, in order to determine its adequate execution (synchronous or asynchronous).

4. *Method body execution.* The fourth step corresponds to the actual local or remote execution of the generic function.

5. *Result future resolution.* The final step is the resolution of the future of the generic function invocation. Such a resolution can occur synchronously or asynchronously as the result of a local or remote execution respectively.

Figure 4.3 shows the data-flow graph that results from the implicit handling of futures for the invocation of the `display` generic function of Listing 4.11. The second argument of this function corresponds to the invocation of the `get-username` generic function which has a remote object reference as actor designator (`remote-chat`). As such, it is internally converted into an asynchronous invocation and immediately returns a future, f_2. The `get-username` generic function is then executed at the actor of the `remote-chat` object. The `display` generic function also returns a future, f_1, which is computed when f_2 is resolved (in the figure, the vertical lines represent the lifetime of the futures, from their creation until their resolution, while the arrow represents the asynchronous return of the future's result). This means that the actor's event loop that executes this code is not blocked: It simply immediately returns a future itself. When the event loop later receives result for f_2, it resumes the computation on f_1. No threads are blocked or created during this process, execution is entirely event-driven, where "events" are either incoming remote method invocations or replies to earlier invocations containing the results for unresolved futures.

Uniform Exception Handling. Lambic enables the use of standard exception handling forms for remote invocations, such as the `try-catch` construct. In this case, the exception handlers defined inside the `try-catch` construct are internally included in the `:catch` argument of the `when-resolved` structure that waits for the result of the invocation. The following code shows an example of the use of the `try-catch` form for the remote invocation presented in Listing 4.10:

```
(try-catch (display “result: ” (in-actor-of remote-calculator (/ x y)))
  (division-by-zero ()
    (display “Error: divided ” x “ by zero.”)))
```

Listing 4.12: Sequential exception handling for remote invocations.

The `try-catch` form receives as arguments the expression to be processed (e.g. the invocation of the `display` function) and one or more exception handlers.[6] Note that in this example we still use the `in-actor-of` form as the actor designator (the `remote-calculator` object) is not part of the argument list passed to the invocation to the division function.

A future that is ruined with an exception also ruins the futures that depend on it (with the same exception), and this propagation continues until the exception is handled in a `try-catch` form (condition known as *promise contagion* [Mil06]). Figure 4.4 illustrates this propagation for the expression of Listing 4.12. Ruining the future of the `in-actor-of` form, f_3, with a `division-by-zero` exception, also ruins the future created by the invocation of the `display` function, f_2. However, f_2 cannot ruin the future of the `try-catch` form, f_1, as this form appropriately handles the `division-by-zero` exception. Thus, f_1 is finally resolved with the result of the corresponding handler.

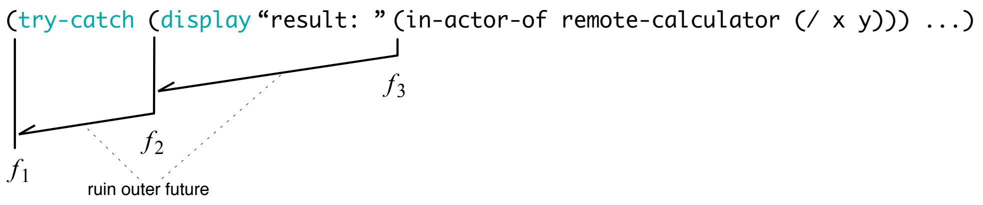

(try-catch (display “result: ” (in-actor-of remote-calculator (/ x y))) ...)
f_3
f_2
f_1
ruin outer future

Figure 4.4: Propagation of the ruin of a future.

[6]In Lambic, we have adapted the `try-catch` form and other control structures so that they work with non-blocking futures while still preserving their evaluation rules. See Section A.3 for further details about this adaptation.

Interruptible and Uninterruptible Methods. Lambic's uniform syntax preserves the sequential execution of the forms that traditionally compose the body of a method. This is even so if the forms are not part of each other's data flow i.e. there is no data dependency among them. This sequentiality is achieved by processing a form only after the execution of the previous form in the method body has finished —and after the resolution of its future, in case that the previous form is an asynchronous invocation. Lambic adopts the approach of the Kilim [SM08] and TaskJava [FMM07] programming language models (cf. Section 3.1.2), to require that methods containing implicit remote invocations are explicitly annotated. For this, an `:interruptible` keyword should be added to the signature of the methods. This annotation is also mandatory for all methods defined in the same actor invoking the "interruptible" method. Therefore, developers acknowledge the methods that are executed asynchronously. Such an annotation is checked at runtime. Lambic throws a warning if a method that cannot be fully processed synchronously, is not annotated as interruptible.

Consider as example the following method definition of the `add-contact` generic function, which has to be identified as `:interruptible` because it invokes the `get-username` generic function of the (remote) actor of the `remote-chat` object:

```
(defmethod add-contact :interruptible ((local-chat chat) remote-chat)
  (let ((username (get-username remote-chat))
        (address-book (get-address-book local-chat)))
    (sethash address-book username remote-chat)))
```

Listing 4.13: Interruptible method definition.

Alternatively, Lambic enables methods containing remote invocations to be annotated with the `:uninterruptible` keyword. This way, the asynchronous execution of the method is prevented from being interleaved with the execution of further invocations. Those invocations remain in the actor's queue until the completion of the "uninterruptible" method, and are then processed in exactly the same order in which they are received. Thus, an uninterruptible method has exclusive access to its enclosing actor's state during the complete asynchronous execution. Note that this feature reintroduces blocking semantics to the communicating event loop model, which in Section 2.2 we claim unsuitable for remote interactions in pervasive computing. Inside an uninterruptible method, the use of the future created by a remote invocation blocks the actor's event loop until the future is resolved. This scheme resembles the *wait-by-necessity* semantics of ProActive [BBC$^+$06] (cf. Section 3.1.2). However, in our case the remote invocation is always implicitly associated with a timeout. As we explain in the next section, this timeout will automatically ruin the future of the invocation with an exception, which can be handled with a `try-catch` form. Thus, Lambic ensures that the event loop will always resume the execution, either with the result of the invocation or with an exception. Furthermore, in Lambic the blocking execution semantics are optional, only applicable for uninterruptible methods. The default execution semantics in our model are still non-blocking.

A special case should be mentioned, when within the body of an uninterruptible method there is an invocation to another (local) method defined as interruptible. In such a situation, Lambic preserves the blocking semantics (to handle futures) of the uninterruptible method. The reason for this is that the "uninterruptible" qualifier indicates that a method *must* be processed without interleavings of other method executions. Instead, the "interruptible" qualifier only indicates that a method contains remote invocations that *may* lead to interleavings. The execution semantics of the interruptible method will not be affected if it is processed in an uninterruptible manner.

The same priority between the execution qualifiers is used in the opposite case, invocations to uninterruptible methods within an interruptible method. The uninterruptible methods always preserve their execution semantics. Finally, methods invoking an uninterruptible method do not need to be annotated themselves. This is because the blocking semantics of the uninterruptible method prevent any possibility of interleaving for those methods.

Section 4.5.4 further illustrates the benefits and limitations of interruptible and uninterruptible methods.

Complementarity of Explicit and Uniform Syntax for Communication

Action	**Explicit syntax**	**Uniform syntax**
Actor definition	`spawn-actor`	`spawn-actor`
Method definition	—	`:interruptible`, `:uninterruptible`
Remote invocation	`in-actor-of`	—
Remote result handling	`when-resolved`	—
Remote exception handling	`when-resolved :catch`	`try-catch`

Table 4.6: Lambic's event-driven programming style for communication.

Table 4.6 summarises Lambic's explicit and uniform syntax for communication. The explicit syntax requires developers to distinguish remote from local interactions by means of dedicated language abstractions, to invoke remote generic functions and to handle the results and exceptions. The uniform syntax does not make this distinction; remote invocations use the standard syntax for local invocations. However, it requires that methods explicitly state that they are interruptible or uninterruptible.

There is little overlap between both syntaxes. Basically, it is limited to the correspondence of expressing a remote generic function invocation using the `in-actor-of` form, and using the standard syntax of local invocations. And this is true only if the actor designator corresponds to the first argument of the generic function invocation. The next equation shows this correspondence:

```
(in-actor-of actor-designator (function-name actor-designator other-arguments))
↔
(function-name actor-designator other-arguments)
```

Lambic also enables these two syntaxes to be used in complement with each other. This is particularly convenient for developing the pervasive computing services we target in this dissertation (cf. Chapter 2). The behaviour of such services depend on local and remote context conditions which are often hard to express in a single control flow (requiring non-trivial conditional structures). Lambic allows developers to build such programs using a uniform syntax, and then to gradually introduce explicit abstractions to differentiate remote from local generic function invocations. We further elaborate on this scenario in Chapter 7. When using both syntaxes in combination, the following interaction rules between the communication abstractions are applied:

- The `in-actor-of` and `when-resolved` forms can be used inside methods annotated as `:interruptible` or `:uninterruptible`. However, none of these annotations have effects on such forms. A `when-resolved` form *never* blocks the actors' event loop.
- An `in-actor-of` form can be replaced by a standard generic function invocation, and vice versa, if the first argument of the invocation corresponds to the actor designator (due to the correspondence described above).
- The return value of a `when-resolved` form does not depend on the result of the execution of its function. The default return value of the `when-resolved` form is `nil`.

4.4.3 Connection-independent Failure Handling

Table 4.7 shows Lambic's language syntax for connection-independent failure handling. As in AmbientTalk, Lambic provides support for handling network failures at two different levels. Our model enables programs to be aware of the changes of connectivity of the discovered services, and to handle the effects of network failures in asynchronous generic function invocations.

Connectivity Event Handlers.

In Lambic, remote references report the changes of connectivity of the remote objects they represent. When a network or machine failure occurs in the host of a remote object, the remote references pointing to it are said to become *disconnected*. Analogously, the remote references become *reconnected* when the object is available in the network again. Programs can be aware of these changes by registering observers for monitoring connectivity events for remote references. This is possible by means of the connectivity event handlers, `when-disconnected` and `when-reconnected`. Both forms consist of a remote reference and

Discovery event handler ::=	
(whenever-disconnected *remote-reference lambda*)	→ *nil*
Discovery event handler ::=	
(whenever-reconnected *remote-reference lambda*)	→ *nil*

Table 4.7: Lambic's syntax for connection-independent failure handling

a body representing the actions to be performed upon the event's occurrence. The body of the `when-disconnected` form is executed whenever the remote reference passed as first argument, becomes disconnected. Similarly, the body of the `when-reconnected` form is executed after the remote reference becomes reconnected. As in the case of the `whenever-discovered` form, connectivity event handlers are processed exclusively by the actor that defines them. The following code shows an extension of the `create-chat` function presented in Listing 4.6, this time using the connectivity event handlers for monitoring the availability of discovered remote `chat` instances:

```
; Inside the create-chat function
...
(whenever-discovered chat-tag
  (lambda (remote-chat)
    ...
    (when-disconnected remote-chat
      (hide-contact local-chat remote-chat))
    (when-reconnected remote-chat
      (display-contact local-chat remote-chat))))
```

Listing 4.14: Connectivity event handlers.

The connectivity event handlers in this example are used to keep the graphic user interface up to date according to the changes of availability of the remote contacts of a chat service.

Handling Partial Network Failures

In Section 2.2, we explain that the main challenge of handling network failures in pervasive computing is that it is impossible to distinguish upfront a permanent from a transient failure [VME$^+$07]. Therefore, it is not always desirable to disrupt the programs' control flow of a distributed interaction upon experiencing a network failure, as the failure might be only transient. To deal with this issue, Lambic adopts AmbientTalk's time-based failure handling mechanism, which abstracts from the volatility of network connections by enabling asynchronous generic function invocations to specify a timeout that delimits the period of time to send the invocation to the remote actor and receive the corresponding result. This time interval is respected even if network failures occur in between. A network failure is considered permanent only after the timeout is reached, in which case an exception is raised. As discussed in Section 4.3.3, raising an exception in an asynchronous generic function invocation corresponds to ruining the future of such invocation.

To support the time-based failure handling mechanism, the `in-actor-of` form provides an optional argument `:due-in` that specifies the deadline to be imposed to a remote invocation. Additionally, our model defines a `timeout-exception` that can be handled in the `:catch` block of a `when-resolved` form.[7] We illustrate the use of this by adding support to deal with partial failures to the example of Listing 4.9:

```
(when-resolved (in-actor-of remote-chat (get-username remote-chat) :due-in 10)
  (lambda (contact-name)
    (display “Contact’s name: ” contact-name))
  :catch (timeout-exception () (display “Contact went offline”)))
```

Listing 4.15: Handling partial failures.

The expression `:due-in 10` imposes a due limit of 10 seconds on the asynchronous invocation of the `get-username` function invocation. If the future is not resolved within 10 seconds, the handler for the `timeout-exception` included in the catch expression is executed.

Our model's uniform syntax also allows developers to handle a `timeout-exception` using a `try-catch` form, as shown in Listing 4.16.

```
(try-catch
  (display “Contact’s name: ”
           (in-actor-of remote-chat (get-username remote-chat) :due-in 10))
  (timeout-exception () (display “Contact went offline”)))
```

Listing 4.16: Handling partial failures.

Using this time-based failure handling mechanism, there can be several scenarios of disconnection to deal with. First, the possibility exists that an asynchronous generic function invocation uses a disconnected remote reference as actor designator. In such a case, the invocation is buffered until the remote reference is reconnected and the invocation is eventually delivered, or the invocation's timeout is reached and the invocation is discarded. In Lambic, each disconnected remote reference buffers the asynchronous invocations using this remote reference as actor designator. As such, the invocations are

[7] Similar to `due` blocks in AmbientTalk [VME+07].

guaranteed to be buffered, and eventually delivered, in the same order in which they occur in the program's control flow.[8]

Second, a remote reference may become disconnected while sending the asynchronous invocation to the remote actor. Lambic handles this case in the same way as when using a disconnected remote reference (buffering the invocation and eventually sending it if the remote reference reconnects before the timeout).

Third, a remote reference may become disconnected after the asynchronous invocation is delivered but before the result is returned. In this case, the actor making the invocation can only wait for the result or the timeout. Thus far, Lambic does not provide any means to notify the actor executing the asynchronous invocation in case the timeout is reached.

4.4.4 Summary

In summary, Lambic fulfils the properties of ambient-oriented programming (cf. Section 2.2), as follows:

- Services are discovered using a decentralised publish/subscribe protocol. Objects can be exported and discovered by means of class tags denoting the service they provide. No address or location is required (space decoupling).

- Events for discovery, communication and failure handling are all represented as generic function invocations. Event handlers can be defined for each of these events.

- Communication events correspond to asynchronous generic function invocations (synchronisation decoupling).

- Asynchronous generic function invocations are resilient to partial failures. When an actor that is supposed to process a remote invocation becomes disconnected, the invocation is buffered until the connection is restored (time decoupling).

4.5 Lambic's Support for Communication Revisited

In this section, we revisit Lambic's programming support for communication. We discuss the benefits and limitations of futurised generic functions in the light of the issues of event-driven programming models for concurrency and distribution, described in Section 3.1.2. We illustrate Lambic's solution for managing mutable state, inversion of control, lost continuations, ripple effects, and event interleaving. We finally discuss the interoperability of futurised generic functions with existing libraries in CLOS.

[8]Note that remote references might also buffer asynchronous invocations that do not indicate any timeout condition. In Chapter 7 we discuss the way in which Lambic avoids inconsistencies related to long-lasting buffering of invocations.

4.5.1 Managing Mutable State

As explained in this chapter, actors in Lambic define boundaries of concurrent execution for objects and generic functions. A method can be executed exclusively by its containing actor's event loop, and an object's state can be directly operated on only by the methods that are co-located with the object (in the same actor). To illustrate this property, consider the scenario of the Geuze collaborative drawing editor introduced in Section 2.1. To preserve a consistent drawing, the collaborating editors can designate a leader that controls the access to the different shapes of the drawing (peer editors ask the leader for the access to the shapes). In Lambic, we can define each editor in a different actor, ensuring that the requests to the leader are possible only via asynchronous generic function invocations. These invocations are conveniently serialised in the message queue of the leader's actor and executed one at a time.

Listing 4.17 shows the method definition for the `mouse-down` generic function (using Lambic's uniform syntax). This method is used by the editors to handle the GUI event of a user pressing the mouse's main button. This method contains an invocation of the `allow-selection` generic function, which the editors use to request the leader for the access to a shape. After the leader grants the access, the requester editor invokes the `select-shape` generic function to perform the selection (e.g. to display a graphic effect on the shape).

```
(defmethod mouse-down :interruptible ((editor geuze) shape x y)
  (let ((leader-editor (get-leader editor)))
    (if (allow-selection leader-editor editor shape)
      (select-shape editor shape))))
```

Listing 4.17: Leader-driven shape selection.

The sequence diagram of Figure 4.5 shows how Lambic's event loop model handles concurrent invocations of the `allow-selection` generic function (for the same `circle` shape). Our model ensures that the invocation coming from `editor 2` is processed only after the execution of the invocation coming from `editor 1` (and thus the access to the shape is granted only to `editor 1`).

4.5.2 Inversion of Control

To cope with the problem of inversion of control, i.e. to avoid fragmented control flows and manual stack ripping (cf. Section 3.1.2), Lambic proposes a two-fold solution. Using Lambic's explicit syntax the inversion of control is solved by handling the asynchronous results of remote generic function invocations, using in-line closures (as in AmbientTalk). These closures can be defined in `when-resolved` forms in the same context where the remote invocations are issued. In-line closures also capture their enclosing execution environment. This way, no manual stack ripping is required.

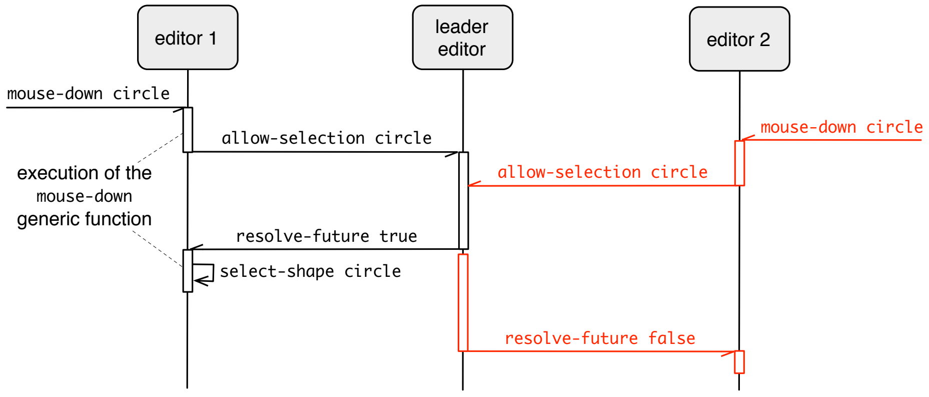


Figure 4.5: Leader editor as synchronisation point.

Using Lambic's uniform syntax, the asynchronous results of remote invocations are implicitly handled by the event-driven execution process. Remote interactions can therefore be written in a direct style. No continuation forms are required in this case. This solution is similar to the implicit asynchronous continuation management performed by ProActive, Kilim, TaskJava, Lua and JaCoBox (cf. Section 3.1.2). The only condition for such implicit handling is to properly annotate the methods containing remote interactions (as *interruptible* or *uninterruptible*). This approach is based on Kilim's and TaskJava's method qualifiers, `@pausable` and `async` respectively. In particular, those qualifiers have the same semantics as Lambic's `:interruptible` qualifier.

4.5.3 Lost Continuations

To cope with the problem of lost continuations, i.e. to cover all the possible ways in which an asynchronous request may terminate, Lambic uses futures as the implicit return address for remote generic function invocations. The result of a remote invocation is always implicitly communicated to its corresponding future. Exceptions occurring during the remote execution of the invocation are also communicated to the future. The future is then said to be ruined by the exception. A ruined future can be handled with a dedicated `:catch` argument in the `when-resolved` forms, in case of Lambic's explicit syntax (as in E and AmbientTalk), and with standard `try-catch` forms in the case of Lambic's uniform syntax (as in ProActive and JCoBox). The same future ruining mechanism is used to notify timeout exceptions. Thus, the future produced by a remote invocation is guaranteed to always be resolved with a result, or ruined with an exception.[9]

[9] Of course, there is always still a chance of lost continuations occurring if the developer forgets to install an observer on the future.

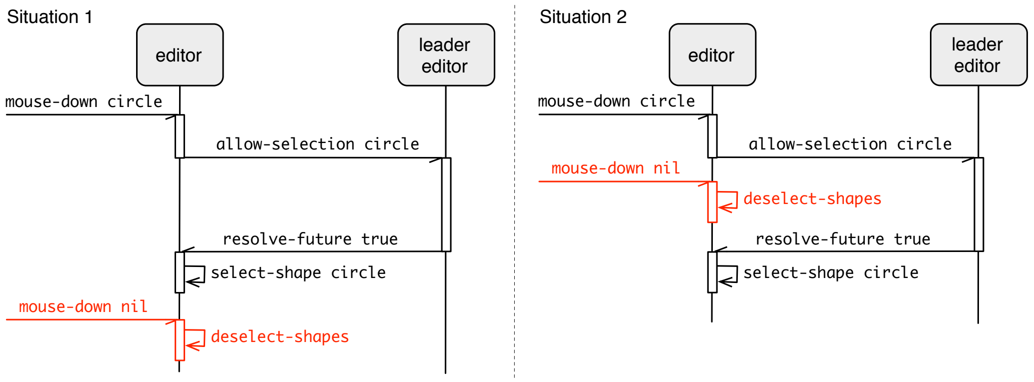


Figure 4.6: The invocation interleaving problem.

4.5.4 Event Interleaving

Lambic's communicating event loops promote non-blocking execution semantics. Actors can therefore interleave the asynchronous executions of different generic function invocations. However, this very possibility can also lead to inconsistencies in the actor's state: Processing another invocation that changes the actor's state may affect the continuation of the interrupted invocation. To illustrate this problem, consider the following extended version of the `mouse-down` method of Listing 4.17:

```
(defmethod mouse-down :interruptible ((editor geuze) shape x y)
  (if shape
    (let ((leader-editor (get-leader editor)))
      (if (allow-selection leader-editor editor shape)
        (select-shape editor shape)))
    (deselect-shapes shape)))
```

Listing 4.18: Two ways to handle the *mouse-down* event.

In this definition, the *mouse-down* event is handled differently depending on whether it occurs on a shape or on the editor's canvas. In the former case, a reference to a shape is passed in the `shape` argument, causing the start of the selection process for the shape. In the latter case, the `shape` argument is `nil`, causing the deselection of all the shapes in the canvas (by invoking the `deselect-shapes` generic function). These two cases can occur one after the other, i.e. when the user presses the mouse button first on a shape and then on the canvas. The result of this sequence of events can vary according to whether the second event is effectively executed *after* or *between* the execution of the first event (as shown in Figure 4.6). Executing the second event after the first one ends with all the shapes deselected, which is the result expected by the user (Situation 1 in the figure). However, executing the second event during the execution of the first one may end with the shape still selected (Situation 2 in the figure), as the `deselect-shapes` generic function is processed before the leader allows (and communicates) the selection of the shape.

The interleaving of events may render the non-blocking execution of a generic function unpredictable. Developers should therefore pay special attention to this problem when enabling methods to be interruptible. Using Lambic's explicit syntax alleviates this problem as it allows developers to distinguish the parts of the methods that are executed asynchronously, by enclosing them in `when-resolved` forms. However, the problem remains as there is no certainty about when these forms are executed.

```
(defmethod mouse-down :interruptible ((editor geuze) shape x y)
  (if shape
    (let ((leader-editor (get-leader editor))
          (future (in-actor-of leader-editor
                                (allow-selection leader-editor editor shape))))
      (when-resolved future
        (lambda (allowed)
          (if allowed
            (select-shape editor shape)))))
    (deselect-shapes shape)))
```

Listing 4.19: Explicit syntax for the `mouse-down` method.

Listing 4.19 shows the definition of the `mouse-down` method using explicit syntax. In the example described in Figure 4.6, the second part of the execution of the `mouse-down` method (when called for the first time) would now correspond to the execution of the function inside the `when-resolved` form. Although explicit, the interleaving between this function and the second call to `mouse-down` can still lead to the two situations described in the figure.

Lambic's uniform syntax provides a basic solution to the event interleaving problem. It consists of enabling methods to be annotated as `:uninterruptible`. This annotation constrains the actor's event loop to execute the uninterruptible method until its completion. This restriction also causes the event loop to block in order to wait for the results of remote interactions inside the method. No further invocations can be processed in between. The event loop can be resumed only to receive the remote result or to signal a timeout exception for the remote invocation.

Annotating the `mouse-down` method of Listing 4.19 with the `:uninterruptible` qualifier will ensure the appropriate execution of this method. This execution is shown in Figure 4.7. The two invocations to `mouse-down` will not be interleaved but executed one after the other.

The blocking execution semantics of uninterruptible methods solve the event interleaving problem in event-driven execution models. Similar approaches have been proposed by

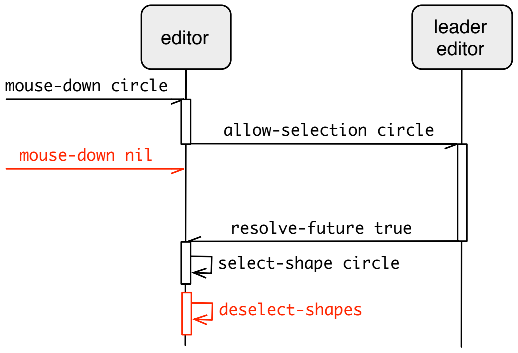


Figure 4.7: Uninterruptible methods in Lambic.

ProActive, JCoBox and TaskJava. Yet, these semantics can be too restrictive for some situations. In particular, a finer-grained mechanism could be added to selectively prevent only specific method invocations from being interleaved, as in Lua's synchronisation constraints (cf. Section 3.1.2). This extension remains an important part of our future work (cf. Chapter 8).

4.5.5 Asynchrony Contagion

As explained in Section 3.1.2, the asynchrony contagion is related to the degree of uniformity in the language support for local and remote interactions. Lambic's explicit syntax enables developers to isolate the effects of asynchronous remote interactions in the programs (i.e. asynchronous generic function invocations and future-based result handling). As in AmbientTalk, the problem of asynchrony contagion of this syntax is related to the explicit distinction between local and remote generic function invocations and result handling. This often entails more verbose code and less straightforward control flows. Lambic's uniform syntax, on the other hand, enables programs to be less verbose and to keep the representation of control flows sequential (as in local object-oriented computations). Still, the asynchrony contagion problem can still be observed in this case when annotating methods as *interruptible* or *uninterruptible*. Not only should the method containing remote invocations provide such annotation, so should other methods invoking the annotated method. This way developers can still be aware of the part of the programs affected by asynchronous executions. However, the support for dealing with remote interactions is more restricted than when using explicit syntax.

Both syntaxes can be used in combination. This is beneficial for the incremental introduction of distribution to the programs. Consider as example an extension to the `receive-text` method of the chat service presented in Listing 4.2, in which this service signals the reception of text messages according to its location (e.g. playing a tone for "normal" locations and blinking lights for "discreet" locations). Listing 4.20 shows the redefinition of this method, now containing the code for the location-dependent signal.

```
(defmethod receive-text :interruptible ((receiver chat) sender text)
  (let ((message (make-text-message :from receiver to: sender :text text
                                    :timestamp (get-system-time)))
        (location (get-location receiver)))
    (if (discreet-location? location)
      (blink-lights receiver)
      (play-tone receiver))
    (display-text receiver sender message)
    (store-text receiver sender message)))

(defmethod get-location :interruptible (service)
  ... make asynchronous request to GeoIP service ... )
```

Listing 4.20: Definition of the `receive-text` method using uniform syntax.

In this definition, we use the `get-location` generic function which sends an asynchronous request to a GeoIP web service,[10] to obtain the geographic location associated to the IP address of the service's hosting device. Because Lambic handles such asynchronous invocations internally, this method can still be written in a sequential style. The test of the `if` form is evaluated only when the future returned by the `get-location` function (bound to the `location` variable) is resolved with the corresponding location. Also, as Lambic's uniform syntax ensures the sequential execution of method bodies, the invocation of the `display-text` and `store-text` generic functions are executed only after the execution of the `if` form. However, in this case there is no need for such sequentiality. In fact, the `if` form is the only part of the method concerned about the resolution of the future bound to the `location` variable, and we it would be better to express this situation by introducing a `when-resolved` form as follows:

```
(defmethod receive-text :interruptible ((receiver chat) sender text)
  (let ((message (make-text-message :from receiver to: sender :text text
                                    :timestamp (get-system-time))))
    (when-resolved (get-location receiver)
      (lambda (location)
        (if (discreet-location? location)
          (blink-lights receiver)
          (play-tone receiver))))
    (display-text receiver sender message)
    (store-text receiver sender message)))
```

Listing 4.21: Definition of the `receive-text` method using explicit syntax.

[10] In Lambic, we have developed a small layer for AJAX-based asynchronous communication with web services which associates a future to each AJAX request and internally resolves the future with a Lambic object that embodies the result returned by the request. See Section 8.4 for further details.

In this new definition, the `when-resolved` form makes explicit the part of the method that is asynchronously executed after the location is obtained. The use of explicit syntax is not always as straightforward as in this case, though, especially when there are data dependencies between the part of the method executed asynchronously and the rest of the method. For instance, assume that to reduce the number of requests to the GeoIP service, we provide means to memorise locations. Thus, the service is accessed only if the there is no cached location that corresponds to the requested IP. Note that in this case the location can be obtained either remotely (sending an asynchronous request to the GeoIP service) or locally (using a standard synchronous access to the cache). Using Lambic's uniform syntax we can define the caching functionality in the following (standard CLOS) *around* method for the `get-location` generic function:

```
(defmethod get-location :interruptible :around (service)
  (let ((location (get-cached-location service (get-system-ip))))
    (if (not location)
      (begin
        (setf location (try-catch (call-next-method)
                          (timeout-exception () nil)))
        (if location
          (cache-location service (get-system-ip) location))))
    location))
```

Listing 4.22: Definition of the `get-location` method using uniform syntax.

The `:around` annotation in the definition above ensures that this method is executed "around" all the other methods. This means that the code from the *around* method is run before the original `get-location` method (the one containing the request to the GeoIP service, outlined in Listing 4.20). The *around* method returns as result a location (bound to the `location` variable) which is first looked up in the local cache, and if it is not found, it is obtained by invoking the original `get-location` method (using the standard CLOS function for "super calls," `call-next-method`). Timeout exceptions for the request to the GeoIP service are handled by binding the `location` variable to `nil`. Thus, the execution of this method can end in the `location` variable bound to either: a cached location, a future asynchronously resolved with a location returned by the GeoIP service, or a future asynchronously resolved with `nil` as a result of handling the timeout exception. Still, Lambic's internal handling of futures enables this method to preserve its sequentiality despite the result returned and the way it is achieved. Using explicit syntax, however, these three cases would have to be manually handled in the method body, as shown in the Listing 4.23.

```
(defmethod get-location :interruptible :around (service)
  (let ((location (get-cached-location service (get-system-ip))))
    (if (not location)
      (let ((location-future (make-instance 'future)))
        (when-resolved (call-next-method)
          (lambda (location)
            (if location
              (cache-location service (get-system-ip) location))
            (resolve-with-result location-future location))
          :catch (timeout-exception ()
                   (resolve-with-result location-future nil)))
        location-future)
      location)))
```

Listing 4.23: Definition of the `get-location` method using explicit syntax.

In this definition, we introduce a `when-resolved` form to handle the future returned by the the request to the GeoIP service (represented by the invocation of `call-next-method`). In addition, this method can return as result a cached location (contained in the `location` variable), or a future (contained in the `location-future` variable) which is resolved by the `when-resolved` form, both inside of the lambda and the handler of the `timeout-exception` (with a location and with `nil` respectively). Needless to say, this method definition is clearer with regard to the kind of result it returns and how it is obtained. Yet, its control flow is partitioned and requires the manual definition and resolution of futures (e.g. the `location-future`). And these issues can become worse as more distributed context conditions (as the location) are taken into account to influence the programs behaviour, as shown in Chapter 7. For this reason, we opt for an incremental introduction of distribution. For all the case studies developed in this dissertation, we first present their implementation using Lambic's uniform syntax and then discuss the use of the explicit language abstractions for distribution.

4.5.6 Interoperability with Existing Libraries

A final concern when introducing support for distribution into a programming language, is to ensure proper interoperability with existing non-distributed libraries. In our case, this means to enable generic functions provided by those libraries to handle asynchronous remote invocations and return values (futures). As explained in Section 4.4.2, Lambic enables generic functions to be oblivious to the way they are invoked (locally or remotely). Similarly, our model transparently resolves or ruins the futures returned by remote invocations, with the result of the execution of the generic functions. Thus, the only requirement to remotely invoke a generic function of an existing library is to be able to identify the actor that contains that generic function, i.e. to have a reference to an object residing in the same actor which can be used as the actor designator argument of the remote invocation.[11] This invocation can then be specified using both explicit and uniform syntax. Listing 4.24 illustrates the remote invocation of the `get-horizontal-scroll-parameters` generic function provided by the Common Lisp graphical library called CAPI (Common Application Programmer's Interface [Ric90]). This generic function queries the scroll parameters of the horizontal scroll bar of an instance of the `simple-pane` class (also defined in the CAPI library).

```
(display "Remote editor's position of horizontal scroll: "
         (get-horizontal-scroll-parameters remote-editor :slug-position))
```

Listing 4.24: Remotely invoking a generic function of the CAPI library.

This listing shows the invocation of the `get-horizontal-scroll-parameters` generic function on a remote Geuze editor,[12], represented by the `remote-editor` argument. The second argument (`:slug-position`) indicates the parameter of the scroll bar to be returned.

Lambic's uniform syntax also requires that generic functions handle futures received as arguments. For generic functions defined in our model, we achieve this by adapting the standard (Common Lisp) semantics. However, it is often neither possible nor desirable to change this process for existing libraries. For these cases, our model enables developers to create a wrapper which implicitly handles the argument futures received by a generic function —waiting for their resolution, invoking the wrapped generic function with the futures' resolved values, and returning a future as result. Wrappers are created by means of the `futurise-functions` form which receives as arguments a list of symbols bound to generic functions. The listing below shows the use of this form for the generic functions of the CAPI library:

```
(futurise-functions '(get-horizontal-scroll-parameters
                      get-vertical-scroll-parameters ...))
```

Listing 4.25: Futurising generic functions.

The `futurise-functions` form creates a wrapper for the functions bound to the symbols included in the list argument. A wrapper corresponds to a futurised generic function which *shadows* the original function.[13] The futurised generic function has a method which invokes the original function in its body.

[11] By default, in our model each actor contains a copy of the libraries inherited from the Common Lisp system.

[12] A remote instance of the `Geuze` class which is a subclass of the `simple-pane` as shown later in Chapter 7.

[13] By *shadowing* we mean that the symbol is rebound to the futurised generic function. This operation has effect only in the Lambic environment which is represented as the `lambic-user` package.

4.6 Conclusion

In this chapter, we have introduced Lambic, our generic function-based object-oriented programming language model for AmOP. We have presented the first and most fundamental feature of our model concerning to support distribution, called *futurised generic functions*. Lambic successfully combines the actor-based event loops model with generic functions. In our model, actors define boundaries of concurrent execution for the objects; methods are specialised on objects (not on actors); inter-actor computations are realised by means of asynchronous generic function invocations; and the results of functions evaluations are asynchronously returned to the actor from which the functions are invoked. Therefore, the actor's message-sending semantics are explicitly separated from the programming level, which enables the programmers to invoke functions instead of sending direct messages to objects, and to define methods without receiver arguments.

Lambic provides explicit syntax for distribution, i.e. for asynchronous generic function invocations and for asynchronous result handling (futures). Additionally, our model provides an internal future-handling process that enables distributed computations to use the same syntax as for local computations, while internally still executing them in an asynchronous manner. In this chapter, we have illustrated the benefits and limitations of either approach (explicit and uniform syntax). Explicit syntax helps developers to better understand the effects of asynchronous remote interactions in the programs, but often entails more verbose code and less straightforward control flows. Uniform syntax enables the programs to be less verbose and to keep control flows sequential, but the support for dealing with remote interactions is more restricted.

Chapter 5

Context Dependency in Lambic

To provide linguistic support for encoding context-dependent behaviour, Lambic proposes a generic function-based multiple dispatching mechanism, called *predicated generic functions*. This mechanism provides abstractions to influence the method dispatch semantics based on the program's execution context. In this chapter, we introduce the syntax and semantics of predicated generic functions, and explain how this model complies with the requirements for context-dependent behaviour (cf. Section 3.4). We then discuss the integration of predicated generic functions into Lambic's model for ambient-oriented programming (cf. Chapter 4).

5.1 Predicated Generic Functions

Predicated generic functions are an extension of the generic function-based multiple-dispatching mechanism of CLOS. As explained in Section 4.1, CLOS supports multiple dispatch by detaching methods from classes, allowing developers to specialise methods on the classes of all the received arguments, as opposed to only the first argument in singly dispatched languages. We augment this mechanism by enabling methods to also specialise on predicates about the program's execution context. Predicated generic functions alleviate the restrictions to deal with method overriding ambiguities, of existing predicate dispatching models [EKC98, MFRW09] (cf. Section 3.2.1). Instead of requiring a logical implication order between predicates, this model fosters the definition of context-specific priorities. Predicated generic functions enable users to establish a priority-based order between logically unrelated predicates. These priorities are specified on a per-generic-function basis, as in the Filtered Dispatch model [CHVD08]. Predicated generic functions not only contain the methods with a common name and argument structure (as in standard generic function approaches [BDG$^+$88, CLCM00]), but also the predicates on which the methods can be specialised. A method is selected for execution when its predicate expression is satisfied, and the order of the predicates specified in a generic function determines the order of applicability of its methods. In the next section, we explain the syntax and informal semantics of predicated generic functions.

```
Generic function definition ::=
(defgeneric function-name parameter-list
            [(:predicates {pred-name})])                    → new-generic

parameter-list ::= ({parameter})

Method definition ::=
(defmethod method-name [execution-qualifier] [combination-qualifier]
            specialised-parameter-list
            [(:when {(pred-name arguments)})]
            body)                                            → new-method

execution-qualifier ::= :interruptible | :uninterruptible
combination-qualifier ::= :before | :around | :after
specialised-parameter-list ::= ({parameter | (parameter specialiser)})
```

Table 5.1: Predicate generic functions and methods in Lambic

5.1.1 Defining Predicated Generic Functions

Table 5.1 presents the syntax for predicated generic functions. It is an extension to the syntax of *futurised generic functions* presented in Table 4.5. As we explain in Section 5.3, the semantics of both models have been cleanly combined. In Table 5.1 we also include another central feature of CLOS —the use of combination qualifiers for methods— to illustrate the full integration of context predicates with the standard dispatching mechanism of generic functions (cf. Section 5.1.2).

Generic Functions with Context Predicates

A generic function is defined using the `defgeneric` construct which receives as arguments a name, a parameter list and an optional list of *context predicate* declarations. A context predicate is a method whose body is a boolean-valued expression about any information computationally accessible by the generic function. Standard (Common Lisp) functions for relational and arithmetic operations can also be used as predicates. Declaring a context predicate corresponds to including its name in the generic function's predicate list, denoted with the `:predicates` keyword. The predicates follow an arbitrary user-defined precedence order represented by the order of the predicate declarations. The last predicate of the list has precedence over the other predicates.

Predicated Methods

Methods are defined independently from their containing generic functions, using the `defmethod` construct. A method is defined with a name, a specialised parameter list and an optional predicate expression (specified with the `:when` keyword). This expression is composed of one or more invocations of the context predicates declared in the method's generic function. In particular, a method whose predicate expression contains several predicate invocations is said to depend on the conjunction of such invocations. Notice that this conjunction is implicit; no `and` operator is required in the predicate expression. The arguments passed to the context predicate invocations can refer to the parameters of the method, and also to literals and references to variables in scope. Two methods cannot use the same combination of context predicates.[1] In such a case, the newest definition replace the previous one.

Consider as an illustrative example of the use of predicated generic functions the definition of the factorial function shown in Listing 5.1. In this function we want to distinguish between negative and positive numbers, and the number zero. We therefore define a `factorial` generic function using as predicates the relational operations `<`, `=`, and `>`. Since these operations are already defined in Common Lisp, we just need to declare them as predicates for the `factorial` generic function, indicating the corresponding symbols as follows:

```
(defgeneric factorial (n)
  (:predicates < = >))

(defmethod factorial (n)
  (:when (> n 0))
  (* n (factorial (- n 1))))

(defmethod factorial (n)
  (:when (= n 0))
  1)

(defmethod factorial (n)
  (:when (< n 0))
  (error “Factorial not defined for negative numbers.”))
```

Listing 5.1: Definition of factorial using predicated generic functions.

Each of these methods uses one of the predicates declared in the generic function using the `:when` keyword. The first method is called if the argument `n` is a positive number and computes the general case of the factorial function. The second method is called if `n` is 0 and returns 1. The third method will be called if `n` is a negative number and signals an error.

Finally, as in CLOS, methods in Lambic can also be annotated with *combination qualifiers* [BDG+88]. Methods without such a qualifier are known as *primary* methods and are responsible for providing the primary implementation of a generic function. Methods with a combination qualifier are referred to as *auxiliary* methods, and provide a implementation complementary to primary methods. An auxiliary method is executed before,

[1] We further comment on this decision in Section 5.2.

after or around primary methods, depending on the qualifier it includes (`:before`, `:after` or `:around` respectively). In the next section, we explain the role of combination qualifiers in the dispatching mechanism of generic functions.

5.1.2 Invoking Predicated Generic Functions

When a predicated generic function is called with particular arguments, it must determine and apply the methods that are suitable to the context of the invocation. In CLOS, this process is known as building the *effective method* [BDG+88] and is normally driven by the class hierarchies in which the invocation's arguments are involved. We extended these semantics to build the effective method also taking into account the context predicates associated to the generic function's methods. Lambic follows CLOS' three-step procedure to determine the effective method [BDG+88]:

1. Select the methods that will handle the invocation (known as the *applicable methods*).
2. Sort the applicable methods according to their specificity.
3. Combine and apply the sorted list of applicable methods to the arguments received by the generic function.

In the remainder of this section, we describe Lambic's adaptation of these semantics for predicated generic functions. In this description we refer to predicated generic functions and generic functions interchangeably.

Context-dependent Selection of Applicable Methods

Given a generic function and a set of arguments, an applicable method is a method contained in that generic function which fulfils the following conditions:

- *The received arguments satisfy the method's parameter specialisers.* Let $a_{\{i=1..n\}}$ be the received arguments and $s_{\{i=1..n\}}$ be the parameter specialisers of a method M. The method M is applicable when a_i is an instance of class C such that $C = s_i$, or C is a subclass of s_i.
- *The invocation's execution context satisfies the method's predicate expression.* Let $\phi_{\{j=1..m\}}(r_{\{i=1..n\}})$ be the predicate expression associated with a method M, where ϕ_j denotes a context predicate and $r_{\{i=1..n\}}$ the arguments passed to such a predicate. Let E be the environment of M with all the bindings accessible by the invoked generic function (including the generic function's parameters bound to the invocation's arguments). The method M is applicable if each $r_i \in E$ and each $\phi_j(r_{\{i=1..n\}})$ evaluates to *true*.

The two conditions are checked in this order. The applicability of a method is determined first according to its parameter specialisers and then according to its predicate expression. A method whose parameters do not have specialisers, and that does not have any associated predicate expression, is referred to as the *default method* of the generic function. A default method is always applicable but may be shadowed by a more specific method. Finally, as in CLOS, if no methods apply for a given invocation, a `no-applicable-method` exception is thrown.

Predicate-based Sorting of Applicable Methods

Determining the specificity order among the applicable methods for a predicated generic function invocation is a two-step process. It implies comparing the methods first by their parameter specialisers (as in CLOS) and then by their predicate expressions.

Precedence by parameter specialisers To compare the precedence of two methods according to their parameter specialisers, the specialisers are examined one by one from left to right.[2] When a pair of parameter specialisers are equal, the next pair is compared. The first pair that differs determines the precedence order between the two methods. The more specific method is the one whose parameter specialiser appears earlier in the class hierarchy of the corresponding argument. If all the parameter specialisers of the two methods are equal, the methods are compared according to their predicate expressions.

Precedence by predicate expression To compare the precedence of two methods according to their predicate expressions, Lambic first sorts the context predicate invocations included in each expression. This sorting is based on the precedence order of the invoked predicates (declared in the generic function). The invocation to the predicate with highest precedence is positioned first while the invocation to the predicate with least precedence is last. Then, the invocations in the predicate expression of each method are examined one by one, from left to right, just as in the case of the evaluation of the parameter specialisers. When the compared pair of invocations refer to the same context predicate, the next pair is compared. The first pair of invocations that differ determines the precedence order between the methods. The more specific method is the one invoking the predicate with higher precedence. If all the context predicate invocations of the two methods are equal, the methods must have different combination qualifiers. In this case, either method can be selected to precede the other. This is because the eventual order will be unambiguously determined by the method combination in a later step.

Note that because the way in which applicable methods are chosen, the parameter specialisers are guaranteed to be present in the class hierarchy of the corresponding arguments. Similarly, the context predicates are guaranteed to be present in the list of context predicate declarations of the methods' generic function. At the end of this sorting step, the resulting list of applicable methods has the most specific method first and the least specific method last.

[2]Mechanism known as *lexicographic ordering* [BDG$^+$88].

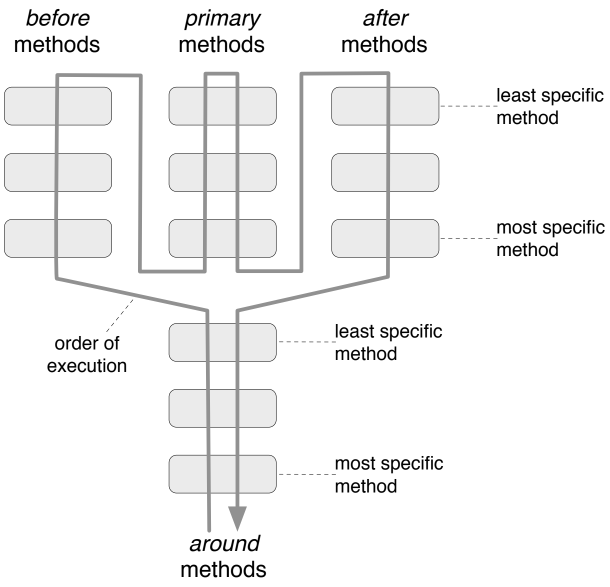


Figure 5.1: Standard method combination of applicable methods

Combining and Applying the Sorted Applicable Methods

Predicated generic functions preserve CLOS' *standard method combination* of applicable methods [BDG+88] (depicted in Figure 5.1). This mechanism combines the applicable methods into a single effective method. In case that there are only primary applicable methods (methods without combination qualifiers), the most specific method is executed first. In CLOS, such a method can invoke the next most specific method by means of the `call-next-method` function. This function is pretty much like a *super call* in singly dispatched object-oriented languages. If the `call-next-method` function is invoked and there is no next most specific method, a `no-next-method` exception is thrown.

In case of having auxiliary applicable methods, the following combination rules are applied:

- First, the most specific `:around` method is invoked, if any. An `:around` method may invoke `call-next-method` to execute the next most specific `:around` method.

- If there are no further `:around` methods, then all `:before` methods are executed, if any, with the most specific `:before` method being executed first, followed by all next most specific `:before` methods.[3]

[3] All `:before` and `:after` methods are executed implicitly. In such methods, invocations to `call-next-method` are not allowed [BDG+88].

- Next, the most specific primary method is invoked. A primary method may invoke `call-next-method` to execute the next most specific primary method.
- After returning from the primary methods, all `:after` methods are executed, if any, with the least specific `:after` method being executed first, followed by all next least specific `:after` methods in order.[3]
- Finally, execution returns to the remaining code to be executed in the `:around` methods, following the respective invocations of `call-next-method`.

To illustrate Lambic's semantics to determine the effective method, consider the definition of the predicated generic function `display-number-property`, shown in Listing 5.2. It displays the properties of a number which can be *prime* and at the same time *odd* or *even*. For this the generic function uses as predicates the built-in functions `primep`, `oddp`, `evenp` (all of them receiving a number as argument and checking whether the number is prime, odd or even respectively).

```
(defgeneric display-number-property (n)
  (:predicates primep oddp evenp))

(defmethod display-number-property ((n number))
  (display n " is:"))

(defmethod display-number-property :after ((n number))
  (:when (primep n))
  (display "- a prime number"))

(defmethod display-number-property :after ((n number))
  (:when (oddp n))
  (display "- an odd number"))

(defmethod display-number-property :after ((n number))
  (:when (evenp n))
  (display "- an even number"))
```

Listing 5.2: Definition of `display-number-property` using predicated generic functions.

Note that there is no logical order between these properties, as required by other predicate dispatching approaches. Yet, in Lambic developers can still define a precedence order among them. It is denoted by the order in which they are declared in the generic function. In the listing above, each of these properties is defined in an `:after` method which are executed after the primary method. Thus, given the invocation `(display-number-property 2)`, the effective method is determined as follows:

- The invocation's argument, `2`, satisfies the parameter specialiser of all the methods (`number`, which is used by all the methods). However, with regard to the second condition of applicability, the `oddp` predicate fails and thus the method associated with it is discarded. Only the other three methods remain applicable.

- In this example, the precedence order between the applicable methods cannot be determined according to the parameter specialisers. It has to be determined according to the precedence order between the context predicates. Because in the `display-number-property` generic function `primep` is declared before `evenp`, the resulting list of applicable methods has the method with the `evenp` predicate as the most specific, and the method without predicate as the least specific.
- Finally, based on the standard method combination of applicable methods described in this section, the list of applicable methods is re-organised as the method without predicates is primary. As such, it must be executed before the two `:after` methods.

The execution of the effective method for the invocation `(display-number-property 2)` will display the number properties in the following, unambiguously defined order:

```
2 is:
- a prime number
- an even number
```

5.2 Requirements for Context Dependency Revisited

We now review Lambic's predicated generic functions with respect to the requirements for context-dependent behaviour and existing solutions (cf. Sections 2.3 and 3.2 respectively).

5.2.1 Modularity

In Lambic, context-dependent adaptations are expressed as predicated methods —our model's units of *partial behaviour definition*. These methods can be specialised using programmer-defined context predicates, providing fine-grained control of method applicability, in a similar way to the Predicate Dispatch programming model (cf. Section 3.2.1). Additionally, method dispatch is driven by the context predicates' precedence order which is fully defined by the methods' generic function. Thus, context predicates can be cleanly associated with methods achieving *modular behaviour selection*. Lambic does not provide explicit means to group predicated methods. Instead, methods can be implicitly grouped by associating them with the same context predicate, as in Ambience [GMH07] and the Filtered Dispatch and Predicate Dispatch models (*implicit groups of partial definitions*).

To illustrate Lambic's modularity property, consider the development of a graphic user interface (GUI), such as the one of the drawing editor introduced in Section 2.1. In this scenario, the context of use plays a key role as it determines the operations of the editor that should handle the GUI events (as discussed in Section 2.3). For instance, the operation that should be executed upon encountering a *mouse-down* event depends on context information such as the coordinates of the mouse pointer, the shape found at those coordinates (if any), and the state of the editor's brush (whether the brush button is selected or not). Depending on this information, the operation corresponding to the *mouse-down* gesture might be moving, drawing, selecting and so forth.

Modular Partial Definitions of Behaviour. Listing 5.3 shows the implementation in Lambic of some of the actions that handle the *mouse-down* event in the drawing editor.[4] Using predicated generic functions, developers can cleanly separate such actions in methods specialised on predicates representing the different editor's operations. These methods act as *context-dependent* handlers which can conveniently associate actions with dynamically determined context conditions. For instance, the first method definition in Listing 5.3 describes how to handle the *mouse-down* event when the editor is in the *moving* context (indicated by means of the `moving?` predicate).

```
(defmethod mouse-down (editor shape x y)
  (:when (moving? shape editor))
  (set-drag-status editor x y))

(defmethod mouse-down (editor shape x y)
  (:when (drawing? shape editor))
  (set-line-status editor x y)
  (draw-point editor x y))

(defmethod mouse-down (editor shape x y)
  (:when (selecting? shape))
  (select-shape editor shape)
  (call-next-method))
```

Listing 5.3: Untangled GUI event handlers in Lambic. Application concerns are indicated on the right side.

Modular Behaviour Selection. All the methods in Listing 5.3 are contained in the `mouse-down` generic function which declares the predicates with the context conditions for the operations. Listing 5.4 shows the definition of this generic function and the context predicates, `moving?`, `drawing?` and `selecting?`, defined as methods. These predicates evaluate conditions such as whether the `shape` parameter has been bound to a non-nil value,[5] or if the editor's brush has been selected (`brush-active`).

[4] We fully develop this scenario in the validation chapter, Section 7.2.

[5] As in Common Lisp, Lambic's boolean symbol for true is `T`. Additionally, any non-nil value is also considered true. Thus, the condition on the `shape` parameter succeeds if it is bound to a non-nil value.

```
; The mouse-down generic function
(defgeneric mouse-down (editor shape x y)
  (:predicates moving? drawing? selecting?))

; Context predicates
(defmethod moving? (shape editor)
  (and shape (not (brush-active editor))))

(defmethod drawing? (shape editor)
  (and (not shape) (brush-active editor)))

(defmethod selecting? (shape)
  shape)
```

Listing 5.4: The `mouse-down` generic function.

Modular Groups of Partial Definitions. Lambic enables developers to modularise context-dependent adaptations that span several generic functions. For instance, in the drawing editor, the same operation can require a combination of GUI events. In Lambic, developers can implement this case using a State-like pattern [GHJV95]. Each operation represents a different editor's state, and each state groups the behaviour required by an operation to handle one or more GUI events. In our model, this corresponds to describing a state as a number of methods using the same predicate expression. A simple example is the state representing the operation for selecting shapes, which only requires one method definition to handle the *mouse-down* event. Another example is the *moving* state presented in Listing 5.5 which defines its behaviour in the methods `mouse-down` (to set a drag status used during the move), `mouse-move` (to move the shape) and `mouse-up` (to remove the drag status at the end of the move).

```
(defmethod mouse-down (editor shape x y)
  (:when (moving? shape editor))
  (set-drag-status editor x y))

(defmethod mouse-move (editor shape x y)
  (:when (moving? shape editor))
  (move-shape shape editor x y))

(defmethod mouse-up (editor shape x y)
  (:when (moving? shape editor))
  (delete-drag-status editor))
```

Listing 5.5: The *moving* state.

This way of specifying the behaviour of the drawing editor cleanly separates the definition of its several states (embodied by the predicates) from the behaviour corresponding to those states. Additionally, this solution avoids the drawbacks exhibited by the original state pattern, namely the identity issues introduced by representing the states as independent objects (typically referred to as the *object schizophrenia problem* [CSJR02]), and the fact that developers have to manually switch the state of the program. Because in Lambic the methods are separated from the classes, a particular editor can always retain its identity, no matter what state it is in. Also, since the state of the editor is automatically derived from the current context conditions, one does not have to worry about managing an explicit state with explicit state switches in the corresponding *mouse-down*, *mouse-move* and *mouse-up* event handlers.

Note that the State-like idiom in the previous example is mostly a naming convention for the predicate used to identify each operation of the drawing editor. It illustrates the fine-grained control at the developer's disposal to influence the applicability of methods based on the context.

5.2.2 Dynamic Selection

In Lambic, context-dependent adaptations represented as predicated methods are dynamically selected as a result of the method dispatch mechanism. This mechanism is driven by the predicates' precedence order declared in generic functions which avoids the problems caused by potential ambiguities when comparing arbitrary predicates that do not designate instance subsets of each other. Lambic alleviates the limitation of existing predicate dispatching models to include user-defined orderings of predicates for cases that cannot be decided solely on the basis of the structure of the predicates. Furthermore, automatic disambiguation of methods by means of logical implication (as originally proposed in Predicate Dispatch) does not always yield the desired semantics. For instance, in the drawing editor the conditions included in the `moving?` predicate are *stronger* than (imply) the condition of the `selecting?` predicate (cf. Listing 5.4). Predicate Dispatch will thus consider that the moving behaviour has precedence over the selection behaviour. However, in the editor the selection behaviour must be performed *before* the moving behaviour.[6]

In Lambic, the proper selection and combination of applicable methods is internally computed in accordance to the predicates and their order of declaration in the generic function. Hence, in the drawing editor, such a case is transparently handled by the `mouse-down` generic function, which selects the methods associated with the `selecting?` and `moving?` predicates for execution. Because the `selecting?` predicate has precedence over the `moving?` predicate (the latter predicate appears first in the list of predicates of the `mouse-down` generic function), the method for selecting the shape is executed first. Finally, as we explained in Section 5.1.2, by default only the most specific method is executed. Therefore, we need to invoke `call-next-method` in the method specialised on

[6]This requirement has to do with the distributed part of the scenario of the drawing editor, explained later in Section 7.2. The selection of a shape is used for control access. Selecting a shape means obtaining a lock from the leader that coordinates the interaction between remote editors.

the `selecting?` predicate, so that the one specialised on `moving?` is invoked next.

Uniqueness and Completeness. Prior predicate dispatching models promote the properties of uniqueness and completeness as the basis for their approaches. The former property ensures that for each invocation there is always a unique method which is more specific than any other. The latter ensures that for each invocation there is always at least one method which is applicable. Lambic ensures uniqueness but not completeness. The reason for this is that completeness requires a static analysis that would limit the expressivity of predicates. Ernst et al. [EKC98] allows predicates to include arbitrary expressions but they are treated as black boxes and the overriding relationship between two syntactically different expressions is considered ambiguous. In Lambic, completeness has to be manually ensured by defining a default unspecialised method without predicates.

5.2.3 Consistent Composition

Generic function-based priorities between predicates ensure a consistent composition of methods, alleviating the issue of combinatorial explosion of context-dependent adaptations described in Section 2.3. This scheme enables developers only to deal with the context that is relevant to the task represented by the predicated generic function. Furthermore, developers do not have to manually determine the combination of methods most appropriate for the context of an invocation, as it is internally computed according to the order of the predicates in the generic function.

5.2.4 Restricted Scope

Lambic delimits a context-dependent adaptation to the scope of a generic function invocation, as do most models for context-oriented programming (cf. Section 3.2.1). Upon every invocation the context predicates are evaluated ensuring that the selected methods are always consistent with the current context conditions. This property is preserved even in the presence of concurrency and distribution, as we explain in Section 5.3 when discussing the combination between futurised and predicated generic functions.

5.2.5 Limitations

Although Lambic can help in tackling some of the challenges for modelling generic functions with context-specific predicates, a number of challenging issues need to be further explored as part of future work.

Efficiency. We have not considered efficiency issues in detail yet. However, efficient implementation techniques for generalised Predicate Dispatch have been investigated in detail in the past [EKC98] (e.g. to cache the results of predicate evaluations), and can probably be adapted to the implementation of predicated generic functions as well.

Logic Implication as a Complementary Approach. In Lambic, we propose an alternative to logical implication order used by existing predicate dispatching approaches to disambiguate method overriding. However, there are situations in which logical implication order would still be desired, e.g. to disambiguate methods using the same predicate expression. For instance, using predicated generic functions the method definitions would lead to an ambiguous situation:

```
(defmethod foo (n)
  (:when (> n 1))
  (print “Number greater than 1”))

(defmethod foo (n)
  (:when (> n 2))
  (print “Number greater than 2”))
```

If `foo` is invoked with `n` greater than 2, as both methods would be selected for execution but none of them is more specific than the other. While Lambic avoids this problem by not allowing the definition of methods with the same predicate, this is clearly a case in which the inclusion of logical implication in Lambic would increase its expressiveness.

Predicates as Filters. Predicated generic functions build on previous work on Filtered Dispatch [CHVD08]. Similar to Lambic's predicates, filters are associated with generic functions. There are however significant differences in both approaches. Firstly, even though many filters can be defined for a given generic function in Filtered Dispatch, corresponding methods can use only one of those filters at a time. As a consequence, each possible combination of the filters that could prove useful needs to be anticipated and encoded as an additional filter in the generic function. Secondly, filtered expressions are parameterised exclusively on the argument they filter; they cannot depend on the value of other arguments of the method. This restriction renders Filtered Dispatch less amenable to express context adaptations, because the conditions for applicability (the predicates) cannot harness all available contextual information. On the other hand, the example where an interpreter dispatches on unwrapped interpreter values [HCD08], seems to be less straightforward to do in Lambic. In this interpreter, values of the interpreted language are represented in the standard way, as wrapped values in the implementation. However, the object-oriented implementation of the interpreter needs to dispatch on the unwrapped values, not on the wrappers themselves. Using Filtered Dispatch, this can easily be expressed by turning the unwrap function into a filter. Another typical use case of Filtered Dispatch is to use a predicate as a filter and specialising the method on the (boolean) return value of that predicate.

5.3 Predicated Generic Functions in AmOP

Now that we have presented predicated generic functions, we discuss its integration with Lambic's manifestation of the ambient-oriented programming paradigm, i.e. with futurised generic functions (cf. Section 4.2). As discussed in Section 3.4, the combination of these two concerns implies dealing with the propagation of asynchrony and network failure handling, and the consistency of the scope of activation of context-dependent behaviour. In Lambic, we can effectively cope with these issues by using the internal future-handling semantics promoted by futurised generic functions. Because in these semantics distributed computations use the same syntax of local computations, remote results can be handled implicitly, and network failures can be tackled using standard exception handling forms. We then enable Lambic's predicate-based dispatch mechanism to support futures as results of the evaluation of context predicates. In the remainder of this section, we discuss the combined semantics for futurised and predicated generic functions, which fulfil the above requirements.

5.3.1 Combining Futurised and Predicated Generic Functions

We analyse the combination of semantics according to the definition and invocation of generic functions. This combination uses the syntax presented in Table 5.1 (Section 5.1.1).

Combined Definition of Futurised and Predicated Generic Functions

The syntax and semantics for defining futurised and predicated generic functions can be straightforwardly combined. This is mainly due to the fact that only predicated generic functions require a syntactic extension to the standard definition of generic functions, for the list of context predicate declarations. This list does not have any impact on the semantics of futurised generic functions.

Combined Definition of Futurised and Predicated Methods

When combining futurised and predicated methods, a number of interactions should be taken into account:

Context predicates and execution qualifiers A method definition can include both an execution qualifier (`:interruptible` or `:uninterruptible`) and a predicate expression. The qualifier does not affect the evaluation of the method's predicate expression. The *interruptible* and *uninterruptible* execution conditions only apply to the evaluation of the methods' body, not to the process of determining the effective method, which is where the evaluation of the context predicates occur. If such execution conditions are required for evaluating the predicates, the methods representing the predicates should be defined with an execution qualifier.

Controlled propagation of asynchrony through *super calls* Invocations to `call-next-method` can return a future as a result. For instance, this can occur when the method invoked through such a form contains an asynchronous invocation of a remote generic function. However, because in Lambic futures can be implicitly processed, the method invoking `call-next-method` needs no special language support to handle such a case. Computations that depend on the resulting future are suspended until the future is resolved, as defined in the futurised method execution semantics presented in Section 4.4.2.

Controlled propagation of asynchronous exceptions through *super calls* To allow the `call-next-method` form to return futures also implies that methods invoking such a form may have to deal with eventual asynchronous exceptions ruining the futures. This is the case of timeout exceptions affecting remote generic function invocations (cf. Section 4.4.3). Lambic does not prevent the propagation of exceptions through super calls. As such, developers can arbitrarily choose the place to handle the exceptions that best suits the application logic. Note that exceptions —including those affecting the futures— can be handled with our model's *futurised* `try-catch` form. No special callbacks are required for this case. This enables method definitions to preserve their sequential and imperative programming style, independently of the place where exceptions are handled.

Definition of Futurised Context Predicates

Context predicates can return a future as a result. Its resolution is implicitly awaited by the method dispatch mechanism of our model, as discussed in the next section. Context predicates must be defined in the same actor as the generic function that uses them. Remote methods cannot be directly used as context predicates. If a remote method is required, e.g. to evaluate a remote context condition, it has to be invoked within a locally-defined predicate. This way we ensure that eventual network failures affecting the remote invocation can be properly handled. A context predicate should not have any observable effect. Thus, network failures and any other kind of exception cannot be handled inside the predicate's body (using a `try-catch` form). No propagation of exceptions from the context predicates to the generic functions is allowed. In case that a future returned by a predicate is ruined, Lambic's dispatch mechanism will raise the corresponding exception.

Combined Execution of Futurised and Predicated Generic Functions

To support futures while building the effective method for a generic function invocation, only the step for the selection of methods needs to be considered (cf. Section 5.1.2). In this step, the context predicates are evaluated and in case that they return futures, their final result is awaited. Waiting for the resolution of such futures will suspend the method dispatch process for the current invocation. At this stage, the only way to avoid that further invocations are interleaved with the current execution, is to annotate the predicate definitions with the `:uninterruptible` qualifier.

5.3.2 Discussion

We evaluate Lambic's solution with respect to the requirements for combining models for event-driven communication of AmOP and for context-dependent behaviour (cf. Section 3.2.3): controlled propagation of asynchrony and network failures, and consistent activation scope of event-driven behaviour.

Controlled Propagation of Asynchrony and Network Failures

In Section 3.2.3, we concluded that the language support required to handle asynchronous interactions and network failures should not hinder the semantics of dynamic method dispatch and inheritance-based composition. Lambic complies with this requirements by enabling the controlled propagation of asynchrony from both evaluation of context predicates and super calls. This is possible thanks to our model's implicit handling of futures and its integration with the process to determine the effective method for invocations. Our model also allows asynchronous exceptions to be propagated through super calls (in the form of ruined futures). However, the exceptions raised during the evaluation of context predicates cannot be propagated to the generic functions. Exceptions must be fully handled in predicates. Our model does not require the use of explicit syntax to handle asynchronous results and exceptions.

To illustrate the propagation of asynchrony and exceptions, consider the following implementation of the messenger example in Lambic, introduced in Listing 3.1 of Section 3.2.3.

```
; receive-call generic function
(defgeneric receive-call (caller callee)
  (:predicates is-urgent-call?))

(defmethod receive-call ((caller messenger) callee)
  ; default behaviour...
  )

(defmethod receive-call ((caller messenger) callee)
  (:when (is-urgent-call? callee))
  ; urgent behaviour...
  )

; Context predicate
(defmethod is-urgent-call? (callee)
  (try-catch
    (is-current-location? callee “hospital”)
    (timeout-exception () nil)))
```

Listing 5.6: Lambic's propagation of asynchrony and exceptions.

The communicator's operation to handle incoming calls is defined by the `receive-call` generic function. Apart from its default behaviour (represented by the first method definition), this generic function contains a method specialised on the `is-urgent-call?` context predicate. This predicate invokes the remote generic function `is-current-location?`. Developers are required to enclose such a remote invocation with a `try-catch` form to handle any `timeout-exception` raised by Lambic's network failure handling mechanism. As such, the predicate returns as a result the future generated by the `try-catch` expression (and not the one of the remote invocation). Note that in this case the future returned by `try-catch` will not be ruined. It will be resolved either with the result of the remote invocation, or with the return value of the handler of the timeout exception (`nil` in this case). Note also that in the example above, no explicit support is required to handle the asynchronous result of the remote invocation and the `try-catch` form.

Consistent Activation Scope of Event-driven Behaviour

A programming language model should provide a consistent scope for context-dependent adaptations, even in the presence of concurrent and interleaved event-driven interactions (cf. Section 3.4). In Lambic the scope of a context-dependent adaptation is delimited by the execution of a generic function invocation, and that the adaptation corresponds to determining the effective method for each invocation. Additionally, in Section 4.3 we declare that generic function invocations are sequentially processed, exclusively by the event loop of the actor containing the generic functions. This means that, at any moment in time, within an actor, there can be only one context-dependent adaptation in use. This condition aims at preventing adaptations required for different invocations to conflict with each other. However, it may still be insufficient when the execution of the invocations are allowed to be interleaved (as discussed in Section 3.1.2). Event interleavings have non-evident effects on the selection of the effective method for the interleaved invocations, in particular in the evaluation of the predicates associated with the generic functions.

To illustrate this issue, consider the case of a shared bank account (an account that can be accessed by several users). Listing 5.7 shows a basic definition of the operation for withdrawing money, represented by the `withdraw` generic function. The `withdraw` generic function has two methods. The first method is specialised on the `is-empty?` predicate, indicating that this method should be executed when the shared account is empty. The second method is the default behaviour of the withdraw operation. This method effectively accomplishes the withdraw (e.g. decrementing the money from the account), only after the account's pin code is confirmed at the ATM making the request. In the example, this corresponds to replying to the `confirm-pincode` invocation. Note that because this invocation is asynchronous, the execution of the `withdraw` method is suspended until the invocation's generated future receives the result. Thus, the execution of the withdraw operation can get interleaved with other requests, as shown in Figure 5.2. Assume that during the execution of an invocation to `withdraw` made by ATM_1, there is a second request to the same function, now coming from ATM_2.

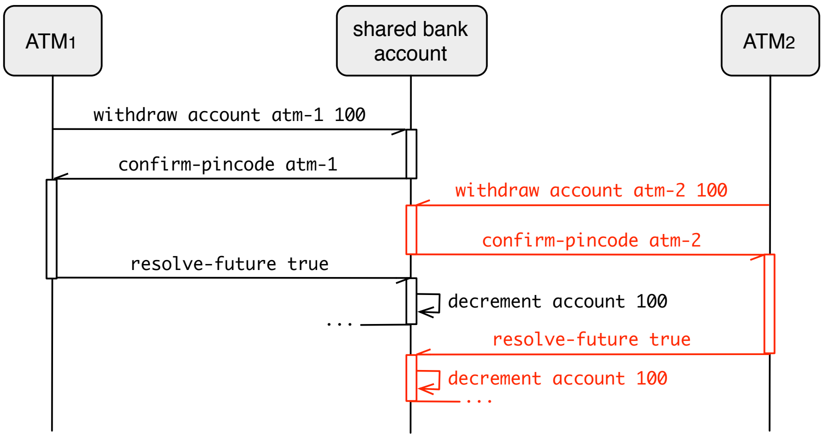


Figure 5.2: Predicate evaluation and invocation interleaving.

```
; Definition of withdraw generic function.
(defgeneric withdraw (account atm amount)
  (:predicates is-empty?))

(defmethod withdraw ((account shared-account) atm amount)
  (:when (is-empty? account))
  (receive-message atm “The account is empty.”))

(defmethod withdraw ((account shared-account) atm amount)
  (if (confirm-pincode atm)
    (decrement account amount) ...))

; Definition of is-empty predicate.
(defmethod is-empty? ((account shared-account))
  (<= (balance account) 0))
```

Listing 5.7: Partial definition of a shared bank account.

Lambic's event loop model ensures that the shared bank account properly executes only one invocation at a time.[7] However, the problem occurs when the process to determine the effective method for the second invocation evaluates the `is-empty?` predicate. Since at that moment the first invocation has not withdrawn the money yet (it only happens after the asynchronous confirmation of the pin code), the evaluation of the predicate for the second invocation will consider the current balance of the account, and not the

[7]In this case, we also assume that the shared bank account and the ATMs are running in different hosts, hence they are contained in different actors.

one after the execution of the first invocation has been completed.

In Lambic, we solve this problem by annotating the second `withdraw` method definition in Listing 5.7 with the `:uninterruptible` qualifier. This ensures that the executions of the two invocations are not interleaved. Thus, the right method definition can be adopted for each case.

5.4 Conclusion

In this chapter, we have introduced *predicated generic functions*, a novel mechanism to allow flexible behaviour selection based on dynamically determined context information. This mechanism has been realised as an extension of the method dispatching mechanism of Common Lisp Object System (CLOS). Lambic method definitions can be guarded by predicates, which are used to decide on the applicability of the method for a list of actual arguments. If more than one predicated method is applicable, the order in which the predicates are declared in the corresponding generic function is used as a tiebreaker. These are the main tools Lambic offers for fine-grained control of applicability and specificity of methods.

Predicated generic functions have been successfully combined with Lambic's support for event-driven distribution, futurised generic functions. As such, method selection and composition can handle asynchronous results and exceptions of remote generic function invocations.

Chapter 6

Group Behaviour in Lambic

We now introduce the third and last component of Lambic. This is our approach to model group behaviour in pervasive computing, called *group generic functions*. Group generic functions propose a novel way to deal with service groups in AmOP. In this chapter, we explain the rationale behind this approach by means of a behavioural pattern, called *empathic group behaviour*. We then present the syntax and semantics for group generic functions and explain how they comply with the requirements for group behaviour, identified in Section 3.4. Finally, we describe the role of the other features of Lambic (futurised and predicated generic functions) in the accomplishment of *empathic* object-oriented group behaviour.

6.1 An Empathic Approach to Group Behaviour

The metaphor we use to explain our approach for group behaviour in ambient-oriented programming is the social conduct, known as *empathy*. When a person talks to somebody in a public space, all the co-located people found at such place can hear, and possibly understand, what the speaker says. However, the context of the conversation makes those people realise that the speaker is addressing only one of them, e.g. because of the speaker's body language or the content of the conversation. In other words, by sharing a common nature (e.g. a common sensory and reasoning system) people are able to *receive* the speaker's message, *contextualise* it (reason about the context in which it occurs), and *react* accordingly. And this occurs despite the speaker's intention to address his message to a specific person.

The previous scenario hints at a recent study made in the field of neuroscience about a particular kind of neurons (called *mirror neurons* [RC04]), responsible for people's *empathic behaviour*. Such neurons can fire an event in response to a stimulus, e.g. a person touching one's hand, and also when one observes the stimulus being produced to somebody else, e.g. a person touching somebody else's hand. Thus, one can *empathise* with the other person being touched. However, to prevent one from actually experiencing the touch sensation merely by observing, receptors in the skin send a feedback signal to the brain that vetoes the signal of the mirror neuron, informing that one has not been touched.

Note that this case follows a pattern similar to the previous example of the human conversation. When a person is touched, nearby people can also *receive* such a stimulus, *contextualise* it (with the help of skin receptors), and *react* accordingly. Again this occurs regardless of the fact that the stimulus is produced for only one person [RC04]. We call this behavioural pattern *empathic group behaviour*.

This pattern conveniently fits the description of pervasive computing services presented in Section 2.1. As stated there, software entities found in the same environment and sharing common attributes (similar functionality, and possibly, similar state), are expected to work together as a single *pervasive service*. However, this grouping should not affect the interactions between the entities and their clients. A client should still be able to address its requests to an entity of the pervasive service, in the same way as when the entity worked independently (e.g. using the same identity, interface and communication style). Then, to ensure the group behaviour of the pervasive service, it should be possible that all its member entities can receive the requests, even when the requests are not explicitly sent to each of them. That is, the entities can *empathise* with the one being requested. Finally, each entity should decide how to respond to the requests based on the context in which they occur. Because such context can change from one entity to another, they should be able to react differently to the same request.

In summary, when a client sends a request to a member of a pervasive service, the other members should implicitly also receive the request, contextualise it, and react accordingly. The basic requirement to achieve such empathic behaviour is that software entities share a *common characteristic* (location, behaviour, and possibly, state). In Chapter 7, we further discuss the suitability of the empathic group behaviour pattern, when implementing the scenarios of pervasive computing presented in Section 2.1. In what follows, we present the *group generic function* model, the manifestation of empathic group behaviour in Lambic.

6.2 Group Generic Functions

In Lambic, we implement the empathic group behaviour pattern pattern in a programming model, called *group generic functions*. This model combines the multiple dispatch semantics of generic functions with two main linguistic abstractions, *group classes* and *group methods*:

- A group class represents the common characteristic that enables objects to *empathise* with each other. Besides allowing for structure and behaviour sharing (as standard classes), a group class enables its instances to be aware of the requests each other receives. Generic function invocations using such an instance as argument can be implicitly propagated to the other instances of the group class. This way, the instances can cooperate in the execution of the invocations, while the programs making the invocations can be oblivious to such a cooperation.

- Additionally, the group behaviour required to handle the invocations is declaratively specified in special methods of the group generic functions, called group methods.

Group methods enable the decoupling of the group behaviour from the base functionality of group classes, which is still defined in standard (CLOS) methods.

To have a quick idea of how Lambic's group generic functions work, consider the following example of the scenario of the drawing editor (cf. Section 2.1). Assume that the editor allows a multiple selection of shapes. Operations addressed to one of the selected shapes (e.g. a user moving a shape) should implicitly also affect the others (e.g. moving along with the addressed shape). In Lambic, we can express this feature by defining the shapes as a group class and their operations as group generic functions. The listing below sketches such definitions:

```
; Definition of the group-shape group class
(defgroupclass group-shape ()
  ((selected :reader is-selected? ... ) ... ))

; Definition of the move-shape group generic function
(defgroupgeneric move-shape (shape editor x y))

; Definition of the move-shape method
(defmethod move-shape ((shape group-shape) editor x y)
  ; standard move behaviour...
)

; Definition of the move-shape group method
(defgroupmethod move-shape ((shape group-shape) editor x y)
  (:propagate ((is-selected? shape)))
  ... (call-next-method) ... )
```

Listing 6.1: Group generic functions in Lambic.

Listing 6.1 shows the definition of the `group-shape` group class and the `move-shape` group generic function (representing the shape's *move* operation). The base logic of the group generic function is described in a standard method (using the `defmethod` form). Its group behaviour is defined in a group method (using the `defgroupmethod` form). The group method contains a *propagation expression* denoted with the `:propagate` keyword, which declaratively specifies that invocations to `move-shape` should be propagated to all the "selected" shapes. That is, to all the instances of the `group-shape` class which satisfy an `is-selected?` predicate.[1] The propagation occurs only when the group method invokes the `call-next-method` form. Because group methods are always executed before standard methods, the evaluation of such a form results in multiple invocations to the standard method of `move-shape`, each time passing a different selected shape in the `shape` argument.

Note that since the group behaviour of the shapes is fully encapsulated in group generic functions, clients need no special group abstractions to interact with them (e.g. group

[1]In this example, this corresponds to the accessor function defined for the `selected` field of the `group-shape` class.

Group class definition ::=
(`defgroupclass` *class-name* (*parent-class*)({*field*})) → *new-groupclass*

Group generic function definition ::=
(`defgroupgeneric` *function-name parameter-list*) → *new-groupgeneric*

parameter-list ::= ({*parameter*})

Group method definition ::=

```
(defgroupmethod method-name [combination-qualifier]
                specialised-parameter-list
                propagation-expression
                body)                                   → new-groupmethod
```

combination-qualifier ::= `:before` | `:around` | `:after`
specialised-parameter-list ::= ({*parameter* | (*parameter specialiser*)})
propagation-expression ::= (`:propagate` ({(*pred-name arguments*)})
[*catch*] [*return*])
catch ::= `:catch` {(*exception-name* ([*argument*])*body*)}
return ::= `:return` ⟨`nil` | *user-defined-function*⟩

Peer method definition ::=

```
(defmethod method-name [combination-qualifier]
           specialised-parameter-list
           body)                                        → new-method
```

Table 6.1: Group generic functions in Lambic

identities, group interfaces or group invocations). A client can address its requests directly at one shape, invoking a group generic function specialised on the `group-shape` group class. Furthermore, the group behaviour is contained in group methods which ensures its modularity and explicit separation from the base functionality of the shapes.

In the remainder of this section, we further explain the definition and execution semantics of group generic functions. Table 6.1 presents the syntax of this model. Group generic functions have been conveniently integrated with the other features of Lambic (futurised and predicated generic functions). We now focus exclusively on the details concerning group generic functions, and defer the discussion about the integration to Section 6.4. This also means that we first describe group generic functions in a local setting, and then discuss their application to a distributed setting (when combining group and futurised generic functions in Section 6.4.1).

6.2.1 Defining Group Classes

As standard classes in CLOS, a group class determines the structure and behaviour of a set of objects (cf. Section 4.1). It is defined in exactly the same way as a CLOS class (with a name, a list of superclasses, and a list of attributes), but with the `defgroupclass` form instead. The main difference is that a group class keeps explicit record of all its instances. It stores each instance created using the `make-instance` form. Such an explicit record is the basis for the definition of empathic behaviour, as we show in the next section. Because of the capacity to perform actions together, we refer to the instances of a group class as *peers*.

A group class can inherit structure and behaviour from both standard and group classes. However, standard classes cannot inherit from group classes.[2] The subclasses of a group class are also group classes. For this reason, Lambic requires that they are defined exclusively using the `defgroupclass` form. This way we ensure the explicit distinction between classes and group classes in the programs. This scheme is similar in spirit to the dissemination of execution qualifiers through method definitions (cf. Section 4.4.2).

Note that by enabling inheritance between group classes, we also allow objects to be part of several (nested) groups. Thus, for a group class G_1 with a super group class G_0, the peers of an object as instance of G_1 are a subset of the peers of the object as instance of G_0.

6.2.2 Defining Group Generic Functions

Operations for group classes can be defined in standard generic functions. However, the operations that require the group behaviour of the group classes have to be defined exclusively by means of group generic functions. As with standard generic functions, a group generic function is a container of methods with a common name and a parameter list the methods can specialise. It is defined with the `defgroupgeneric` construct indicating the name and the parameter list. Methods are defined separately using the same syntax and semantics of CLOS methods. They describe the base functionality for the group classes, i.e. the behaviour executed independently for a peer instance of a group class. For this reason, we refer to such methods as *peer methods*. Peer methods represent the *peer level* definition of the group generic function. Additionally, a group generic function has a *group level* definition required to specify coordination schemes between the group class instances. Such specification is contained in *group methods*.

6.2.3 Defining Group Methods

Group methods are defined with a dedicated `defgroupmethod` form. This form receives a name, an optional combination qualifier, and a specialised parameter list. A group method has one or more parameters specialised on group classes. For such group classes, the group method provides a *propagation expression* to declaratively specify how to propagate invocations through their instances. This expression is denoted with the `:propagate`

[2]This situation is signalled as an error during the definition of the class.

keyword, and consists of one or more predicates on the group classes. The invocations are propagated to all the instances of the group classes that satisfy the predicates. We further explain the propagation process in the next section, when discussing the execution semantics of group generic functions.

Propagation predicates follow the same definition semantics as context predicates, presented in Section 5.1.1. They can be modelled as methods. Exceptions raised during their evaluation have to be handled within their body.

In addition to the predicates, a propagation expression can contain the following optional specifications:

Handling exceptions during propagation The propagation expression can also specify how to handle exceptions raised during the propagation. That is, exceptions that occur while executing each of the invocations resulting from the predicate evaluation process. The handlers are specified using the `:catch` argument which works in a similar way to the `try-catch` form (cf. Section 4.4.2). It receives one or more handler definitions indicating a name of an exception, an optional parameter containing the exception, and a body. The result of the execution of the body takes the place of the return value of the failed invocation.

Handling propagation results The propagation expression also enables developers to determine the return value of the propagation. This is done using the `:return` argument. It receives a function that specifies how to handle the results of the invocations originated by the propagation. The function is parameterised with the list of results of the invocations. Our model provides the `first` and `all` functions to indicate that the final result of the propagation should correspond to the *first* obtained result, or to *all* of them, respectively. Additional functions can be defined. Alternatively, the `:return` argument can be set to `nil` to indicate that no results are expected from the invocations. In this case, the return value of the propagation is `nil`. By default, the `:return` argument is set to `first`.

In a group method, the propagation expression determines the evaluation semantics of the `call-next-method` function. This means that the propagation occurs only if the group method invokes such a function in its body. The execution of `call-next-method` potentially results in multiple invocations to the next most specific method of the group generic function. The return value of this function corresponds to what is specified in the `:return` argument of the propagation expression.

Finally, a group generic function can have more than one group method. This is possible by defining group methods with different parameter specialisers, or by using *auxiliary group methods* (group methods annotated with the combination qualifiers `:before`, `:after` and `:around`). In the case when two group methods are defined with the same parameter specialisers and combination qualifier, the newly created group method overrides the other, even if they have different propagation expressions. Auxiliary group methods cannot provide a propagation expression.

Listing 6.2 shows the use of Lambic's group abstractions in another variation of the multiple selection of shapes, presented at the beginning of this section. Assume an operation that calculates the area of a shape. In case that several shapes are selected, the operation returns the total area of all of them.

```
; GROUP CLASSES
; Definition of group class for rectangles
(defgroupclass group-rectangle (group-shape)
  ((width :accessor width)
   (height :accessor height)))

; Definition of group class for circles
(defgroupclass group-circle (group-shape)
  ((radius :accessor radius)))

; GROUP GENERIC FUNCTION
; Definition of get-area group generic function
(defgroupgeneric get-area (shape editor))

; PEER LEVEL DEFINITION
; Definition of get-area peer method for rectangles
(defmethod get-area ((rectangle group-rectangle) editor)
  (* (width rectangle) (height rectangle)))

; Definition of get-area peer method for circles
(defmethod get-area ((circle group-circle) editor)
  (* pi (expt (radius circle) 2)))

; GROUP LEVEL DEFINITION
; Definition of get-area group method for shapes
(defgroupmethod get-area ((shape group-shape) editor)
  (:propagate ((is-selected? shape)) :return #'total-area)
  (call-next-method))

; Definition of reduction method calculating the total area
(defmethod total-area (areas)
  (reduce #'+ areas))
```

Listing 6.2: `get-area` group generic function.

We implement such an operation as a `get-area` group generic function. It has two peer methods computing the area of two different kinds of shapes, rectangles and circles. They are represented by the group classes `group-rectangle` and `group-circle`, both subclasses of `group-shape`. Such methods correspond to the *peer level* definition of the `get-area` group generic function. Additionally, this group generic function has a *group level* definition composed of a group method propagating the invocations to all the selected shapes, and adding their results. This is expressed in the group method's propagation expression using the `is-selected?` predicate (as explained at the beginning of this section), and the `total-area` method in the `:return` argument.[3] In this case, the body of the group method contains only the `call-next-method` form.[4] Note that in the example the group method specialises the `shape` parameter on the `group-shape` group class (and not on `group-rectangle` or `group-circle`). As such, we ensure that the propagation comprises both rectangles and circles.

6.2.4 Invoking Group Generic Functions

To process an invocation of a group generic function, our model first computes the generic function's group level. This involves exclusively the evaluation of the group methods. The group level resolves in one or more invocations of the peer level behaviour of the group generic function. This includes only the evaluation of the peer methods. Because these methods correspond to standard CLOS methods, they are executed using the standard semantics of generic functions. In this section, we focus on the evaluation of the group methods.

To process the group level of a group generic function, we extended CLOS' semantics for building the effective method [BDG+88] with the following steps:

1. Selecting and sorting the applicable group methods.
2. Determining the effective propagation.
3. Combining and applying selected group methods.

In what follows, we review each of these steps in detail.

Selecting and Sorting Applicable Group Methods

Lambic determines the applicability of group methods using CLOS' mechanism to select methods [BDG+88]. Given a group generic function invocation, a group method is applicable if its parameter specialisers correspond to the classes or superclasses of the received arguments. In case that none of the group methods comply with this condition, the invocation to the group generic function is handled using the standard execution semantics of CLOS generic functions (cf. Section 5.1.2). The same selection process is applied to the auxiliary group methods, if any.

The selected group methods are then sorted using CLOS' lexicographic ordering algorithm [BDG+88] (cf. Section 5.1.2). The precedence between two group methods is determined by the precedence between their parameter specialisers, which are by default compared from left to right. As a result of this step, the list of applicable group methods is sorted from most to least specific.

[3]In Lambic, as in CLOS, The notation `#'` is an abbreviation for the `function` operator. This operator returns the functional value associated with a name in the current lexical environment [BDG+88].

[4]By default the propagation of an invocation returns all the results. This means that alternatively the group method of this example can be defined without an explicit `:return` argument, and computing the total area within its body. Still, we keep the other definition mainly to illustrate the use of the `:return` argument.

Determining the Effective Propagation

Once a group method has been selected, the predicates of the propagation expression are evaluated. This is a two-step process in which we first compute the *most general* propagation of the group method, and then determine the *effective* propagation:

- Determining the *most general* propagation for an invocation corresponds to making a list of all the possible invocations according to the group classes specialising the group method. For a group method with a parameter specialised on a group class, this means creating as many invocations as peers of the group class are available in the execution environment. For each invocation, we replace the peer originally passed as argument parameter with a different peer of the group class.
- To each invocation of the list resulting from the previous step, our model applies the predicates defined in the propagation expression of the group method. The subset of invocations whose arguments satisfy the predicates becomes what we call the *effective propagation* of the group method. If no invocation satisfies the predicates, our model signals a `no-applicable-propagation` error.

In the case when the group method does not specialise any parameter on a group class, the propagation expression is evaluated only on the arguments received in the invocations. Conversely, when the group method has several parameters specialised on group classes, the most general propagation is obtained by creating an invocation for every combination between the peers of the different group classes. Each invocation uses a different combination of peers as arguments. Then, as before, the effective propagation results from applying the predicates to all the invocations.

Combining and Applying Selected Group and Peer Methods

Finally, the group generic function combines all the applicable group methods into a single effective method. In case that there are no auxiliary group methods, the effective method corresponds to the most specific group method. The group method performs its effective propagation by calling the `call-next-method` function. Each invocation included in the effective propagation is handled with the next most specific group method. Only for the least specific group method the invocations are handled with the peer level of the group generic function. The results of the propagation are handled using the function passed as the `:return` argument in the propagation expression of the group method.

In case of having auxiliary group methods, CLOS' combination rules are applied (cf. Section 5.1.2). That is, the *around* group methods are executed first. Invoking `call-next-method` in its body will cause the sequential execution of *before* group methods, primary group methods (including its effective propagation) and *after* group methods.

Figure 6.1 illustrates how to determine the effective method for an invocation of the `get-area` group generic function (cf. Listing 6.2). Assume that at the moment of the invocation, there exist four shapes: two instances of the `group-square` group class, `square-1` and `square-2`, and two instances of the `group-circle` group class, `circle-1` and `circle-2`.

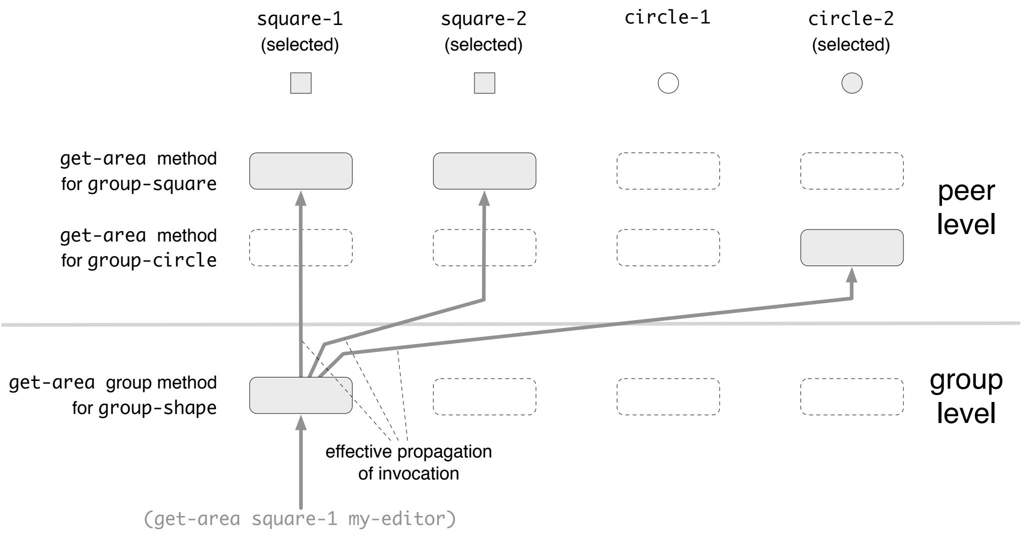


Figure 6.1: The effective method for an invocation to the `get-area` group generic function.

Assume also that `square-1`, `square-2` and `circle-2` are part of a multiple selection. Thus, Lambic handles the invocation `(get-area square-1 my-editor)` as follows:

- First, our model determines the applicability of the `get-area` group method. In this case, the group method is applicable as the argument list of the invocation satisfies the only parameter specialiser of the group method: the `group-shape` group class, specialiser of the `shape` parameter. This parameter is bound to the argument of the invocation, `square-1`, which is an instance of `group-square`, a subclass of `group-shape`.

- Second, the group method determines the effective propagation. For this, the group method first builds the most general propagation which implies generating as many invocations as instances of `group-shape` are available. Each invocation uses a different instance as first argument. The resulting list looks as follows:

```
(get-area square-1 my-editor)
(get-area square-2 my-editor)
(get-area circle-1 my-editor)
(get-area circle-2 my-editor)
```

Listing 6.3: The most general propagation of `get-area` group method.

Then, the group method applies the `is-selected?` predicate to each of the above invocations. From them, only the one using `circle-1` does not satisfy the predicate (`circle-1` is not part of the current selection). Thus, the effective propagation of the `get-area` group method is the following:

```
(get-area square-1 my-editor)
(get-area square-2 my-editor)
(get-area circle-2 my-editor)
```

Listing 6.4: The effective propagation of `get-area` group method.

- Third, the applicable peer methods are computed for the invocations of the effective propagation. Only one method is applicable in each case: the one specialised on `group-square` for the invocations using `square-1` and `square-2` as argument, and that specialised on `group-circle` for the invocation using `circle-2`.
- Finally, the `get-area` group generic function combines and applies the group method and the selected peer methods. The group method is executed first. As it invokes `call-next-method` in its body, the effective propagation is performed. The peer method selected for each invocation of the propagation computes the area of the corresponding shape. The results are returned to the group method which handles them using the `total-area` function. The result of this function, the total area of all the selected shapes, is returned as result of the `call-next-method` form. This result also corresponds to the return value of the group method and thus of the invocation of the `get-area` group generic function.

In Figure 6.1, the effective method for the invocation of `get-area` forms a tree. The nodes of this tree, the grey rectangles, correspond to the group and peer methods applied to the selected shapes, whereas the arrows represent the execution flow. The white rectangles with dashed borders are cases discarded by the effective method (methods that are not applicable for the selected shapes, or methods using as argument a shape that is not selected). At the group level, the group method is applied only to the shape passed as argument, `square-1`. This is because in this scenario we assume that this shape is chosen by the user from the GUI of the drawing editor. Note, however, that using any other selected shape leads to the same effective method.

Handling Multiple Group Methods

Applying more than one group method for an invocation could lead to multiple executions of the same group or peer methods. Assume that we add the following group method definition to the example of the `get-area` group generic function of Listing 6.2:

```
(defgroupmethod get-area ((square group-square) editor)
  (:propagate ((is-selected? square)) :return #'total-area)
  (call-next-method))
```

Listing 6.5: The `get-area` group method for `group-square`.

The new execution of the invocation `(get-area square-1 my-editor)` is depicted in Figure 6.2. The group method specialised on `group-square` is more specific than the one specialised on `group-shape`. As such, the former is executed first. The problem occurs as both methods propagate the invocations to all the selected shapes (using the `is-selected?` predicate in their propagation expression). The group method for `group-square` invokes the group method for `group-shape` twice (one for `square-1` and another for `square-2`). These invocations result in the double execution of the peer methods for `square-1`, `square-2` and `circle-2`.

Our execution model does not feature any implicit means to avoid this situation. Instead, a `call-peer-method` form is provided (as an alternative to `call-next-method`) to enable a group method to propagate invocations directly to individual peer methods. This form enables skipping the execution of any other less specific group method. Thus, the following definition of the group method specialised on `group-square` will circumvent the one specialised on `group-shape`:

```
(defgroupmethod get-area ((square group-square) editor)
  (:propagate ((is-selected? square)) :return #'total-area)
  (call-peer-method))
```

Listing 6.6: Using the `call-peer-method` form.

The `call-peer-method` form can be used only within the body of a group method. Apart from directly invoking peer methods, the semantics of this form are similar to those of `call-next-method`. Both forms implicitly trigger the propagation of the group method, handling its results and exceptions.

Using `call-peer-method` makes the execution of group generic functions more predictable. On that account, it may seem a good idea to make the semantics of this form the default behaviour of `call-next-method` (within group methods). However, in our experiments we have observed that the original semantics of `call-next-method` are essential to build more advanced group coordination schemes (especially when allowing group methods to have context predicates, as we explain later in Section 6.4.2).

Keeping the Original Arguments of Invocations of Group Generic Functions

Thus far, we have explained that a group method can propagate an invocation to the peers of a group class sharing a similar state. Alternately, there are some cases in which the

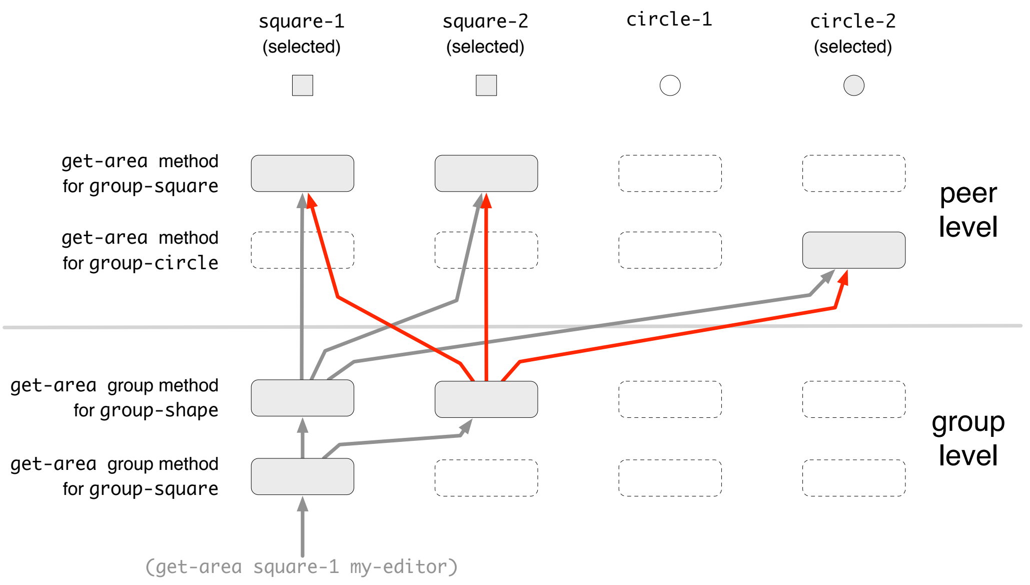

square-1
(selected)
square-2
(selected)
circle-1
circle-2
(selected)
get-area method for group-square
get-area method for group-circle
peer level
get-area group method for group-shape
get-area group method for group-square
group level
(get-area square-1 my-editor)

Figure 6.2: Default execution of multiple group methods.

propagation should include the peers that share a common state with the peer originally passed as argument in the invocation. For this, it should be possible that such peer is available when determining the effective propagation for the invocation (i.e. when evaluating the propagation predicates). This way, the peer's context can be compared with the context of the other peers.

Consider as an example a variation of the `get-area` group generic function. In this variation, invocations are propagated exclusively to the shapes of the same colour as the shape originally passed as argument. It should be possible that the shape is accessible for the propagation predicates so that its colour can be compared with the colour of the other shapes.

To deal with such a special case, Lambic enables referring to the original arguments of an invocation of a group generic function. This is achieved by means of the `original` function. This function can be used only in the propagation expression as an argument passed to a propagation predicate. Listing 6.7 illustrates the use of the `original` function in the implementation of the `get-area` group method. In the listing, the `same-colour?` method is the propagation predicate of `get-area`. Using the `original` function ensures that the method will always receive the shape passed originally in the invocation as second argument.

```
(defgroupmethod get-area ((shape group-shape) editor)
  (:propagate ((same-colour? shape (original shape))) :return #'total-area)
  (call-next-method))

(defmethod same-colour? (shape-1 shape-2)
  (equal (get-colour shape-1) (get-colour shape-2)))
```

Listing 6.7: Use of `original` function.

6.3 Lambic's Support for Empathic Group Behaviour

In this chapter, we have described empathic group behaviour as the capacity of an entity to implicitly receive invocations addressed to similar other entities, to contextualise such invocations, and to respond to them accordingly. In the group generic functions model, we introduce such empathic behaviour to object-oriented programming as follows:

- We define a group class abstraction which can explicitly keep track of all its instances, and a group method abstraction to propagate invocations among such instances. Such propagation occurs implicitly for client programs. It is declaratively specified inside the group methods. The client programs can invoke the group generic function using only one instance of the group class.

- The invocations resulting from the propagation are handled independently. For each invocation, the group generic function applies an effective method which is determined according to the context of the invocation. In this case, such context corresponds to the classes of the arguments of the invocation.

In the remainder of this section, we evaluate group generic functions with respect to the requirements for group behaviour presented in Section 2.4.

6.3.1 Plurality Encapsulation

The main purpose of plurality encapsulation is to access a remote service without regard for the number of objects that provide the service. In Lambic, group generic functions encapsulate the group behaviour for objects. As such, the group concern is abstracted from the communication concern. This means that a client can interact uniformly with a service represented by one object or a group of objects. In our model, the group does not require a special group identity. It can be addressed by invoking a generic function on *one* of its members. In addition, invocations can be implicitly handled by several members of the group. The clients can be unaware of such collaborative executions. Thus, we achieve arity decoupling as in the Mailer/Encapsulator model, Gaggles, AWED, DACs, Typed Groups and Ambient References (cf. Section 3.3.1). Yet, our model does not require a generic group interface (as in the case of DACs), or a special communication protocol (as in Ambient References and Typed Groups). Group generic functions are invoked in exactly the same way as standard generic functions.

6.3.2 Group Protocols

Thus far, we have presented group generic functions in a non-distributed setting. In this setting, we cannot yet discuss the group protocols for discovery and communication analysed in Section 3.3.2. In the next section we describe the integration of group generic functions into the AmOP paradigm, which then provides such protocols.

6.3.3 Modularity

Lambic separates the group behaviour (contained in group methods) from the base functionality of the group classes (contained in standard methods). When the group generic function is invoked, the group and peer methods are dynamically selected, sorted and combined according to the classes and group classes used as parameter specialisers. Applicable group and peer methods can uniformly access the next most specific by means of the `call-next-method` form. Alternatively, a group method can directly access the peer methods by using `call-peer-method`.

With group generic functions, we achieve similar results in terms of modularity as the Mailer/Encapsulator model [GFGM98], AWED [NSV$^+$06] and DACs [EGS00]. However, in our approach group and base behaviour are uniformly defined, in terms of classes and methods. Most of the extra support required for group classes and group methods is hidden in Lambic's execution model (apart from the group method's propagation expression). This is unlike those models where group behaviour is defined using aspects (AWED) and reflective programming (Mailer/Encapsulator model and DACs).

6.4 Group Generic Functions in AmOP

We now discuss the integration of group generic functions with Lambic's support for distribution and context dependency. We first explain the combination between group and futurised generic functions, and then between group and predicated generic functions. At the end of this section we discuss the combination of the three features. Table 6.2 shows the syntax of this combination.

6.4.1 Combining Futurised and Group Generic Functions

We distinguish three aspects in the combination of futurised and group generic functions: decentralised group class discovery, future-based group communication, and time-based failure handling for group communication.

```
Group class definition ::=
(defgroupclass class-name (parent-class)({field}))                 → new-class

Group generic function definition ::=
(defgroupgeneric function-name parameter-list
                 [(:group-predicates {pred-name})]
                 [(:predicates {pred-name})])                      → new-groupgeneric

parameter-list ::= ({parameter})

Group method definition ::=
(defgroupmethod method-name [execution-qualifier] [combination-qualifier]
                specialised-parameter-list
                [predicate-expression]
                propagation-expression
                body)                                              → new-groupmethod

execution-qualifier ::= :interruptible | :uninterruptible
combination-qualifier ::= :before | :around | :after
specialised-parameter-list ::= ({parameter | (parameter specialiser)})
predicate-expression ::= (:when {(pred-name arguments)})
propagation-expression ::= (:propagate ({(pred-name arguments)})
                           [in-actor-of] [due-in] [catch] [return])
  in-actor-of ::= :in-actor-of actor-designator
  due-in ::= :due-in ⟨nil | seconds⟩
  catch ::= :catch {(exception-name ([argument])body)}
  return ::= :return ⟨nil | user-defined-function⟩

Peer method definition ::=
(defmethod method-name [execution-qualifier] [combination-qualifier]
           specialised-parameter-list
           [predicate-expression]
           body)                                                   → new-method
```

Table 6.2: Lambic's integrated syntax for futurised, predicated and group generic functions.

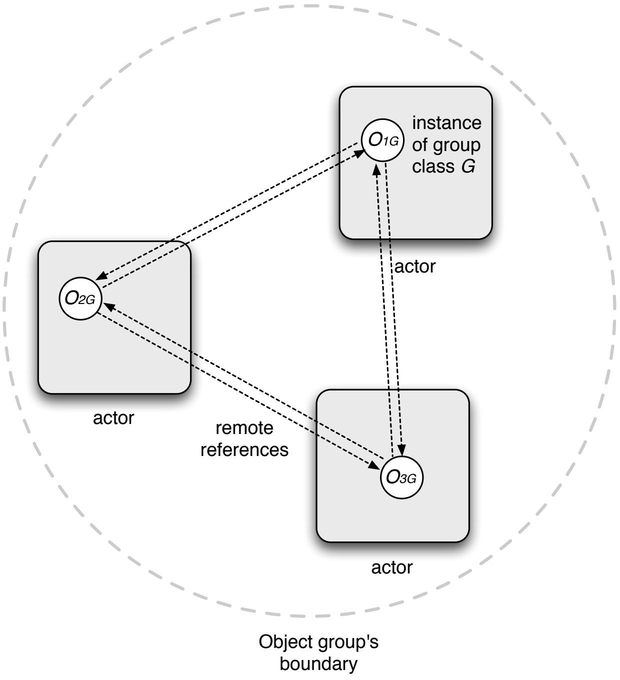


Figure 6.3: Distributed group class.

Decentralised Group Class Discovery

In a distributed environment, there can be instances of a group class residing at different locations (devices). In Lambic, such locations are represented as actors. Each actor can contain a copy of the group class definition and a set of its instances (the instances created within the actor). To allow the empathic behaviour in such a setting, our model enables the group class to implicitly discover and collect all the instances available in its environment, even if they are contained in different actors. This occurs in a decentralised publish/subscribe fashion, using the discovery mechanism of Lambic presented in Section 4.4.1. Each actor publishes its instances of the group class, and discovers the instances located in other actors. As any object in Lambic, when an instance of a group class spans its actor boundaries, it is converted into a remote reference. Thus, the discovery process results in a list of local and remote references to the instances stored at each copy of the group class. Figure 6.3 illustrates the result of this process.

Peer Discovery Event Handler. Lambic enables group classes to react to the discovery of their instances. Our model provides a `peer-discovered` generic function which is invoked whenever an instance of a group class is discovered. This generic function has two parameters representing two instances of the same group class: a local instance (the one being notified about the discovery) and the discovered remote instance. Methods specific for a group class can be defined by specialising the first argument on the group class. The `peer-discovered` generic function is called as many times as local instances of the group class exist in the actor discovering the remote instance.

User-defined methods for the `peer-discovered` generic function do not intervene with the implicit gathering of instances effectuated by the group classes.

External Discovery. Outside the definition of the group class, programs can discover its instances using the `whenever-discovered` form, described in Section 4.4.1. This form notifies the discovery of every instance of the group class.

Futurised Group Communication

Group generic functions can be remotely accessed only via asynchronous invocations. The return value of such invocations is a future which is resolved with the result of the collaborative execution performed by the group generic function. A group generic function abstracts the group concern from the communication. As such, no special abstractions are required to remotely invoke them, other than those provided by futurised generic functions (cf. Section 4.4.2). In what follows, we review the combined definition and execution of futurised and group generic functions.

Combined Definition of Futurised and Group Generic Functions. The definition semantics of futurised do not interfere with those of group generic functions. Apart from using the `defgroupgeneric` form, group generic functions are specified in exactly the same way as futurised generic functions. Both definitions can thus be straightforwardly combined.

Combined Definition of Futurised and Group Methods. Group methods can contain asynchronous remote invocations in their body and propagation expressions. For the propagation expression this means including remote invocations in the propagation predicates, exception handlers and return function. To handle the futures returned by such computations, we establish the following semantics:

Future handling The futures returned by the propagation expression are implicitly handled by Lambic's execution model. This does not include support for exceptions ruining those futures. Our model requires that such exceptions are properly tackled using the `try-catch` form (cf. Section 4.4.3).

Propagation expression and execution qualifiers Group methods can include an execution qualifier (`:interruptible` or `:uninterruptible`). Such a qualifier concerns only the execution of the body of the group methods, as explained in Section 4.4.2. Regarding a group method's propagation expression, it is affected by the execution qualifier as follows:

- The execution qualifier does not affect the evaluation of the propagation predicates. This is because such predicates are evaluated before the execution of the group method's body (while determining the effective group method for an invocation).
- The execution qualifier does affect the exception handlers (included in the `:catch` option of the propagation expression). The handlers are processed during the propagation of an invocation which occurs as part of the execution of the group method's body (upon a call to the `call-next-method` form).
- The execution qualifier also affects the return function (included in the `:return` option of the propagation expression), as it is evaluated during the propagation of an invocation.

Definition of Futurised Propagation Predicates. *Futurised* propagation predicates share the same definition semantics as *futurised* context predicates (cf. Section 5.3.1). They must be defined in the same actor of the group method that uses them. A remote method cannot be used as propagation predicate. Instead, a predicate has to be defined invoking the remote function within its body. Thus, the predicate can handle network failures possibly affecting the remote invocation (using Lambic's `try-catch` form, cf. Section 4.4.3).

Combined Execution of Futurised and Group Generic Functions. The propagation of invocations to remote actors has two main effects on the execution semantics of group generic functions. First, the process to determine and apply the effective method has to implicitly deal with futures. As said before, these futures can result from the execution of the body and propagation expression of the group methods. Second, the effective method may not be fully processed in the same actor. Once an invocation has been propagated to a remote actor, it is that actor which has to determine and apply the rest of the effective method. These two effects require the following adaptations to the execution semantics for group generic functions, introduced in Section 6.2.4:

Selecting and sorting applicable group methods This step does not require modifications. The most specific applicable group method is consistently executed at the actor receiving the invocation. As such, the group methods should still be selected and sorted at this actor.

Determining the effective propagation For a group method specialised on a distributed group class, the effective propagation is determined by applying the group method's predicates to the local and remote instances of the group class. In case of predicates that need to be remotely evaluated, our execution model implicitly waits for their asynchronous results (i.e. the resolution of the corresponding futures). Similarly to the evaluation of *futurised* context predicates (cf. Section 5.3.1), at this stage the only way to avoid interleaving executions is by annotating the propagation predicates with the `:uninterruptible` execution qualifier.

Combining and applying selected group methods Because the effective method may not be completely processed in the actor receiving the invocation, the list of applicable group methods should not be directly combined. The effective method for a group generic function still corresponds to the most specific group method of such list. However, the next most specific method is applied in the same actor only if the previous method includes a local invocation in its the effective propagation.[5] Remote invocations are processed by the remote actors. At those actors, the group method that caused the current propagation is not executed again. The invocations are then handled with the next most specific (group or peer) method.

Auxiliary (*around*, *before* or *after*) group methods are integrated into the effective method using the combination rules described in Section 6.2.4. In particular, when the effective method is processed in different actors, the auxiliary group methods are executed by the same actor that handles their corresponding primary group method[6]. If there is no corresponding primary group method, they are executed together with the next most specific primary method.

Finally, the group method handles the results and exceptions produced by the propagation, with the return function and exception handlers. For propagations to remote actors, the group method implicitly awaits the resolution or ruin of the futures returned by the propagation.

Group Generic Functions and Actor Designators. By default, an invocation to a group generic function is executed in the actor of its first argument (the implicit *actor designator*, cf. Section 4.3.2). This is also true for its propagation. The invocations included in the effective propagation of a group method are processed in the actor of their first argument. This default behaviour works properly for a group method defining a propagation for a group class specialising its first parameter on a group class. Assuming that the group method defines a propagation for this group class, each invocation will have a different instance of the group class as first argument, ensuring its execution at the actor of each instance. However, to enable all the parameters of a group method to be used as actor designator, the propagation expression of the group method provides an `:in-actor-of` argument. Similarly to the `in-actor-of` form (cf. Section 4.4.2), this argument associates the invocations of the propagation with an explicit actor designator. It can correspond to any parameter of the group method.

Connection-independent Failure Handling for Group Communication

In a distributed group class, network failures can affect the communication of an instance with its clients and with its peers. Developers can cope with such failures by using Lambic's time-based failure handling mechanism (cf. Section 4.4.3). Invocations to group generic functions are annotated with a timeout delimiting the period of time to receive their result. The timeout is respected even in the presence of partial network failures.

[5] And only if such previous method invokes `call-next-method` in its body.

[6] I.e. the group method with the same signature as the auxiliary group methods but without combination qualifier.

During this period, invocations to disconnected actors are buffered until the connection is restored. Similarly, the result of the invocation is awaited even when the remote actor becomes temporarily unavailable. Only after the timeout is reached, our model ruins the future of the remote invocation with a `timeout-exception`. It can then be handled using Lambic's futurised `try-catch` form. This mechanism applies uniformly to all the interactions involved in the execution of an invocation of a group generic function. This includes the remote invocations made by the client program, those required to propagate the client requests to the remote instances of the group classes, and any other remote invocation made in the body of the group methods. For the case of the propagation, the timeout variable and handler for the `timeout-exception` are specified in the `:due-in` and `:catch` arguments of the group method's propagation expression. If no timeout is specified, our model takes the default value stored in the `*response-timeout*` variable. Alternately, if the `:due-in` option is set to `nil`, the result of the remote invocation will be awaited indefinitely.

Peer Connectivity Event Handlers. Lambic allows group classes to react upon changes in the connectivity of their instances. For this, we adapt the connectivity event handlers of AmbientTalk's event loop model. Lambic provides the generic functions `peer-disconnected` and `peer-reconnected` which are invoked whenever a remote instance of the group class becomes disconnected or reconnected. As in the case of `peer-discovered` (cf. Section 6.4.1), these generic functions have two parameters representing a local instance of a group class, and the disconnected (or reconnected) remote instance. Methods specific for a group class can be defined by specialising the first argument in the group class.

All the peer event handlers (for discovery and connectivity) can use execution qualifiers to specify how to react to possible event interleaving.

Listing 6.8 illustrates the integration between futurised and group generic functions in the context of the drawing editor scenario (cf. Section 2.1). Let us assume a case of several editors running at different mobile devices. The editors can start a collaborative session as soon as their devices get in the same network range. In this session, the editors can share and modify each other's shapes.

```
; Definition of the editor as a group class
(defgroupclass group-editor ()
  ((username :initarg :username :accessor username)
   (shapes :initform nil :accessor shapes)))

; Peer discovered event handler for group-editor
(defmethod peer-discovered ((local-editor group-editor) new-editor)
  (update-shapes local-editor new-editor)
  (alert discovery new-editor))
```

```
; Peer disconnected event handler for group-editor
(defmethod peer-disconnected ((local-editor group-editor) disconnected-editor)
  (remove-shapes local-editor disconnected-editor)
  (alert disconnection disconnected-editor))

; Peer reconnected event handler for group-editor
(defmethod peer-reconnected ((local-editor group-editor) reconnected-editor)
  (update-shapes local-editor new-editor)
  (alert reconnection reconnected-editor))
```

Listing 6.8: Defining a distributed group class.

In Lambic, we can model the collaborative editors as a group class (`group-editor`). This enables the editor objects to implicitly discover each other. The editors can set up the collaborative session using a `peer-discovered` method. This includes exchanging the information about the shapes between the local and the discovered editors (using the `update-shapes` generic function), and notifying the discoveries to the users (using the `alert` generic function). Modelling the editors as a group class also enables them to be aware of each other's changes of connectivity, via the `peer-disconnected` and `peer-reconnected` methods. In the example, we use these methods to remove the shapes of an editor each time it gets disconnected, and to update the shapes when it rejoins the session.

```
; Definition of the paint-shape group generic function
(defgroupgeneric paint-shape (editor shape color))


; PEER LEVEL DEFINITION
; Definition of the paint-shape peer method
(defmethod paint-shape ((editor group-editor) shape color)
  ; standard paint behaviour...
)

; GROUP LEVEL DEFINITION
; Definition of the paint-shape group method
(defgroupmethod paint-shape :uninterruptible ((editor group-editor) shape color)
  (:propagate ((each editor)) :catch (timeout-exception () nil)
   :return nil)
  (call-next-method))
```

Listing 6.9: Futurised group generic function.

Listing 6.9 presents the definition of a *paint* operation for the collaborative editors. It is represented by the `paint-shape` group generic function. The peer method of this function

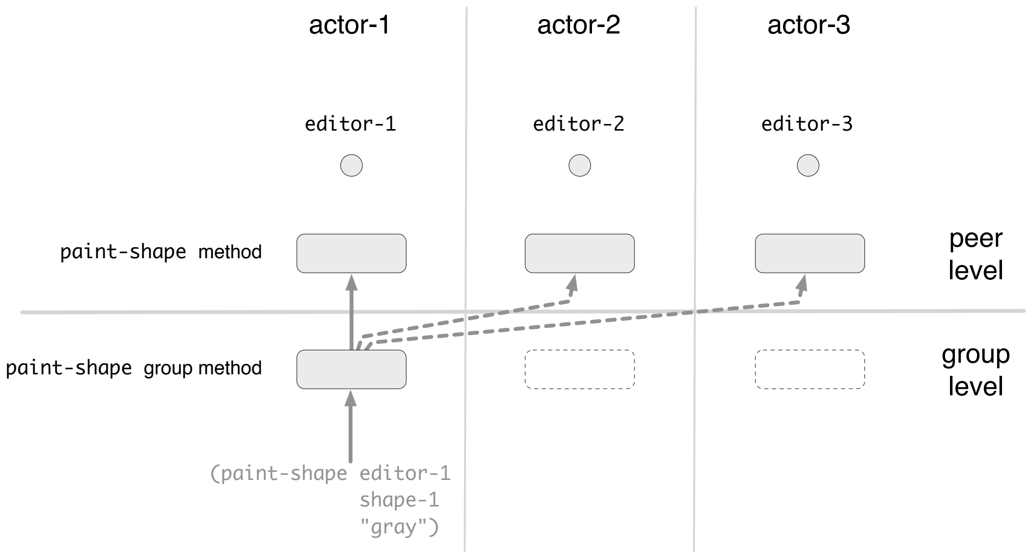


Figure 6.4: Distributed propagation.

contains the standard behaviour for painting shapes. The group behaviour specifies in its propagation expression that invocations should be communicated to all the editors of the session (using the `(each editor)` predicate).

Figure 6.4 illustrates the execution of the `paint-shape` group generic function. The remote invocations resulting from the propagation are indicated with dashed lines (to `actor-2` and `actor-3`). To deal with network failures possibly affecting such invocations, we use Lambic's time-based handling mechanism. In this simple implementation, we use the standard timeout (`*response-timeout*` variable). Timeout exceptions are discarded (returning `nil` as result). Similarly, no return value is expected from the execution of `paint-shape`. We indicate this by passing `nil` to the `:return` argument of the propagation expression. Finally, note that we can also avoid that the propagation is interleaved with other invocations, by annotating the group method with the `:uninterruptible` qualifier.

6.4.2 Combining Predicated and Group Generic Functions

The combination of predicated and group generic functions allows for *predicated* group and peer methods. Concerning the empathic group behaviour pattern (cf. Section 6.1), this feature augments the capacity of group class instances to react to invocations according to the context.

Combined Definition of Predicated and Group Generic Functions

Lambic enables group generic functions to specialise both peer and group methods in context predicates. Yet, because these two kinds of methods respond to different concerns, we let the group generic functions keep two independent lists of predicates. One list is denoted with the `:predicates` keyword and includes the predicates used by the peer methods. The other is denoted with the `:group-predicates` keyword and contains the predicates used by the group methods. Everything else operates as prescribed by the predicated generic function model (cf. Section 5.1.1). The priority between context predicates is established by the order in which they are declared in the group generic functions. Peer and group methods indicate the context predicate they specialise on, using a predicated expression headed with the `:when` keyword.

Both group and predicated generic functions have been combined with futurised generic functions. As such, group generic functions with context predicates can cleanly deal with asynchronous remote invocations.

Combined Execution of Predicated and Group Generic Functions

Allowing context predicates in group generic functions has one main effect on their execution semantics. The effective method is determined according to the context conditions of their arguments. This means that when an invocation is propagated to the instances of a group class, the set of applicable group and peer methods can vary from one instance to another —each instance can be in a different execution context. Hence, the reevaluation of the effective method is required not only at each actor involved in a propagation, but also for every invocation resulting from the propagation. This insight requires the following adaptations to the execution semantics for group generic functions, introduced in Section 6.2.4 and extended in Section 6.4.1:

Selecting and sorting the applicable group methods The sorted list of applicable group methods is obtained in the same way as for predicated methods (cf. Section 5.1.2). This implies selecting and sorting the group methods according to their parameter specialisers and context predicates. In case of predicates containing asynchronous remote invocations, our model awaits the resolution of the corresponding futures.

Determining the effective propagation Context predicates do not affect the semantics for determining the effective propagation of a group method. Thus, this step remains as described in Section 6.4.1.

Combining and applying selected group methods The effective method for an invocation corresponds to the most specific applicable group method. For each new invocation resulting from the propagation specified in this group method, the next most specific method is recomputed again. This applies for every invocation with an argument list different from the one of the original invocation. Unless the actor designator is explicitly indicated in the group method, the invocations are processed at the actor of their respective first arguments.

To illustrate the combined use of predicated and group generic functions, reconsider the example of painting shapes in a collaborative drawing editor, presented in Listing 6.9. In that implementation, a group method ensures that each time an editor paints a shape, this operation is propagated to all the editors of the session. Yet, in such a case there is still no control on the order in which each editor receives the updates. Listing 6.10 shows an alternative implementation of this example. We designate a leader to ensure that the updates arrive in the same order for all the editors. Instead of propagating a painting operation to all the editors, an editor communicates the painting actions to the leader. Then, it is the leader which eventually propagates the operation to the session.

```
; Definition of the paint-shape group generic function
(defgroupgeneric paint-shape (editor shape color)
  (:group-predicates not-leader?))

; PEER LEVEL DEFINITION
; Definition of the paint-shape peer method
(defmethod paint-shape ((editor group-editor) shape color)
  ; standard paint behaviour...
)

; GROUP LEVEL DEFINITION
; Definition of the paint-shape group method
(defgroupmethod paint-shape :uninterruptible ((editor group-editor) shape color)
  (:propagate ((each editor)) :catch (timeout-exception () nil)
   :return nil)
  (call-next-method))

; Definition of the paint-shape group method for non leaders.
(defgroupmethod paint-shape :uninterruptible ((editor group-editor) shape color)
  (:when (not-leader? editor))
  (:propagate ((leader? editor)) :catch (timeout-exception () nil)
   :return nil)
  (call-next-method))
```

Listing 6.10: Context-dependent group generic function.

To achieve leader-driven group behaviour, we define a second group method for all the editors that do not lead the session (the last definition of Listing 6.10). This is ensured by attaching the `not-leader?` context predicate to the group method. This context predicate is also declared at the definition of the group generic function (using the `:group-predicates` form). Then, the group method propagates the invocations to the leader editor by including the `leader?` predicate in its propagation expression. As we explained in Section 5.1.2, a method with a context predicate is considered more specific than one without it. As such, the group method with the `not-leader?` context predicate is executed before the one without predicates.

Figure 6.5 shows the execution of two invocations to the `paint-shape` group generic function. The invocation `(paint-shape editor-1 shape-1 “gray”)` is made at a non-leader editor. The other invocation, `(paint-shape editor-3 shape-1 “blue”)`, is made at the leader editor. For the first invocation, the most specific applicable method is the group method containing the `not-leader` context predicate. It propagates the invocation to the actor of `editor-3` (the only editor that satisfies the group method's propagation predicate). At this actor, the next most specific method corresponds to the group method without context predicates. Its execution results in the invocation of the peer method of the group generic function at each actor of the session.

The second invocation, instead, is directly handled by the group method without context predicates (as `editor-3` does not fulfil the context predicate of the other group method). Then, its propagation continues in the same way as the propagation of the other invocation.

Note that because the leader editor is contained in an actor, it can only process one invocation at a time.[7] The containing actor also ensures that invocations to the leader are propagated to the other editors, in the same order in which they are received. Finally, the `:uninterruptible` qualifier of the two group methods guarantees that the propagations of different invocations are not interleaved.

6.4.3 Discussion

Combining futurised, predicated and group generic functions enhances the implementation of empathic group behaviour as follows:

- Our model enables the definition of empathic behaviour for a distributed group of entities. Group classes implicitly collect their instances even if they reside in different actors. Group methods can propagate invocations over such actors. The propagation to remote actors is asynchronous and with explicit support to deal with network failures.

- Our model enables a group of services to further contextualise the incoming invocations. Group and peer methods can be specialised on any context condition. This condition is represented by a context predicate.

We now review Lambic in the light of the requirements for group behaviour in ambient-oriented programming (Section 3.4.2).

Distributed Group Behaviour

Lambic's integrated model for distributed and group behaviour complies with the following requirements.

[7]This also includes invocations coming from its GUI, which are also asynchronously processed. We show this in Section 7.2.

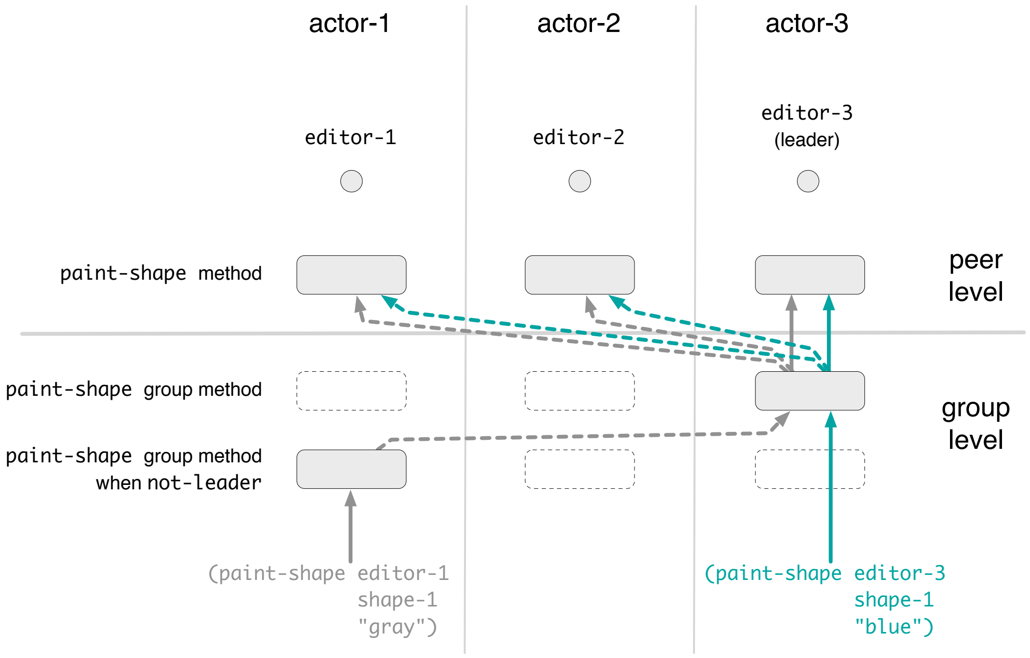


Figure 6.5: Context-dependent distributed propagation.

Decentralised Group Behaviour Management. In Lambic, group classes can implicitly discover their instances located in remote actors. This occurs in a decentralised fashion, using Lambic's publish/subscribe discovery mechanism. Additionally, our model enables each instance of a group class to be aware of the discovery, disconnection and reconnection of its peers. This is achieved by means of event handlers in the form of methods specialised on the group class.

Decoupled Group Communication. Lambic ensures a decoupled communication model for both interactions with the group and interactions between the group members:

Space decoupling Group generic functions can be invoked without mentioning the physical addresses or locations of the instances of the group class. Similarly, no address or location is required to propagate the invocations to the different instances of the group class. The propagation is declaratively specified (by means of propagation predicates), and implicitly executed (by invoking the `call-next-method` form).

Synchronisation decoupling Remote group generic functions can be only invoked asynchronously. The result of the invocations can be handled also exclusively asynchronously. Likewise, the propagation of an invocation to remote actors is also strictly asynchronous.

Time decoupling A group generic function can be remotely invoked even if the actor containing it becomes temporarily unavailable. For such a case, our model buffers the invocation until the connection with the remote actor is re-established. The same support is provided for the propagation to an instance of the group class that becomes temporarily unavailable.

Arity decoupling Invocations to group generic functions can be propagated and processed by all or a subset of the peer objects of a group class. However, no explicit enumeration is required. The peers are implicitly selected through the execution of the intensional propagation predicates.

Connection-independent Failure Handling. In addition to the event handlers for disconnection and reconnection, Lambic provides a time-based failure handling mechanism. An invocation of a remote group generic function, as well as its propagation to different actors, can be annotated with a timeout. The result of the invocation is awaited for the period indicated in the timeout, even if temporary disconnections affect the invoked actor.

Context-dependent Group Behaviour

Lambic's integrated model for context dependency and group behaviour, complies with the following requirements.

Context-dependent Selection of Group Behaviour. Group methods with context predicates enable group generic functions to provide multiple ways to propagate invocations. A group generic function can have several group methods specialised on different context conditions. As such, they can be dynamically selected and composed according to the context of each invocation. This allows the definition of more advanced collaboration schemes among the peer objects of a group class, as the example of leader-driven group behaviour presented in Listing 6.10 (further developed in Section 7.2).

Activation Scope of Group Behaviour. A group generic function can define the behaviour for a group class whose peer objects are located in different actors. The effective method for an invocation to such a group generic function has to be reevaluated after every propagation performed by one of its group methods. This is because each peer can be in a different context (e.g. located on a different actor), which may require different combinations of group and peer methods.

When processing invocations to group generic functions, the applicable methods have to be recomputed after the execution of every group method. This is because the propagation specified in a group method may lead to invocations to remote actors (the different locations of the instances of a group class).

Preserving Modularity and Composability. Finally, Lambic preserves the modularity and composability of the programs even when combining futurised, predicated and group generic functions. The group and peer level of a group generic function are still separated. Each level can have context-dependent variations, but they are also separated in different group or peer method definitions. There is no explicit reference between the methods. They can only anonymously access each other by means of the `call-next-method` form, and for the group level, also by means of the `call-peer-method` form. Both forms call the next most specific method to be processed, which is dynamically selected and sorted (based on their parameter specialisers and context predicates).

6.5 Conclusion

In this chapter, we have introduced a novel approach to model collaborations between similar services, called the *empathic group behaviour pattern*. In this pattern, services can implicitly receive invocations addressed to their peers, and respond to them according to their execution context. The key property for achieving this is to make the group behaviour part of the shared definition of the services.

In Lambic, we introduce empathic group behaviour to ambient-oriented programming in a model called *group generic functions*. This model provides two main abstractions: group classes and group methods. Group classes enable peer objects to be aware of each other. Group methods declaratively define the propagation of invocations among the instances of the group classes. The group generic function model allows for full encapsulation of the group concern (clients can be unaware of the group behaviour), modularity (the group behaviour is explicitly separated from the base or peer behaviour) and dynamic composability (group and peer methods are dynamically selected and sorted for every invocation).

We have successfully integrated group generic functions with the other features of Lambic, futurised and predicated generic functions. As such, group generic functions can be propagated to a group class with instances located in different actors. In addition, the group and base level behaviour of group generic functions can have several definitions specialised on different contextual conditions. Finally, group generic functions can be remotely invoked in an asynchronous fashion. Our model enables group generic functions to implicitly deal with asynchronous results (i.e. futures).

Lambic's integrated solution enables developers to gracefully deal with event-driven distribution, context dependency and group behaviour. In what follows, we illustrate the benefits of this integration through the development of the scenarios introduced in Chapter 2.

Chapter 7

Lambic at Work

This chapter presents the validation of the Lambic programming language model. We focus on the *expressiveness* of Lambic to modularise context dependency and group behaviour in the AmOP paradigm. We evaluate this expressiveness with respect to the requirements for modularity identified in Chapter 3. These requirements are derived from our study of the programming language research literature on event-driven distribution, context dependency and group behaviour. As explained in Chapter 3, the main claim of our work is that these concerns cannot always be handled in isolation. Thus, it should be possible to integrate the support provided for the three concerns.

The following list summarises the requirements for such an interdependent support (further details can be found in Section 3.4):

Context dependency in AmOP The language support required for event-driven distribution should not hinder the semantics of inheritance-based composition and dynamic method dispatch. There should be a controlled propagation of asynchrony ($R_{\mathbf{I.1}}$) and network failures ($R_{\mathbf{I.2}}$) in programs. Additionally, a programming language model should provide a consistent scope for context-dependent adaptations of event-driven programs ($R_{\mathbf{I.3}}$).

Group behaviour in AmOP Protocols for group membership should work in a peer-to-peer fashion ($R_{\mathbf{I.4}}$). In addition, intra- and inter-group computations should use a decoupled communication model ($R_{\mathbf{I.5}}$). Finally, the support to handle network failures affecting group interactions should be resilient to transient disconnections ($R_{\mathbf{I.6}}$).

Context dependency in Group Behaviour Programming languages should enable the definition and dynamic selection of group protocols according to context ($R_{\mathbf{I.7}}$). The model should also provide a consistent scope for context-dependent variations of group behaviour ($R_{\mathbf{I.8}}$). Finally, the modularity required for context-dependent behaviour should not hamper the modularity required for group behaviour ($R_{\mathbf{I.9}}$).

In this chapter, we show two expressiveness experiments we have carried out in Lambic. These experiments correspond to the implementations of the two pervasive computing services introduced in Chapter 2. In that chapter, we have argued that in pervasive computing, software entities providing similar functionality should abstract from specific locations and identities, and run in the environment as a single pervasive service. We have called such a property the *pervasive identities* of the services. We now illustrate how Lambic helps developers to model pervasive identities in software services.

We assess the resulting services' programs with respect to the above requirements. Finally, we discuss our solution in relation to the research literature presented in Chapter 3.

7.1 *Kriek*: A Pervasive Communication Service

Kriek is a communication service for pervasive computing. It is an extension of the scenario of the chat application presented in the AmbientTalk literature [VME$^+$11]. Kriek features two main properties:

Step #1 First, this pervasive computing service allows nearby people to talk with each other. For this, users must be able to discover all proximate chat instances without any preliminary configuration [Van08].

Step #2 Additionally, Kriek enables its user to maximise the devices at his disposal. The different functionalities required for chatting can be dynamically distributed across such devices. This distribution can be adapted according to the dynamic changes in the environment. The user can have a single account that he can simultaneously use on all devices.

Figure 7.1 illustrates these properties. Step #1 considers the case of several users co-located in the same environment. We assume that each user has a device with an instance of the Kriek chat. In Step #2, we assume additionally that some users have more than one device, also with an instance of Kriek. In what follows, we show how Lambic provides the appropriate programming support to build this service. We highlight the main details of the implementation of Kriek and refer the reader to Appendix C for further details.

7.1.1 Step #1: Basic Behaviour for Communication

Listing 7.1 shows the implementation of the Kriek communication service as the `kriek-chat` group class. This group class contains the standard state of a chat such as the user's name (the `username` field), the address book with his contacts (the `address-book` field), and the record of the user's conversations (the `history` field). Additionally, `kriek-chat` includes a `gui` field bound to the chat's user interface.

The fields of the `kriek-chat` group class are defined with different (standard CLOS) options. The `username` field provides an `:initarg` option with a keyword that is used when instantiating `kriek-chat`, to indicate an initial value for this field. The `address-book` and `history` fields provide an `:initform` option to set their default values (a hash table and

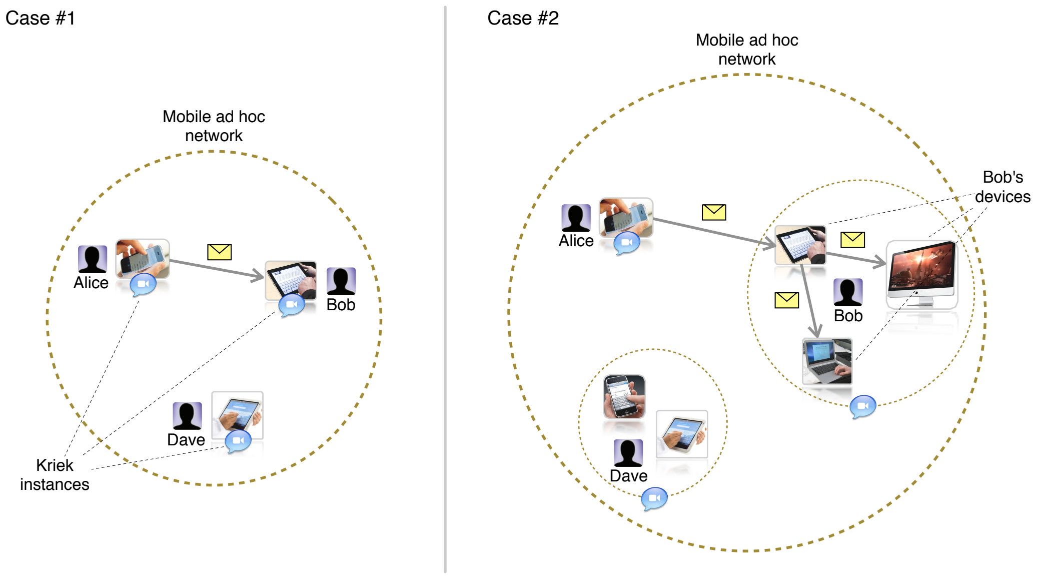


Figure 7.1: The Kriek chat service.

an empty list respectively). Finally, the four fields define an accessor function (denoted with the `:accessor` option) to get and set values from and to the fields.

```
; Definition of Kriek chat service as a group class
(defgroupclass kriek-chat ()
  ((username :initarg :username
             :accessor chat-username)
   (address-book :initform (make-hash-table :test 'equal)
                 :accessor chat-address-book)
   (history :initform '()
            :accessor chat-history)
   (gui :accessor chat-gui)))

; Sample driver code
(defvar *alice-chat* (make-instance 'kriek-chat :username “Alice”))
```

Listing 7.1: The Kriek chat service as a group class.

Managing Contacts.

Listing 7.2 illustrates the method definitions for handling discovery and connectivity events of the Kriek service. The `kriek-chat` group class implicitly gathers its instances available in the environment. This group class notifies the discovery of the instances by invoking the `peer-discovered` generic function. We can then define a `peer-discovered` method specialised on the `kriek-chat` group class to add the chat's contacts to its address book. In this method, we first request the contact's name by invoking `chat-username` on the remote chat. The invocation is specified within a `try-catch` form so that we can handle timeout exceptions (returning `nil` in such a case). Only if the contact's name is received, the contact is added to the local chat's address book and notified to the user. Finally, because this method contains an asynchronous remote invocation, it is annotated with the `:interruptible` execution qualifier.

The connectivity of a chat's contacts can be notified to its user by defining methods for the `peer-disconnected` and `peer-reconnected` generic functions. We use these methods to reflect the contacts' changes of status in the chat's GUI.

```
; Handling discovery of contacts
(defmethod peer-discovered :interruptible ((chat kriek-chat) remote-chat)
  (let ((remote-username (try-catch (chat-username remote-chat)
                            (timeout-exception () nil))))
    (if remote-username
      (begin
        (add-contact chat remote-username remote-chat)
        (display-contact chat remote-chat)))))

; Handling disconnection of contacts
(defmethod peer-disconnected ((chat kriek-chat) remote-chat))
  (hide-contact chat remote-chat))

; Handling reconnection of contacts
(defmethod peer-reconnected ((chat kriek-chat) remote-chat))
  (display-contact chat remote-chat))
```

Listing 7.2: Handling discovery and connectivity of contacts.

Sending and Receiving Text Messages.

Figure 7.2 presents a sequence diagram of the communication between two `kriek-chat` instances. The local chat receives an invocation to send a text to a remote chat. This text is wrapped in a text message and transmitted to the corresponding chat. Subsequently, both local and remote chats have to display the text message and to store it in their `history` field. In Lambic, rather than replicating these invocations in the methods for

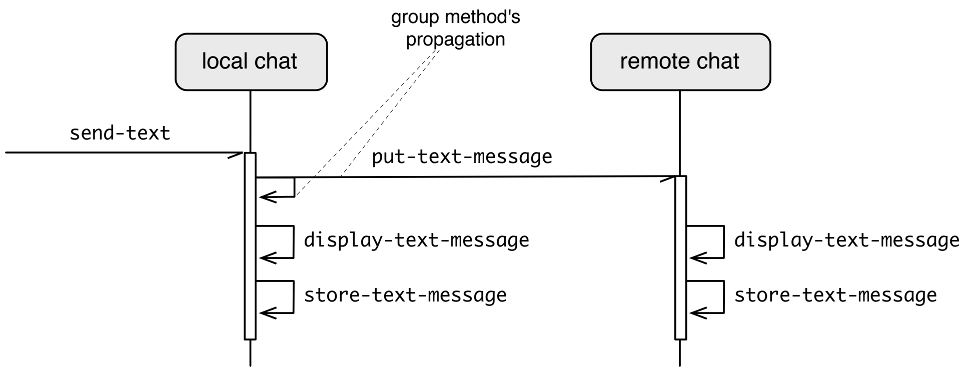


Figure 7.2: Sending and receiving chat messages.

sending and putting text messages, we include them in a `put-text-message` group generic function which, when invoked, is propagated to the two chat instances.

Listing 7.3 shows the definition of this behaviour. To model text messages we define a `message` struct. A struct is a typed data structure of Common Lisp which, unlike Lambic classes, is serialised using by-copy semantics [BDG$^+$88]. As such, we can ensure that the remote chat receives a copy of the message, and not a remote reference to it.

We then define a `send-text` generic function with a method specialising its first argument on the `kriek-chat` group class. This method creates an instance of the `message` struct and passes it as argument to the invocation of the `put-text-message` group generic function.

The `put-text-message` group generic function contains a group method and a peer method. To propagate invocations to the local and remote chats, the group method uses an `in-text-message?` predicate in its propagation expression. This predicate checks whether an instance of the `kriek-chat` group class is either the sender or receiver of the message. In the propagation expression, we set the `:due-in` option to `nil` to allow that messages are transmitted to the remote chat, despite any disconnection. We also set the `:return` option to `nil` as no result is expected from this group generic function. Finally, the peer method of the `put-text-message` group generic function invokes the `display-text-message` and `store-text-message` group generic functions. We explain these group generic functions in the following section.

```
; Text message
(defstruct message sender receiver text)

; Send text to contact
(defmethod send-text ((chat kriek-chat) remote-chat text)
  (let ((message (make-text-message :sender chat :receiver remote-chat :text text)))
    (put-text-message chat message)))
```

```
; Put text message in contact's and sender's chat
; GROUP LEVEL
(defgroupmethod put-text-message ((chat kriek-chat) text-message)
  (:propagate ((in-text-message? chat text-message)) :due-in nil
   :return nil)
  (call-next-method))

; PEER LEVEL
(defmethod put-text-message ((chat kriek-chat) text-message)
  (display-text-message chat text-message)
  (store-text-message chat text-message))

; Propagation predicates
; Check if chat is the message's sender or receiver
(defmethod in-text-message? ((chat kriek-chat) text-message)
  (or (equal chat (text-message-sender text-message))
      (equal chat (text-message-receiver text-message))))
```

Listing 7.3: Sending text messages.

7.1.2 Step #2: Adding Support for Pervasive Identities

In the second part of the scenario of Kriek, we assume that a user can have several chat instances running on different devices. This setting enables the definition of coordination schemes for the Kriek instances. In this example, we focus on the execution of the group generic function to receive text messages, `put-text-message`. As shown in Listing 7.3, this group generic function contains a method that displays and stores the messages. Instead of plainly performing both operations on every device, we assume the following distribution of tasks:

1. Displaying a message is a two-step process. First, we notify the message on every device. For this, we use a global notification system [Wik11b] which allows showing the message in a short-lived pop-up window.[1] Second, we add the message to the window displaying the conversation between the user and the contact sending or receiving the message. This window is open on only one device at a time (on the device that the user is actively using).

2. Messages are always stored on the same device. This means that messages sent or received on other devices should be eventually transmitted to this device. If the device is offline the transmission of the messages is postponed until it becomes available again.

[1]In our implementation we use the Growl notification system [Tea11]

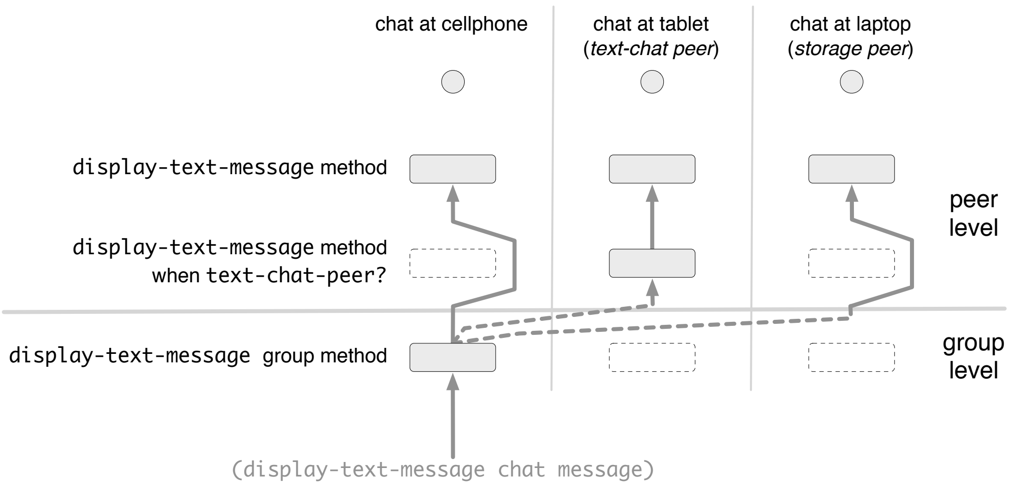

chat at cellphone
chat at tablet
(text-chat peer)
chat at laptop
(storage peer)
display-text-message method
display-text-message method
when text-chat-peer?
display-text-message group method
peer level
group level
(display-text-message chat message)

Figure 7.3: Execution of `display-text-message` group generic function.

To accomplish these requirements we establish a number of roles for the chat instances. We refer to the chat instance responsible to display the text-based conversations as the *text-chat peer*. Similarly, we call the instance in charge of storing the messages the *storage peer*. With these roles we can easily define the distributed behaviour required to display and store messages. We represent these operations by the `display-text-message` and `store-text-message` group generic functions, respectively.

Displaying Text Messages

Listing 7.4 shows the implementation of the `display-text-message` group generic function. Its execution is illustrated by Figure 7.3. This group generic function provides a group method that propagates the invocations to all the chat instances that belongs to the same user. We check this context condition using the `same-user?` propagation predicate. It compares the user of each chat instance available in the environment, with the user of the instance originally passed as argument in the invocation. The propagation returns `nil` as result. In this case, this result also corresponds to the return value of the group method.

At the peer level, the `display-text-message` group generic function has two methods. The first one is specialised on the *text-chat peer* role. We represent this role dependency by associating the method with a `text-chat-peer?` context predicate. This method displays the message in a chat window (invoking the `display-text-message-in-window` generic function), and calls the next method. The second method has no context predicate and as such it is executed by all the chat instances of the user. In this method we include the invocation to the notification system (using the `notify-text-message` generic function).

```
; Display text message
(defgroupgeneric display-text-message (chat text-message)
  (:predicates text-chat-peer?))

; GROUP LEVEL
(defgroupmethod display-text-message ((chat kriek-chat) text-message)
  (:propagate ((same-user? chat (original chat)))
   :return nil)
  (call-next-method))

; PEER LEVEL
; When chat plays the text-chat-peer role
(defmethod display-text-message ((chat kriek-chat) text-message)
  (:when (text-chat-peer? chat))
  (display-text-message-in-window chat text-message)
  (call-next-method))

; Default peer behaviour
(defmethod display-text-message ((chat kriek-chat) text-message)
  (notify-text-message chat text-message))

; Propagation predicate
; Check if chat's username equals the username of the chat
; passed originally as argument to the group generic function
(defmethod same-user? ((chat kriek-chat) original-chat)
  (let* ((address-book (chat-address-book original-chat))
         (contact (get-contact address-book chat))
         (original-username (chat-username original-chat)))
    (equal (contact-username contact) original-username)))
```

Listing 7.4: Displaying text messages.

Storing Text Messages

Listing 7.5 shows the implementation of the `store-text-message` group generic function. Its execution is illustrated by Figure 7.4. Unlike the case of `display-text-message`, this group method of `store-text-message` propagates the invocation only to the chat designated as the *storage peer* (using the `storage-peer?` predicate). Also, the propagation's timeout is set to `nil` to ensure that messages are eventually transmitted to the storage peer, despite temporary disconnections. Finally, at the peer level, the `store-text-message` group generic function has a method that save the messages in the chat's history.

Note that in the body of the method implementing the `storage-peer?` propagation predicate, we define the invocation of the `chat-role` accessor function within a `try-catch` form. The reason for this is that `storage-peer?` is tested for local and remote chats. Thus, we have to handle timeout exceptions for invocations of `chat-role` on remote chats.

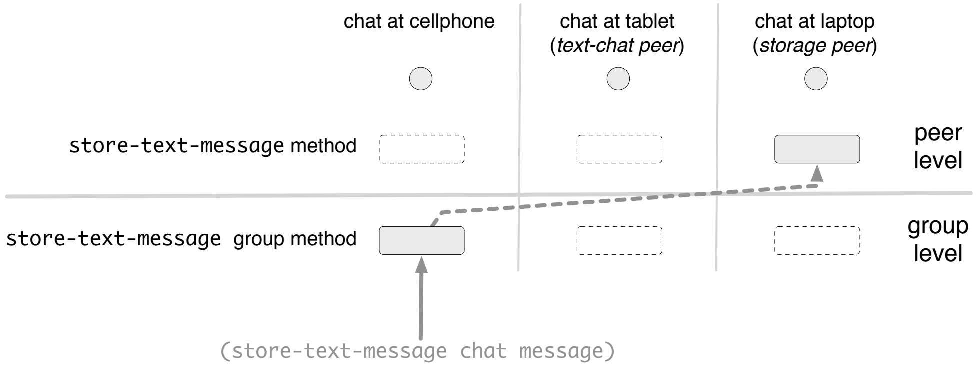


Figure 7.4: Execution of `store-text-message` group generic function.

For the same reason, the `storage-peer?` method is annotated with the `:interruptible` execution qualifier.

Note also that the `storage-peer?` propagation predicate will be re-evaluated for all the peer chats, at every invocation of the `store-text-message` group generic function. This may lead to important execution and network traffic overheads (due to the local and remote invocations made by this predicate). In Section 7.2.7, we further discuss these issues.

```
; Store text message
(defgroupgeneric store-text-message (chat text-message))

; GROUP LEVEL
; Propagate to chat playing storage-peer role
(defgroupmethod store-text-message ((chat kriek-chat) text-message)
  (:propagate ((storage-peer? chat)) :due-in nil
   :return nil)
  (call-next-method))

; PEER LEVEL
; Record text message in chat's history
(defmethod store-text-message ((chat kriek-chat) text-message)
  (push text-message (chat-history chat)))

; PREDICATE
(defmethod storage-peer? :interruptible (chat)
  (let ((role (try-catch (chat-role chat)
              (timeout-exception () nil))))
    (equal role "storage-peer")))
```

```
; Definition of Kriek chat service as a group class
(defgroupclass kriek-chat ()
  (...
   (role :initform nil :accessor chat-role)))
```

Listing 7.5: Storing text messages.

7.1.3 Evaluation

Using Lambic, we ensure the following properties in the implementation of the Kriek service:

Context Dependency in AmOP. Lambic's implicit handling of asynchronous results (using futures) avoids that the semantics of method selection and composition are affected by asynchronous remote invocations. In the implementation of Kriek, we can notice this in the direct programming style used to define the group generic functions. For instance, the `storage-peer?` propagation predicate invokes the `chat-role` accessor function on remote chat instances. This invocation is internally converted and executed asynchronously. As such, its result can be directly used in the rest of the predicate's body. Furthermore, the result of the predicate is also directly used by the `store-text-message` group generic function.

Lambic enables dealing with network failures in a direct style as well. Remote invocations are implicitly associated with a timeout. Timeout exceptions are handled with a `try-catch` form. In the Kriek service, we use this form for the remote invocation made by the `storage-peer?` propagation predicate. This way we avoid that timeout exceptions interfere with the execution semantics of the group method that use such predicate (`store-text-message`).

In Lambic, the activation scope of context-dependent adaptations is restricted to the execution of an invocation of a generic function. This scope is respected even in case of interleaved executions of several invocations. In Kriek, we can observe this property in the execution of the peer level of the `display-text-message` group generic function. The selection and composition of peer methods is determined by the context and is preserved until the invocation is completely processed. Note, however, that these methods are executed synchronously. As such there are no possibilities of interleaving.

Group Behaviour in AmOP. In Lambic, the group management occurs in a fully decentralised fashion. Group peers can be informed about each other's discovery and changes in connectivity (by means of the `peer-discovered`, `peer-disconnected` and `peer-reconnected` generic functions). We define methods for such generic functions specialised on the `kriek-chat` group class. We use these methods to keep each chat's contact up to date.

Lambic features a decoupled model for group communication (as described in Section 4.4.2). In the group behaviour of Kriek, there is no mention of concrete addresses or locations of the chats (space decoupling). Communication between chats is only asynchronous (synchronisation decoupling). Messages to disconnected chats can be implicitly stored until they become available again (time decoupling). Finally, a user can have several chat instances. Still, he can be reached at one instance which will propagate the message. This propagation is transparent for the sender of the message (arity decoupling).

Lambic enables communication to abstract from network failures. As said before, this is achieved by means of a time-based network failure handling mechanism. In the implementation of Kriek, this mechanism is used in the definition of its group behaviour (inside the propagation predicates).

Context Dependency in Group Behaviour. Lambic enables group behaviour to have context-dependent variations. However, this property is not relevant for this scenario. We better illustrate this in the next scenario (Section 7.2.4).

As in the case of context-dependent behaviour, in Lambic the activation scope of group behaviour is also restricted to the execution of an invocation of a generic function. This scope is preserved even in case of interleaved executions of invocations. This feature is also better illustrated by the next scenario (Section 7.2.4).

Lambic successfully modularises the definition of Kriek. None of the functionalities required by this service are intermingled. There is a strict separation between base and group behaviour, as in the cases of the `put-text-message`, `display-text-message` and `store-text-message` group generic functions, and between base and context-dependent behaviour, as in the peer level definition of the `display-text-message` group generic function. Furthermore, the programs invoking the group generic functions can be unaware to such group and context-dependent behaviour. Our model enables the Kriek service to dynamically select and compose its behaviour according to the execution context. For instance, invocations of the `display-text-message` group generic function are executed and propagated differently according to the roles of the chat peers received as arguments.

Open Issue

Finally, the implementation of Kriek also reveals an important issue in our model. The propagation and context predicates are evaluated for every invocation of the generic function that uses them. This may have negative consequences for the efficiency and network traffic. We further discuss this issue and propose an elementary solution, in the context of a larger and more complex scenario presented in the next section.

7.2 *Geuze*: A Collaborative Drawing Editor

The second scenario in this validation chapter is a drawing editor for pervasive computing, called *Geuze*. Geuze features a set of graphical operations for creating, selecting, drawing, moving and painting shapes. Additionally, this editor enables nearby devices to create collaborative editing sessions on a common canvas. These features entail a number of non-trivial issues for distribution, context dependency and group behaviour. Yet, Lambic enables dealing with such issues in an incremental way. For this scenario we propose the following development steps:

Step #1 We first define the base functionality of Geuze (Section 7.2.2). This includes the behaviour and state required for the editor's graphical operations.

Step #2 We add context-dependent variations to the base functionality (Section 7.2.3). In this implementation, such variations are related mainly to the handlers of GUI events.

Step #3 We define the group behaviour of Geuze, i.e. the coordination protocol required to ensure a consistent replica of the drawing at each editor (Section 7.2.4).

Step #4 We add context-dependent variations to the group behaviour (Section 7.2.5). In this case, the variations correspond to different propagation strategies for the graphical operations.

Step #5 Finally, we complete the support for distribution (Section 7.2.6). This comprises handling the editors' discovery and disconnections, adding timeouts and exception handlers to remote interactions, and annotating methods with the appropriate execution qualifiers.

In this section, we present the implementation of Geuze following the above steps sequentially.[2] For the sake of conciseness, we focus our discussion on a particular use case:

> A user moves a shape in his editor. For this, the user has to select the shape first and then perform the move. Both the selection and move operations have a visual effect. The former is manifested by drawing a halo around the shape. The latter causes a shift in the shape's position.

In the remainder of this section, we focus on the development of the operations for selecting and moving shapes. Further details about the implementation of this service can be found in Appendix D.

7.2.1 A Quick Overview of the Implementation of Geuze

Figure 7.5 gives a quick overview of the implementation of Geuze. In particular, this figure shows the *select-shape* graphical operation. All the other graphical operations require a similar support. We have developed this operation incrementally following the five steps presented above:

- Definition of the basic behaviour of the *select-shape* operation (Step #1). For instance, the behaviour to draw a halo around the selected shape.

[2]Nevertheless, some program listings can still contain implementation details explained only in further steps.

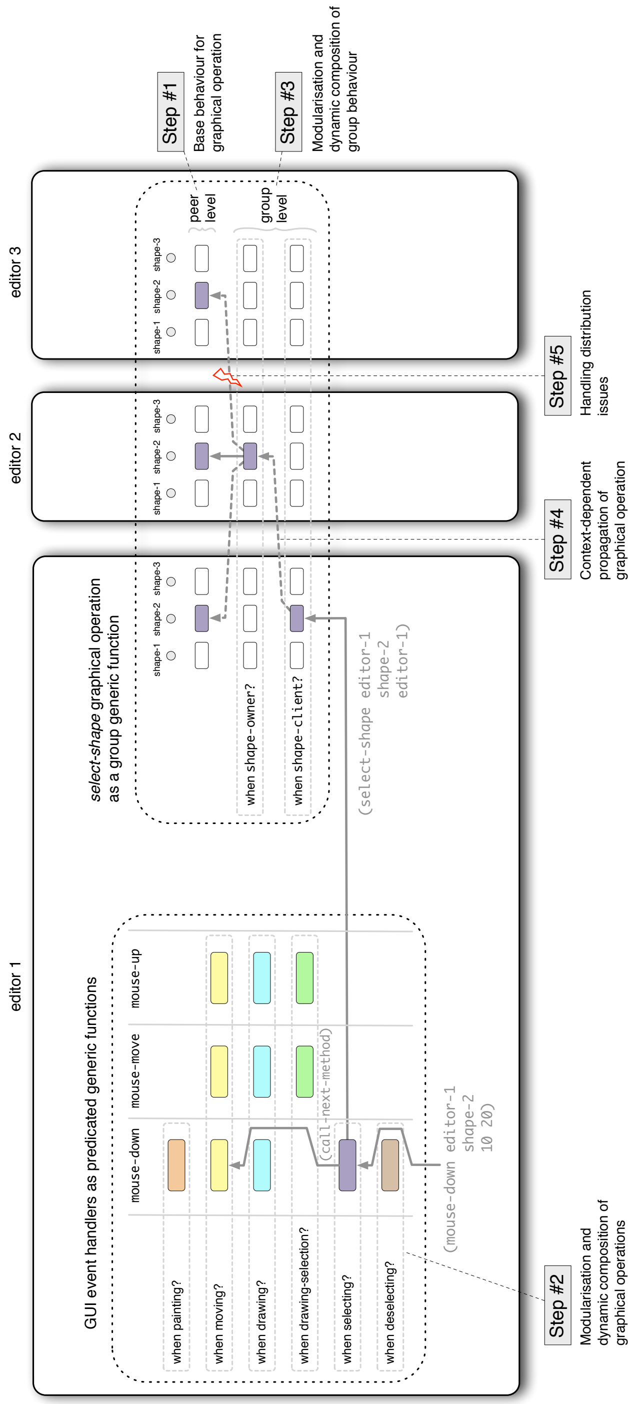

editor 1
editor 2
editor 3
GUI event handlers as predicated generic functions
mouse-down
mouse-move
mouse-up
when painting?
when moving?
when drawing?
when drawing-selection?
when selecting?
when deselecting?
(call-next-method)
(mouse-down editor-1 shape-2 10 20)
select-shape graphical operation as a group generic function
shape-1
shape-2
shape-3
when shape-owner?
when shape-client?
(select-shape editor-1 shape-2 editor-1)
peer level
group level
Step #1
Base behaviour for graphical operation
Step #2
Modularisation and dynamic composition of graphical operations
Step #3
Modularisation and dynamic composition of group behaviour
Step #4
Context-dependent propagation of graphical operation
Step #5
Handling distribution issues

Figure 7.5: Overview of the *select-shape* graphical operation in Geuze.

- Definition of the event handler that triggers the *select-shape* operation (Step #2). Because several operations can use the same event, it should be possible to define different event handlers (one for each operation), and to dynamically select them according to the context of use.

- Definition of the group behaviour for *select-shape* (Step #3). It corresponds to a leader-based coordination scheme required to ensure that only one editor can select a shape at a time.

- Definition of context-dependent propagation of invocations of the *select-shape* operation, e.g. according to the roles of the editors (Step #4).

- Definition of support for disconnections affecting group behaviour of *select-shape* (Step #5).

In what follows, we review each of these steps in detail.

7.2.2 Step #1: Basic Behaviour for Graphical Operations

The basic functionality of Geuze is structured around two main entities: editors and shapes. These entities are represented by the group classes, `geuze-editor` and `geuze-shape` (we justify this decision in Section 7.2.4). The `geuze-editor` group class contains the state required for the graphical operations of the editor, e.g. a list of shapes, a canvas, drawing tools such as a brush, etc. The `geuze-shape` group class contains the standard properties of shapes, e.g. position, size, colour, etc. Listing 7.6 outlines the definition of these group classes.

```
; Definition of geuze-editor group class
(defgroupclass geuze-editor (pinboard-layout)
  ((username :initarg :username :accessor editor-username)
   (shapes :initform nil :accessor editor-shapes)
   (canvas :accessor editor-canvas)
   (brush-active :initform nil :accessor brush-active) ...)

; Definition of geuze-shape group class
(defgroupclass geuze-shape (pinboard-object)
  ((name :initarg :name :accessor shape-name)
   (user-editor :initform nil :accessor shape-user) ...))

; Definition of geuze-rectangle group class
(defgroupclass geuze-rectangle (geuze-shape) ())
```

```
; Definition of geuze-circle group class
(defgroupclass geuze-circle (geuze-shape) ())

; Definition of geuze-drawn-shape group class
(defgroupclass geuze-drawn-shape (geuze-shape) ())
```

Listing 7.6: Geuze's group class definitions.

Part of the state and behaviour of the Geuze group classes is inherited from a Common Lisp library for graphical interfaces, called the CAPI library [Ric90]. In the listing, we mark the references to the library in grey.[3] The `geuze-editor` group class extends the `pinboard-layout` class which defines the functionality for a pane of graphical objects. The `geuze-shape` group class extends the `pinboard-object` class which defines the drawing capabilities for objects contained in a `pinboard-layout` object. In the listing above, we also include three concrete kinds of shapes —circles, rectangles and drawn shapes (i.e. shapes drawn by the user)— which we represent as the `geuze-circle`, `geuze-rectangle` and `geuze-drawn-shape` group classes respectively. The three of them are subclasses of `geuze-shape`.

Listing 7.7 shows the definition of the graphical operations for moving and selecting shapes. In short, the `move-shape` method sets a new position for the shape by changing its `pinboard-pane-position` property. The `select-shape` method sets a new user for the shape and invokes the `invalidate-rectangle` generic function. Both actions cause the shape to be redrawn by implicitly invoking the `draw-pinboard-object` generic function.

```
; Set and display new position for shape
(defmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y)
  (apply-in-pane-process editor
    (lambda (shape x y)
      (with-geometry shape
        (setf (pinboard-pane-position shape)
              (values (+ %x% x) (+ %y% y)))))
      shape x y))

; Set editor as shape's user and display selection effect
(defmethod select-shape ((editor geuze-editor) (shape geuze-shape) user-editor)
  (setf (shape-user shape) user-editor)
  (apply-in-pane-process editor
    (lambda (shape)
      (invalidate-rectangle (pinboard-object-pinboard shape)))
    shape))
```

[3] In this section, we give only an intuitive idea of the CAPI operations. Further details about their syntax and semantics are not necessary to understand our model. We include the operations in the listings mostly to show Lambic's compatibility with existing libraries.

```
; Draw pinboard object
(defmethod draw-pinboard-object ((editor geuze-editor) (shape geuze-shape))
  (if (shape-user shape)
    (draw-selection-effect editor shape))
  (draw-shape editor shape))
```

Listing 7.7: Geuze's base behaviour.

7.2.3 Step #2: Modularisation and Dynamic Composition of Graphical Operations

Each graphical operation of the Geuze editor requires an interaction pattern. A pattern is expressed in handlers for one or more GUI events. Additionally, the same GUI event can be used in several patterns. Table 7.1 shows the patterns for the operations that depend on mouse events. We refer to such patterns as the editor's *modes* of operation. Pressing the mouse button will trigger a different set of actions depending on the mode used. For instance, a same *mouse-move* event provokes a displacement of the shape when the editor is in *moving* mode, whereas a line is drawn —a completely different behaviour— when the mode is *drawing*.

The decision which operation mode should handle an event depends on the context of use. This context comprises the editor's state and any data passed to the event. Table 7.2 identifies the context conditions for each operation mode. For instance, the *moving* mode will handle a *mouse-move* event whenever this event occurs on a shape and the brush of the editor is not active. Conversely, if no shape is found and the brush is active, the *drawing* mode will be used instead.

There are cases in which two operations correspond to the same *mouse-down* event. If a deselected shape is moved, the shape is first selected, and then moved. This order of operations, although not apparent in tables 7.2 and 7.1, is an integral part of the normal behaviour of the editor and needs to be properly encoded.

Figure 7.6 illustrates Lambic's solution for modularising and dynamically composing graphical operations based on the context. As previously discussed in Section 5.2, in Lambic we model GUI event handlers as predicated generic functions. As such:

- We can cleanly handle the mouse events in separate methods. Each method is specialised on a context predicate representing a different operation mode. The predicate contains the context conditions specified for the mode.
- Methods defined for different generic functions can use the same predicate. This way we model operation modes that span several mouse event handlers.
- The priority order among the predicates defined at each generic function allows the consistent composition of the operation modes.

Figure 7.6 also shows the *context-dependent* execution of an invocation of the `mouse-down` generic function. This invocation occurs on a shape that is not in use, and when

Mode	**Actions**		
	Mouse down	**Mouse move**	**Mouse up**
Painting	paint shape	—	—
Moving	set drag status	move shape, update drag status	delete drag status
Drawing	set line status	draw line, update line status	create drawn shape, delete line status
Drawing selection	—	draw selection square, select found shape	remove selection
Selecting	select shape	—	—
Deselecting	deselect shape	—	—

Table 7.1: Actions for Geuze operations

Mode	**Context Conditions**
Painting	shape found, brush active
Moving	shape found, brush not active
Drawing	shape not found, brush active
Drawing selection	shape not found, brush not active
Selecting	shape found, shape not in use
Deselecting	shape not found

Table 7.2: Context conditions for Geuze operations

the brush of the editor is not active. As such, the invocation should be handled with the `mouse-down` methods specialised on the `selecting?` and `moving?` context predicates.

Listing 7.8 shows the definition of the *mouse-down* event handler as a predicated generic function. It declares the editor's different operation modes as context predicates (`moving?`, `selecting?`, etc.). These predicates correspond to methods containing the mode's context conditions. At the end of the listing we show the methods implementing the `moving?` and `selecting?` predicates. The listing also presents the methods specialised on such predicates. Remember from Section 5.1.1 that the order in which the predicates are declared in the generic function determines the precedence order among them.[4] Thus, the method specialised on the `selecting?` predicate is more specific than the method specialised on the `moving?` predicate. This means that if both predicates evaluate to true only the former method is executed. Still, because that method includes the `call-next-method` form in its body, the method with the `moving?` predicate can be eventually also processed. Note, however, that in this implementation the invocation to `call-next-method` is based on the result of the selection operation (the invocation to the `select-shape` generic function).

[4]The last predicate of the list has precedence over the others, cf. Section 5.1.1.

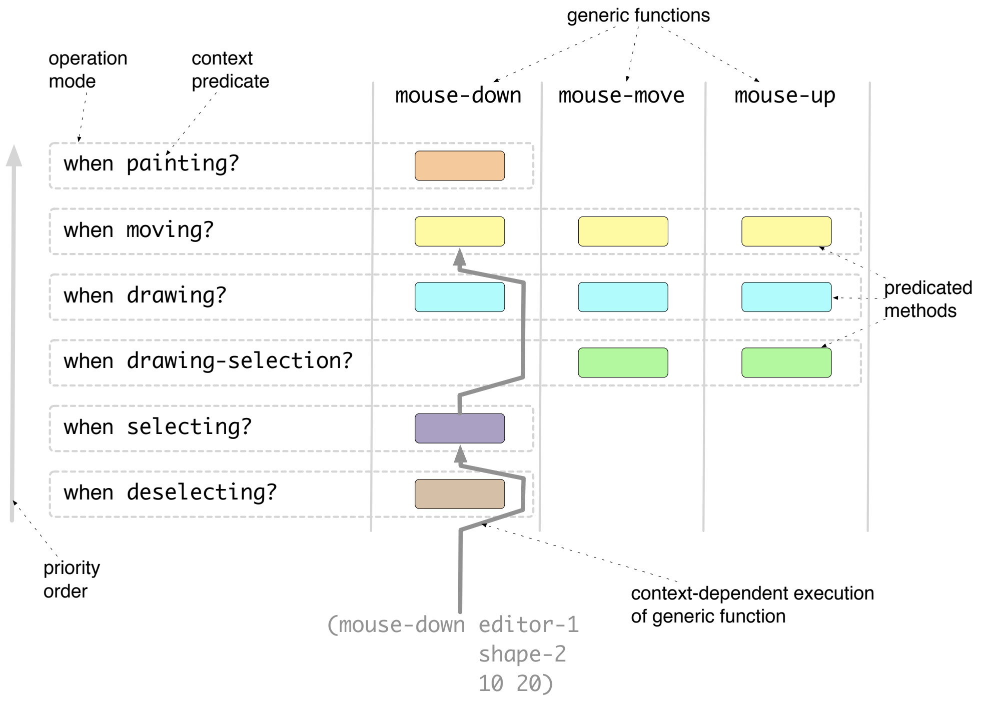


Figure 7.6: Modularisation and dynamic composition of graphical operations.

```
; The mouse-down generic function
(defgeneric mouse-down (editor shape x y)
  (:predicates painting? moving? drawing? drawing-selection?
               selecting? deselecting?))

; mouse-down when moving shape
(defmethod mouse-down ((editor geuze-editor) shape x y)
  (:when (moving? shape editor))
  (setf (drag-status editor) (make-drag-status :x x :y y)))

; mouse-down when selecting shape
(defmethod mouse-down ((editor geuze-editor) shape x y)
  (:when (selecting? shape editor))
  (if (select-shape editor shape editor)
    (call-next-method)
    (display-message “Shape already in use.”)))

...
```

```
; GUI predicates
; Enters into moving mode if a shape is passed as argument
; and the editor's brush is not selected
(defmethod moving? ((editor geuze-editor) shape)
  (and shape (not (brush-active editor))))

; Enters into selecting mode if a shape is passed as argument
; and it is not in use already
(defmethod selecting? ((editor geuze-editor))
  (and shape (not (shape-user shape))))
```

Listing 7.8: The `mouse-down` generic function.

The `mouse-down` method specialised on the `moving?` predicate only sets a temporary variable `drag-status`. The actual move of the shape occurs in a method defined for the `mouse-move` generic function. Listing 7.9 shows the rest of the implementation of the *moving* operation mode. The `mouse-move` method invokes the `move-shape` generic function and updates the `drag-status` variable. Finally, a method defined for `mouse-up` generic functions delete the drag status by setting this variable to `nil`.

Note that the `drag-status` variable is relevant only for the *moving* operation mode, i.e. for the methods specialised on the `moving?` predicate. Yet, in Lambic we cannot associate state to the predicates (unlike layers in ContextL [CH05]). Therefore, we have to define the `drag-status` variable as part of the state of the geuze-editor class.

```
; The mouse-move generic function
(defgeneric mouse-move (editor shape x y)
  (:predicates moving? drawing? drawing-selection?))

; mouse-move when moving
(defmethod mouse-move ((editor geuze-editor) shape x y)
  (:when (moving? shape editor))
  (move-shape editor shape x y)
  (let ((status (drag-status editor)))
    (setf (drag-status-x status) x (drag-status-y status) y)))

; The mouse-up generic function
(defgeneric mouse-up (editor shape x y)
  (:predicates moving? drawing? drawing-selection?))

; mouse-up when moving
(defmethod mouse-up ((editor geuze-editor) shape x y)
  (:when (moving? shape editor))
  (setf (drag-status editor) nil))
```

```
; Definition of drag status struct
(defstruct drag-status x y)

; Definition of geuze-editor group class
(defgroupclass geuze-editor (pinboard-layout)
  (...
   ; GUI temporary state
   (drag-status :initform nil :accessor drag-status)
   ...)
```

Listing 7.9: The *moving* operation mode.

7.2.4 Step #3: Modularisation and Dynamic Composition of Group Behaviour

We now present Geuze's group behaviour for collaborative drawing sessions. In this implementation, we achieve a consistent propagation of the graphical operations among the editors. For this, we assume that co-located editors can spontaneously start a session. In this session, they can share and modify each other's shapes. To allow such modifications to be consistent we adopt the following leader-based coordination protocol:[5]

1. We assume that each shape has an *owner* (the shape's leader). This owner corresponds to the editor that created the shape.

2. A shape can be modified by only one editor at a time. This requirement is ensured by the shape's owner. To modify a shape, an editor has to request the access from its owner. The owner grants the access if the shape is not already in use, and communicates this decision to all the editors of the session.

3. The editor that modifies the shape has to propagate the changes to all the editors of the session.

4. An editor can modify more than one shape at a time. In such a case, the editor has to request the access for each shape, as described in the second step. Modifications uniformly affect all the shapes being used by the editor, even if it actually operates on one of them only.

5. After the editor finishes the modification, it releases the shape (or shapes). For this, it contacts the shape's owner which eventually communicates the release to the session.

In the implementation of Geuze, we associate this coordination protocol with the editor's graphical operations. We relate the behaviour for requesting the access to a shape, with the graphical operation for selecting shapes. This means that selecting a

[5]We discuss this protocol's support for network failures in a different section (cf. Section 7.2.6).

shape in an editor is internally handled by making a request for access from the shape's owner. The shape can be selected only if its owner grants access. Additionally, we enable each graphical operation that modifies the shape (e.g. moving) to define its own propagation strategy (e.g. to propagate the move to some or all the editors, and to all the selected shapes). Finally, releasing the shape is represented as deselecting the shape.

We model the graphical operations as group generic functions. This way, we avoid that the coordination protocol becomes entangled with the operations' base functionality. The coordination protocol can be cleanly encapsulated in group methods, while the base behaviour is contained in peer methods (i.e. in standard methods, as previously presented in Listing 7.7).

To enable the propagation of graphical operations among the editors, we represent them as a group class (`geuze-editor`). Similarly, to allow the propagation of operations among shapes in case of multiple selection, we also model them as a group class (`geuze-shape`).

The `select-shape` Group Generic Function

Listing 7.10 shows the group-level definition of the `select-shape` group generic function. In this definition, we distinguish the group behaviour required for the editor selecting the shape (i.e. the shape's *client*), from the one required for the shape's *owner*. We model these two cases in different group methods. The first group method of the listing above defines the group behaviour for the owner editor. This condition is checked by the `shape-owner?` context predicate (we review the implementation of the predicates used in the group behaviour of Geuze later in Listings 7.12 and 7.13). The group method checks in its body whether the shape is available. If this is the case, the group method propagates the invocation to all the editors using the `same-shape-name?` propagation predicate. This leads to the execution of the peer method of the `select-shape` group generic function at each editor.

```
; Definition of select-shape group generic function
(defgroupgeneric select-shape (editor shape user-editor)
  (:group-predicates in-session? shape-owner? shape-client?))

; Group method for shape's owner
; Propagate selection to all editors
(defgroupmethod select-shape ((editor geuze-editor) (shape geuze-shape)
                              user-editor)
  (:when (in-session? editor)
         (shape-owner? shape editor))
  (:propagate ((same-shape-name? editor shape (original shape)))
   :return nil)
  (if (not (shape-user shape))
    (begin
      (call-next-method)
      shape)))
```

```
; Group method for shape's client
; Propagate selection to shape's owner
(defgroupmethod select-shape ((editor geuze-editor) (shape geuze-shape)
                              user-editor)
  (:when (in-session? editor)
         (shape-client? shape editor))
  (:propagate ((shape-owner? shape editor)
               (same-shape-name? editor shape (original shape)))
   :catch (timeout-exception () nil))
  (call-next-method))
```

Listing 7.10: Group-level definition of the `select-shape` group generic function.

The second group method of Listing 7.10 defines the group behaviour for the client editor (condition checked by the `shape-client?` context predicate). This group method's only purpose is to propagate the invocation of `select-shape` to the owner editor. This is achieved by using the `same-shape-name?` and `shape-owner?` predicates in the propagation expression.

Both group methods are also specialised on the `in-session?` context predicate. This predicate ensures that the group methods are applied only if the editor invoking `select-shape` is participating in a collaborative session. Otherwise, the editor will behave as a stand-alone application (executing only the peer level definition of the `select-shape` group generic function).[6]

Figure 7.7 illustrates the execution of an invocation of `select-shape`. In this example, `editor-1` selects `shape-2` which is owned by `editor-2`. The invocation is handled at the host of `editor-1` using the group method with the `shape-client?` context predicate. This group method propagates the invocation to the host of `editor-2`. At that location, the invocation is processed with the group method specialised on the `shape-owner?` predicate. Assuming that `shape-2` is not already in use, this second group method propagates the invocation to (the hosts of) all the editors of the session. The peer method is then executed at each editor, displaying the selection effect for `shape-2` and setting `editor-1` as its user.

[6]In our current implementation, the `in-session?` predicate only checks that a `session` field of the editor is not `nil`. We assume that this field is directly modified by users, based on whether they want to work alone or in collaboration.

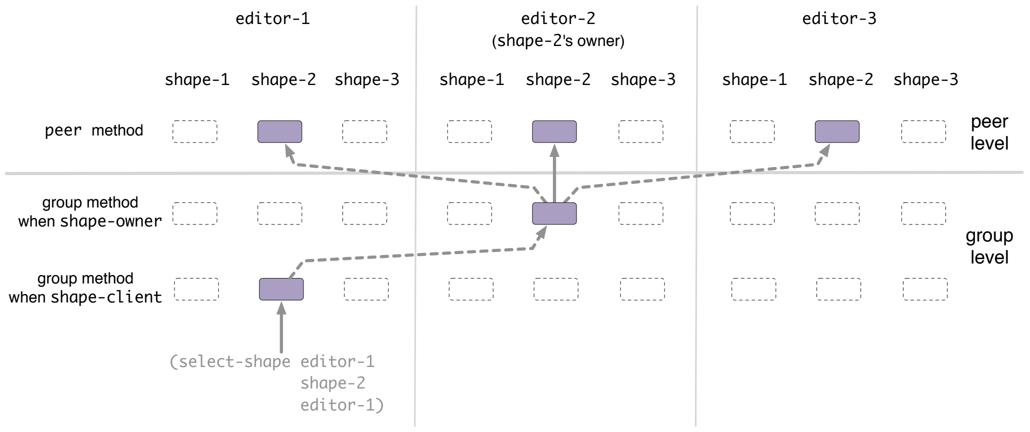

editor-1
editor-2
(shape-2's owner)
editor-3
shape-1
shape-2
shape-3
peer method
peer level
group method when shape-owner
group method when shape-client
group level
(select-shape editor-1 shape-2 editor-1)

Figure 7.7: Propagation of `select-shape` group generic function.

```
; Definition of move-shape group generic function
(defgroupgeneric move-shape (editor shape x y)
  (:group-predicates in-session?))

; Propagate the move to all the shapes selected by the editor
; performing the move
(defgroupmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y)
  (:propagate ((same-selection? shape (original shape)))
   :return nil)
  (call-next-method))

; Propagate the move to all editors
(defgroupmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y)
  (:when (in-session? editor))
  (:propagate ((same-shape-name? editor shape (original shape)))
   :return nil)
  (call-next-method))
```

Listing 7.11: Definition of *move-shape* operation.

The `move-shape` Group Generic Function

Listing 7.11 shows the group level definition of the `move-shape` group generic function. In this definition, we propagate the move to all the editors of the session. This propagation is made directly by the editor moving the shape (without intervention of the shape's owner, as in the case of `select-shape`). Additionally, if the editor has several shapes selected on

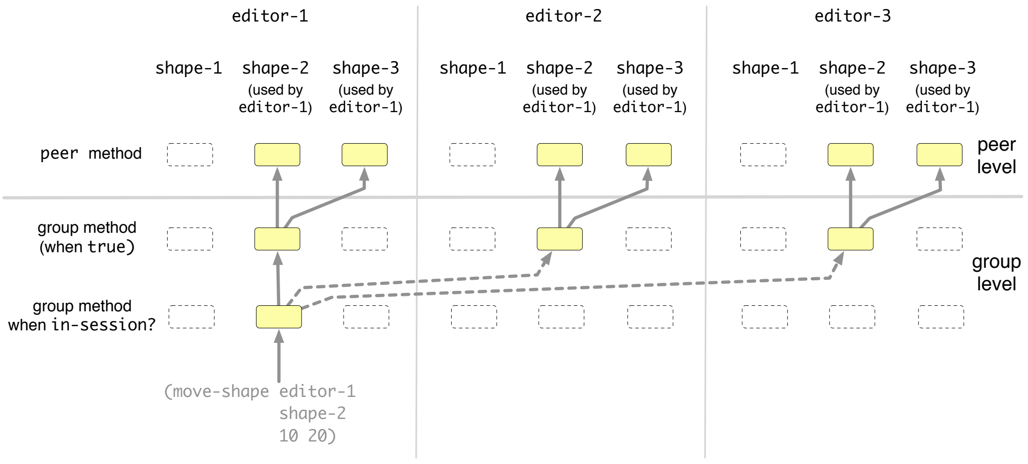

editor-1
editor-2
editor-3
shape-1
shape-2 (used by editor-1)
shape-3 (used by editor-1)
peer method
group method (when true)
group method when in-session?
peer level
group level
(move-shape editor-1 shape-2 10 20)

Figure 7.8: Local propagation of `move-shape` group generic function.

the canvas, we propagate the move to each of them. We model these two propagations in two independent group methods.

The first group method propagates the move to all the shapes selected by the editor. This is ensured by means of the `same-selection?` predicate. The second group method propagates the move to all the editors (using the `same-shape-name?` predicate). It is specialised on the `in-session?` predicate (as the group methods of `select-shape`). This predicate makes this group method more specific than the other without any predicate. As such, it is executed first. This means that invocations of the `move-shape` group generic function are first propagated to each editor, and then to each selected shape.

Figure 7.8 illustrates the execution of an invocation to the `move-shape` group generic function. In this example, we assume that `editor-1` has selected `shape-2` and `shape-3`. When this editor moves `shape-2`, this operation is propagated first to the three editors of the session, and then to `shape-2` and `shape-3`.

Predicates for the Group Behaviour of Geuze

Listing 7.12 shows the implementation of the context predicates specialising the `select-shape` and `move-shape` group methods. The `shape-owner?` predicate checks if the editor received as second argument is the owner of the shape received as first argument. This information is contained in the `owner` field of the `geuze-shape` group class. Conversely, the `shape-client?` predicate verifies that the editor is not the owner. The `in-session?` predicate corresponds to the accessor of the `in-session` field of the `geuze-editor` group class. Note that in this implementation the information about the shape's owner and the editor's session is relevant only for the group behaviour of Geuze. However, Lambic does not provide any means to separate group-level from peer-level state. As such, we have to model this information using standard fields in the group classes.

```
; Check whether editor owns shape
(defmethod shape-owner? (shape editor)
  (equal (shape-owner shape) editor)))

; Check whether editor does not own shape
(defmethod shape-client? (shape editor)
  (not (shape-owner? shape editor)))

; Definition of geuze-shape group class
(defgroupclass geuze-shape (pinboard-object)
  (...
   ; Group management
   (owner :initarg :owner :accessor shape-owner)
   ...)

; Definition of geuze-editor group class
(defgroupclass geuze-editor (pinboard-layout)
  (...
   ; Group management
   (in-session :initform nil :accessor in-session?)
   ...)
```

Listing 7.12: Context predicates of Geuze's group behaviour.

Listing 7.13 presents the implementation of the propagation predicates of the group behaviour of Geuze. The `same-shape-name?` generic function is used as a propagation predicate for the group methods of the `select-shape` group generic function. This predicate ensures that invocations to `select-shape` are propagated to the "same shape" at each editor. In this implementation, there is a replica of each shape per editor of the session. All the replicas correspond to instances of the `geuze-shape` group class. To distinguish the replicas of the different shapes, we assume that each shape has a unique name. Thus, `same-shape-name?` ensures that invocations to `select-shape` are propagated to the instances of `geuze-shape` with the same name as the one originally received as argument.

The fact that the `select-shape` and `move-shape` group methods have two parameters specialised on group classes (`geuze-editor` and `geuze-shape`) implies an extra complexity for the specification of propagation predicates. On the one hand, this design decision facilitates the propagation of the graphical operations to the different editors and shapes, as we have explained so far. On the other hand, it also implies that the propagation can potentially include an invocation for every combination between the instances of the two group classes (where each invocation uses a different combination as arguments, as discussed in Section 6.2.4). Hence, the propagation predicates are required to filter the invocations that are not pertinent to the program logic. For instance, the `same-shape-name?` predicate should ensure not only that an invocation to `select-shape` is propagated to the replicas of the same shape. It should also ensure that in every invocation resulting from the propagation, the editor and the replica of the shape used as arguments are co-located in the same host (i.e. the same actor). We check this condition by using Lambic's `co-located?` built-in predicate in the `same-shape-name?` method.

Finally, Listing 7.13 also shows the `same-selection?` generic function used as the propagation predicate in the `move-shape` group generic function. This predicate ensures that invocations to `move-shape` are propagated to all the selected shapes. For this, the user of each shape is compared with the user of the shape originally received as argument.[7] As previously explained in this section, this propagation occurs inside each editor. Therefore, only local shapes (i.e. local instances of the `geuze-shape` group class) should be compared. Remote shapes (i.e. remote references to `geuze-shape` instances) will be immediately discarded. In our implementation, we achieve this by defining two different methods. The first one specialises its two parameters on the `geuze-shape` group class. This method effectively compares the shapes. The second method does not specialise the parameters. It is applied when `same-selection?` is invoked with at least one argument which is not a local instance of `geuze-shape` (returning `nil` as result).

```
; Check whether shape has the same name as the shape originally
; passed as argument to the group generic function
(defmethod same-shape-name? (editor shape original-shape)
  (and (equal (shape-name shape) (shape-name original-shape))
       (co-located? editor shape)))

; Check whether shape has the same user as the shape originally
; passed as argument to the group generic function
(defmethod same-selection? ((shape geuze-shape) (original-shape geuze-shape))
  (equal (shape-user shape) (shape-user original-shape)))

; Return nil for arguments that are not local instances of
; geuze-shape
(defmethod same-selection? (shape original-shape)
  nil)
```

Listing 7.13: Propagation predicates of Geuze's group behaviour.

7.2.5 Step #4: Context-dependent Propagation of Graphical Operations.

We now introduce context-dependent variations to the group behaviour of the Geuze editor. To control the network traffic that the collaborative edition entails, we define several propagation strategies for the graphical operations. By default, when moving a

[7]The shape's user should not be confused with the shape's owner. The former is the editor that has requested the access to modify the shape. The latter is the creator of the shape.

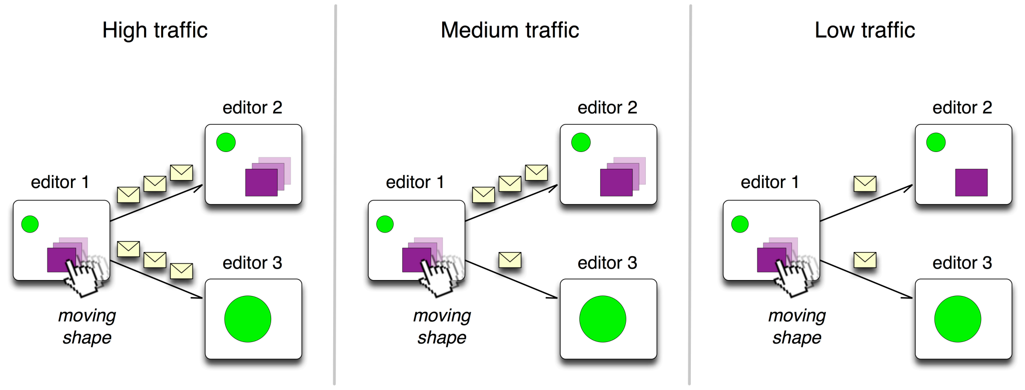


Figure 7.9: Context-dependent propagation of graphical operation.

shape we propagate each intermediate position of the shape (strategy called *high traffic*). Alternatively, we can propagate only few intermediate positions, and to a limited number of editors, according to some well-chosen criteria (strategy called *medium traffic*). For instance, the intermediate positions can be propagated only to the editors currently displaying the shape.[8] Eventually, we can propagate only the final position to all the editors (strategy called *low traffic*). Figure 7.9 illustrates these three strategies.

Listing 7.14 shows the redefinition of the `move-shape` group method that propagates the move to all the editors of the session (introduced in Listing 7.11). We include a `relevant-position?` propagation predicate that determines whether an intermediate position should be propagated. Because this predicate corresponds to a standard Lambic generic function, there can be several methods defining the predicate. Each method can be associated with a different context condition, also represented as a predicate. In the example, the `relevant-position?` generic function has a method without any context predicate representing the *normal traffic* strategy (propagating each position). Additionally, this generic function provides two more methods specialised on the `medium-traffic?` and `low-traffic?` context predicates respectively.

```
; Propagate the move to all the shapes selected by the editor
; performing the move
(defgroupmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y
                            final-position)
  (:propagate ((relevant-position? editor shape final-position)
               (same-shape-name? editor shape (original shape)))
   :return nil)
  (call-next-method))
```

[8]A similar approach has been proposed in the Beernet self-managing system [MV10].

```
; Check whether new shape's position should be
; propagated to the session.
(defgeneric relevant-position? (editor shape final-position)
  (:predicates medium-traffic? low-traffic?))

; Always propagate the new positions (normal traffic)
(defmethod relevant-position? (editor shape final-position)
  t)

; Propagate the new positions only to the editors
; that are currently displaying the shape. Additionally, propagate
; the final position to everyone.
(defmethod relevant-position? (editor shape final-position)
  (:when (medium-traffic? local-editor))
  (or final-position
      (on-focus? editor shape))

; Propagate only the final position.
(defmethod relevant-position? (editor shape final-position)
  (:when (low-traffic? local-editor))
  final-position)

; Check whether shape is being displayed by editor
(defmethod on-focus? :uninterruptible (editor shape)
  (let ((shapes (try-catch (shapes-on-focus editor)
                    (timeout-exception () nil))))
    (and (co-located editor shape)
         (find shape shapes))))
```

Listing 7.14: Context-dependent propagation of `move-shape` group generic function.

The method specialised on `medium-traffic?` accepts the propagation of a new position only to the editors displaying the shape, or to every editor if the new position is the final one. In this case, we assume that there is a zoom functionality so that the canvas can have a different size at each editor's window. Thus, we use an `on-focus?` generic function which checks whether an editor is currently displaying the part of the drawing where the shape appears.

Finally, the method specialised on `low-traffic?` accepts the propagation of a new position only if it is the final one.

7.2.6 Step #5: Handling Distribution Issues

In the final part of the definition of the group behaviour of Geuze, we add the support for distribution required by this service. This includes handling discovery and connectivity of the editors, handling disconnections affecting remote interactions, and adding the execution qualifiers to methods and group methods.

Handling Discovery and Connectivity Events.

We first review how to handle the discovery and connectivity of the editors. The main challenge in this example is to preserve a consistent collaboration among the editors despite their dynamic changes of availability. To cope with this issue we adopt the following policy:

- Editors that disconnect from each other should no longer be able to see and edit each other's shapes.
- Editors should automatically share their shapes when they discover each other, and whenever they reconnect to each other.

We implement this policy by means of Lambic's generic functions for peer discovery and connectivity events. Listing 7.15 shows the methods for such generic functions and which are specialised on the `geuze-editor` group class. In the `peer-discovered` and `peer-reconnected` methods, an editor sends its shapes to the discovered or reconnected peer by calling the `send-owned-shapes` generic function. In the `peer-disconnected` method, the editor hides the shapes of the disconnected peer using the `hide-disconnected-shapes` generic function. Additionally, the possibility exists that some of the editor's shapes was in use by the disconnected peer. In such a case the editor deselects the shapes using the `deselect-shapes` generic function. Thus, they can become available for the session. Figure 7.10 shows the execution of the three methods.

```
(defmethod peer-discovered ((local-editor geuze-editor) remote-editor)
  (send-owned-shapes local-editor remote-editor))

(defmethod peer-disconnected ((local-editor geuze-editor) remote-editor)
  (hide-disconnected-shapes local-editor remote-editor)
  (deselect-shapes local-editor remote-editor))

(defmethod peer-reconnected ((local-editor geuze-editor) remote-editor)
  (send-owned-shapes local-editor remote-editor))
```

Listing 7.15: Peer discovery and connectivity event handlers.

Handling Disconnections Affecting Remote Interactions.

Listing 7.16 shows the redefinition of the `shape-owner?` and `same-shape-name?` propagation predicates. Both predicates invoke an accessor function for a field of the `geuze-shape` group class (`shape-owner` and `shape-name` respectively). Because they are tested on all the instances of `geuze-shape`, we need to add special support for the tests on the remote instances. This implies using `try-catch` forms to handle timeout disconnections possibly affecting remote invocations of `shape-owner` and `shape-name`.

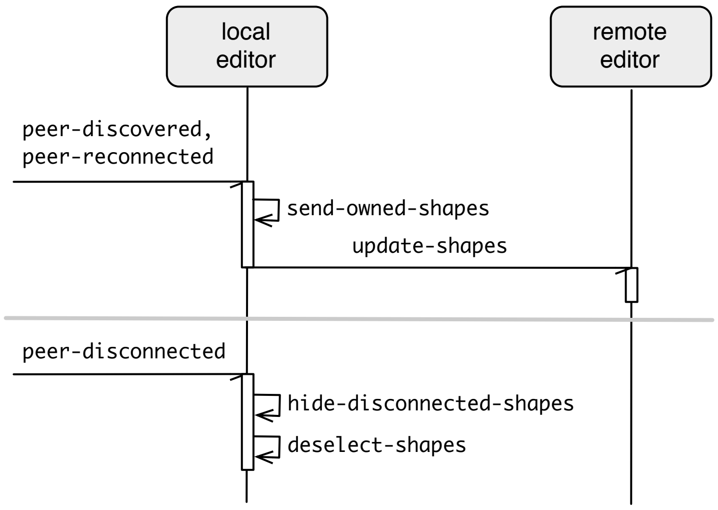


Figure 7.10: Handling discovery and connectivity events.

```
; Check whether editor owns shape
(defmethod shape-owner? (shape editor)
  (let ((owner (try-catch (shape-owner shape)
                  (timeout-exception () nil))))
    (equal owner editor)))

; Check whether shape has the same name as the shape originally
; passed as argument to the group generic function
(defmethod same-shape-name? (editor shape original-shape)
  (let ((shape-name (try-catch (shape-name shape)
                       (timeout-exception () nil))))
      (and (equal shape-name (shape-name original-shape))
           (co-located? editor shape))))
```

Listing 7.16: Handling disconnections affecting propagation predicates.

Note that the group-level definition of the `select-shape` group generic function also needs to handle timeout disconnections. As previously explained, the two group methods of `select-shape` specialised on the `shape-client` predicate propagates the invocations to the shape's owner (Listing 7.10). For the case in which the shape's client and owner are located on different hosts, this propagation corresponds to a remote invocation to the shape's owner. Lambic implicitly associates a timeout with this invocation. Timeout exceptions are handled by the group method's propagation expression, using the `:catch` argument.

Adding Execution Qualifiers.

Finally, we add the necessary execution qualifiers to the methods and group methods of Geuze. As explained in Section 4.4.2, Lambic throws a warning if a method that cannot be fully processed synchronously, is not annotated with an execution qualifier. This way, developers become aware that the execution of such a method can be interleaved with the execution of further invocations. Table 7.3 shows the list of all the methods and group methods discussed in this chapter which throw such a warning. For each method, we explain why they cannot be synchronously processed. We have made this analysis manually, tracing the execution flows of all the methods. Providing tool support for this task remains an important part of our future work, as discussed later in Section 8.5.5.

In Lambic, asynchronous executions are related to remote invocations. However, in Table 7.3 we can observe that only few of the methods throwing a warning make remote invocations. We highlight such cases in red. For instance, the `mouse-down` method specialised on the `selecting?` predicate is asynchronously processed only because it depends on the result of the `select-shape` group generic function. The most specific group method of this function (the one specialised on the `shape-client?` and `in-session?` predicates) is executed on the same actor where `mouse-down` is invoked. However, this group method's result depends on the remote execution of the second group method of `select-shape` (specialised on the `shape-owner?` and `in-session?` predicates). Furthermore, determining the propagation of the first group method implies testing the `shape-owner?` and `same-shape-name?` propagation predicates on remote shapes. Although both predicates are evaluated locally, they have to remotely invoke accessor functions for the fields of the shapes (`shape-owner` and `shape-name` respectively).

In the scenario of Geuze, allowing the interleaving of executions of different invocations may have undesirable consequences with regard to the consistency of the collaborative drawing. For instance, this is the case when interleaving two requests of selection for the same shape (two invocations of the `select-shape` group generic function made from different editors). We have discussed this case in detail in Section 4.5.4. To avoid the interleavings, we annotate each method making a remote invocation with the `:uninterrupt-ible` qualifier. As explained in Section 4.4.2, methods invoking such an uninterruptible method do not have to be annotated.

7.2.7 Evaluation

Using Lambic, we ensure the following properties in the implementation of the Geuze service:

Context Dependency in AmOP. Several parts of the implementation of Geuze show the clean alignment between Lambic's implicit handling of asynchronous remote invocations, and the semantics of dynamic method dispatch and composition. An example of this is the execution of the graphical operations for selecting and moving shapes. This includes processing the GUI events that trigger the graphical operations (`mouse-down`, `mouse-move` and `mouse-up`), and the group generic functions representing such operations (`select-shape` and `move-shape`). Lambic enables these cases to be written in a direct style

#	**Method or group method to be annotated**	**Reason for asynchronous execution**
1	`mouse-down` method specialised on `selecting?`	depends on local execution of method #2
2	`select-shape` group method specialised on `shape-client?` and `in-session?`	depends on local execution of methods #4 and #5, and on **remote** execution of group method #3
3	`select-shape` group method specialised on `shape-owner?` and `in-session?`	depends on local execution of method #5
4	`shape-owner?` method	depends on **remote** execution of `shape-owner` accessor
5	`same-shape-name?` method	depends on **remote** execution of `shape-name` accessor
6	`move-shape` group method specialised on `in-session?`	depends on local execution of methods #5 and #7
7	`relevant-position?` method	depends on local execution of method #8
8	`on-focus?` method	depends on **remote** execution of `shapes-on-focus` generic function

Table 7.3: Methods and group methods processed asynchronously.

while internally still executing the remote invocations in an asynchronous fashion. For instance, the `mouse-down` event handler can implicitly wait for the result of the invocation of the `select-shape` group generic function. This group generic function makes several remote invocations (to contact the shape's owner, to propagate the selection to the session, and to evaluate propagation and context predicates). Yet, these invocations do not obstruct the dynamic selection and composition of group and peer methods.

Geuze avoids that network failures interfere with the execution semantics of group methods. For this, we handle possible timeout exceptions raised by remote invocations. As in the case of Kriek, we find this support in the body of propagation predicates (the `try-catch` form used in the `shape-owner?` and `same-shape-name?` methods). We also handle timeout exceptions raised on the group methods of the `select-shape` group generic function (more specifically, in the group method specialised on the shape's client).

The handling of GUI events is a clear example of Lambic's consistent scope for context-dependent adaptations of behaviour. The invocation of the `mouse-down` generic function results in the selection of one or more of its methods (according to the context predicates associated with the methods). Each selected method invokes (the group generic function of) a different graphical operation. These operations perform several remote invocations causing the evaluation of `mouse-down` to be suspended. However, when the results of such invocations are obtained, the `mouse-down` generic function resumes its execution using the same list of methods initially selected. Also, the execution is resumed at the same point where the group generic function of the graphical operation was invoked.

The fact that the execution of `mouse-down` can be suspended also implies that it can be interleaved with other executions. In the present implementation we disallow this possibility by annotating the methods making remote invocations with the `:uninterruptible` qualifier.

Group Behaviour in AmOP. Lambic's decentralised group behaviour management enables the Geuze service to properly react to dynamic changes of availability of the peer editors. We use the `peer-discovered`, `peer-disconnected` and `peer-reconnected` generic functions to keep a consistent list of the shapes available and in-use at each editor of the session.

Lambic's decoupled model for group communication can also be observed in the Geuze service. In the definition of its group behaviour there are no references to the locations of the editors (space decoupling). The propagation of graphical operations is exclusively asynchronous (synchronisation decoupling). The propagation of operations to disconnected editors is stored until they become available or the timeout is reached (time decoupling). Finally, the editor modifying a shape is unaware of the number of editors the modification will be propagated to (arity decoupling, cf. Section 2.4.1).

Possible disconnections affecting the communication between peer editors are supported by Lambic's time-based network failure handling. This mechanism is used in the definition of the `shape-owner?` and `same-shape-name?` propagation predicates, and the `select-shape` group method specialised on the `shape-client` predicate.

Context Dependency in Group Behaviour. In Lambic, context-dependent selection of group behaviour is achieved by allowing group methods to be specialised on context predicates. In the development of Geuze, we use such context-dependent group methods to provide different strategies to propagate graphical operations. For instance, in the `select-shape` group generic function we define two group methods specialised on two different context predicates (`shape-client?` and `shape-owner?`). This way, we can propagate the selection differently according to whether the editor is or is not the owner of the shape. Furthermore, we associate both group methods with an additional `in-session?` predicate. Thus, we ensure that they are evaluated only when the editor is in a collaborative setting.

In Lambic, the scope of group behaviour is restricted to the execution of an invocation of a group generic function. This scope is preserved even in case of remote interactions suspending the execution. For instance, for an invocation of the `select-shape` group generic function this means that the editor selecting the shape will execute only the `select-shape` group method specialised on the `shape-client` predicate. Similarly, the owner editor will execute only the `select-shape` group method specialised on the `shape-owner` predicate.

Lambic enables the modular definition of Geuze. There is a clear separation between the code required for the editor as a stand-alone and a collaborative service. There is also a clear distinction between the different context-dependent adaptations of behaviour, both at the group and peer levels. This modularity is especially beneficial to deal with the complexity of this service in an incremental way.

Open Issues

In this scenario, we also encounter a number of issues:

- In Geuze, the re-evaluation of context and propagation predicates is even more problematic than in the case of Kriek. An editor propagates every single modification to the shapes (e.g. every shift of position). Each modification corresponds to an invocation of a group generic function (e.g. `move-shape`). The context-dependent propagation strategies alleviate this problem to some extent (e.g. by adding the `position-relevant?` predicate to restrict the propagation of `move-shape`). However, the predicates are still re-evaluated for each invocation of the group generic function. In our implementation, we have solved this issue by introducing a cache mechanism (cf. Section A.7). Incorporating this mechanism to Lambic's model remains an important challenge for future research.

- Identifying which methods to annotate with execution qualifiers may be a non-trivial task, especially because of the asynchrony contagion problem (the clients of methods containing asynchronous remote invocations, are also executed asynchronously). Lambic throws a warning for each method that is not properly annotated with an execution qualifier. However, further support should also be provided to make developers aware of the reasons for the warnings, e.g. based on the dependencies between methods.

- In Lambic, group behaviour is specified on a per-generic-function basis. This has enabled the Geuze service to define different group methods for the `select-shape` and `move-shape` group generic functions. However, in our implementation we have also found group generic functions requiring the same group level definition. For instance, `paint-shape` and `move-shape` are propagated in exactly the same way. As such, they have similar group method definitions.[9] A way to solve this is to define a generic operation that is used for both painting and moving the shape. A similar approach can be found in [MR07]. Still, the question remains whether it is always possible to combine the base behaviour of two operations requiring the same group behaviour. A different approach to this issue is using an aspect-based solution, as the one of AWED (cf. Section 3.3.1).

- Finally, the flexibility gained by specialising group methods on more than one parameter on a group class can also complicate the definition of propagation predicates.

[9]The definition of `paint-shape` can be found Appendix D

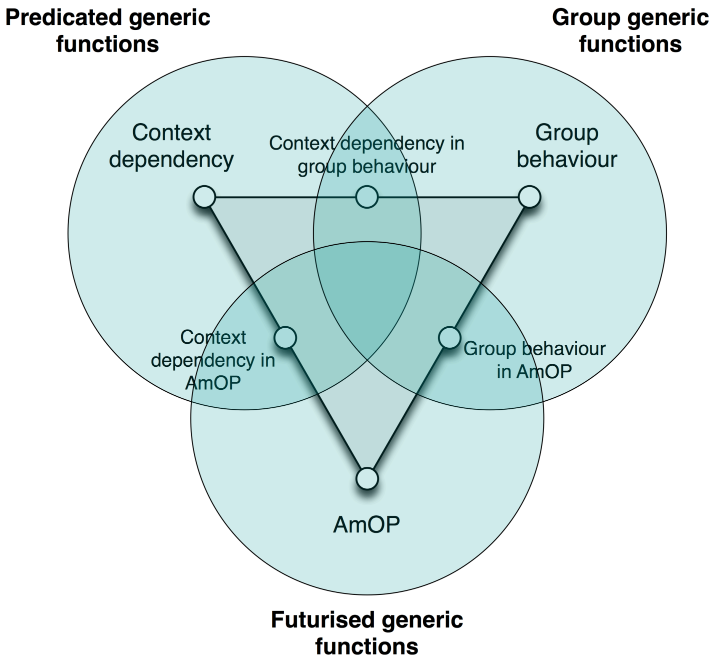

Predicated generic functions
Group generic functions
Context dependency
Context dependency in group behaviour
Group behaviour
Context dependency in AmOP
Group behaviour in AmOP
AmOP
Futurised generic functions

Figure 7.11: Lambic's support for context dependency and group behaviour in AmOP.

7.3 Discussion

We have illustrated Lambic's support for modularity of context dependency and group behaviour in the AmOP paradigm. We now review our model with respect to the requirements listed at the beginning of this chapter. Then, we conclude this chapter by comparing Lambic to the programming language approaches for event-driven distribution, context dependency and group behaviour presented in Chapter 3.

7.3.1 Modularity of Context Dependency and Group Behaviour in AmOP Revisited

Lambic fulfils the requirements for modularity in AmOP by means of three main features: *futurised generic functions* (for event-driven distribution), *predicated generic functions* (for context dependency), and *group generic functions* (for group behaviour). A common underlying execution process ensures the effective integration of the three features. Figure 7.11 shows Lambic's integrated support for these features. This figure is based on Figure 3.2 of Section 3.4 which summarises the requirements for modularity of context dependency group behaviour in AmOP.

Context Dependency in AmOP

By integrating futurised generic functions and predicated generic functions Lambic ensures the modularity of context dependency in AmOP as follows:

$R_{I.1}$ Controlled propagation of asynchrony By providing support for implicit future handling, Lambic allows a controlled propagation of asynchronous executions from both context predicates and super calls. The results of asynchronous remote generic function invocations can be handled using the standard sequential and imperative object-oriented programming style. Still, to be aware of such asynchronous invocations, our model requires that method definitions are properly annotated with execution qualifiers (*interruptible* and *uninterruptible*).

$R_{I.2}$ Controlled propagation of network failures Lambic also allows controlling the effects of network failures affecting asynchronous generic function invocations. First, our model provides a time-based failure handling mechanism (adopted from AmbientTalk [VME+07]) which abstracts the programs' control flows from the volatile network connectivity of pervasive computing services. The effects of network failures on asynchronous remote invocations are manifested in the form of timeout exceptions. These exceptions *ruin* the futures generated by the asynchronous remote invocations. Second, Lambic's implicit future handling allows dealing with timeout exceptions in a sequential style. This means that such exceptions can be implicitly propagated through super calls in the form of ruined futures. This also means that the exceptions can be handled using standard `try-catch` forms (no special asynchronous exception handler is required). However, the exceptions raised during the evaluation of context predicates cannot be propagated to the generic functions. They should be manually handled within the predicates.

$R_{I.3}$ Restricted scope of event-driven behaviour Lambic ensures a consistent activation scope for context-dependent adaptations. This scope is delimited by the execution of a generic function invocation. The adaptation corresponds to the methods selected for the invocation. The activation scope is preserved even in the case of concurrent invocations. In our model, concurrent generic function invocations are processed sequentially, exclusively by the event loop of the actor that contains the generic function. As such, within an actor there can be only one context-dependent adaptation in use at a time. Finally, the asynchronous executions of generic function invocations can still be interleaved. To avoid that such execution interleaving hinders the activation scope of an invocation, our model provides the *uninterruptible* execution qualifier for methods.

Group Behaviour in AmOP

By integrating futurised generic functions and group generic functions Lambic ensures the modularity of group behaviour in AmOP as follows:

$R_{I.4}$ Decentralised group behaviour management In our model, the instances of group classes discover each other in a decentralised fashion. Each instance can be aware of the discovery, disconnection and reconnection of its peers by means of event handlers in the form of method specialised on the group class.

$R_{I.5}$ Decoupled group communication Lambic ensures a decoupled communication model for interactions with the group and interactions between the group members. Group generic functions can be invoked without mentioning the physical address of the instances of the group class (space decoupling). Similarly, the propagation of an invocation to the group class peers is declaratively specified in terms of propagation predicates. Remote group generic functions can be invoked and propagated only asynchronously (synchronisation decoupling). A group generic function can be remotely invoked and propagated even if the group class peers are temporarily unavailable (time decoupling). For such a case, our model buffers the invocation until the connections with the peers are re-established. Finally, the propagation of a group generic function invocation to the group class peers is transparent for the program making the invocation (arity decoupling). No explicit enumeration is required.

$R_{I.6}$ Connection-independent failure handling Lambic's mechanism for time-based failure handling is also used for the propagations of group generic function invocations.

Context Dependency in Group Behaviour

Finally, by integrating predicated generic functions and group generic functions Lambic ensures the following properties:

$R_{I.7}$ Dynamic selection of group behaviour Our model enables associating group methods with context predicates. As such, a group generic function can have several group methods specialised on different context conditions. This allows the definition of advanced coordination schemes among group class peers, as the one shown in the scenario of Geuze (cf. Section 7.2.4).

$R_{I.8}$ Restricted scope of group behaviour Lambic ensures a consistent activation scope at each actor involved in the execution of a group generic function invocation. The applicable methods for an invocation are evaluated independently by each actor.

$R_{I.9}$ Preserving modularity and composability Finally, Lambic preserves the modularity and composability of the peer- and group-level definition of group generic functions. Both levels can have context-dependent variations which are cleanly separated in peer and group methods. These methods are dynamically selected and composed according to the context of the invocations. The methods can also anonymously access each other by means of *super calls* (using the `call-next-method` or `call-peer-method` forms).

7.3.2 Related Work Revisited

We now evaluate the Lambic programming language model with respect to the existing approaches for event-driven distribution, context dependency and group behaviour included in Chapter 3. We apply to Lambic the same criteria we used for reviewing those approaches. Tables 7.4, 7.5 and 7.6 shows the results of the evaluation of Lambic.

Event-driven Distributed Programming for AmOP

Lambic adheres to the AmOP paradigm by adopting the event-driven distribution execution model of AmbientTalk [VME$^+$07], known as the event loop model. In this dissertation, we have studied the language support required for this and other event-driven execution models (Section 3.1). Concerning our focus on modularity, we have observed that such a dedicated support often entails more verbose code and less straightforward control flows. More concretely, in Section 3.1.2 we have identified four issues of the event-driven programming style: *inversion of control*, *lost continuations*, *asynchrony contagion* and *the event interleaving hazard*. We have analysed each of these issues and then reviewed the solutions proposed by existing event-driven programming language approaches.

Lambic's solution to these issues is two-fold. Our model provides explicit syntax for distribution, i.e. for asynchronous generic function invocations and for asynchronous result handling (futures). This syntax complies with the properties of AmbientTalk's event loop model. Additionally, our model provides an internal future-handling process that enables distributed computations to also use the same syntax as for local computations, while internally still executing them in an event-driven manner. In Section 4.5, we have discussed the benefits and limitations of either approach (explicit and uniform syntax). However, it is important to notice that the implicit future-handling mechanism is a key feature for achieving modularity in the AmOP paradigm. This feature allows not only the uniform syntax for local and remote computations. It also keeps the control flows of programs sequential which has facilitated considerably the integration of models for modularising context dependency and group behaviour.

It is also important to note that Lambic's uniform syntax for local and remote computations does not completely hide distribution from the programs. Developers have to still acknowledge the parts of the programs that can be affected by distribution issues (by means of execution qualifiers in methods, cf. Section 4.4.2). However, this explicit support does not interfere with the programs' modularity, as we have shown in the two case studies of this chapter.

Lambic's solution to the issues of event-driven programming style can be compared to existing solutions as follows.

Inversion of Control. Lambic provides two solutions to the inversion of control problem, i.e. to avoid fragmented control flows and manual stack ripping due to the handling of asynchronous events. Our model's explicit syntax adopts AmbientTalk's solution based on *in-line closures* (via the `when-resolved` form). This solution is also found in E, Scala actors and Lua `rpc.async`. In-line closures enable computations that depend on the re-

Model	Event-driven programming issues									
	Inversion of control	Lost continuations	Asynchrony contagion					Event interleaving	Synchronisation decoupling	Failure handling
			obj. def.	meth. def.	msg send	msg rec.	res. hand.			
Lambic's explicit syntax	✓ in-line closures	✓ implicit future resolution/ruin	✓	✓	✗	✓	✗	✗ non-blocking futures	✓ non-blocking event loops	✓ based on time
Lambic's uniform syntax	✓ direct style	✓ implicit future resolution/ruin	✓	✗	✓	✓	✓	✓ uninterruptible methods	✓ non-blocking event loops	✓ based on time

Table 7.4: Lambic's programming style for event-driven distributed communication

Model	Modularity			Dynamic selection	Consistent composition	Restricted scope
	partial behaviour definition	groups of partial definitions	behaviour selection			
Lambic	✓ predicated methods	✓ using same predicate as specialiser	✓ predicates	✓ based on evaluation of predicates	✓ priority order of predicates per generic function	✓ generic function execution

Table 7.5: Lambic's programming style for context-dependent behaviour

Model	Plurality encapsulation		Group protocols						Modularity
			membership		communication				
	communication style	communication interface	group manag.	failure handl.	method propag.	param. passing	results handl.	failure handl.	
Lambic	✓ standard request/reply	✓ member class	✓	✓	✓	✓	✓	✓	✓ group method

Table 7.6: Lambic's programming style for group behaviour

sult of asynchronous invocations, to be defined in the same context where the remote invocation is done. This way, no manual stack ripping is required.

Lambic's uniform syntax provides an *implicit asynchronous continuation management mechanism* similar to those of ProActive, Kilim, TaskJava, Lua `rpc.sync` and JCoBox. In our case, this mechanism corresponds to the implicit handling of futures. No special forms are required to handle the futures. As in the case of Kilim and TaskJava, in Lambic this semantics are allowed only on methods annotated with an execution qualifier (*interruptible* or *uninterruptible*).

Lost Continuations. Avoiding lost continuations implies covering all the possible ways in which an asynchronous request may terminate. Our model tackles this problem by using futures as implicit return address for remote calls (as in AmbientTalk, E, ProActive, JCoBox and Lua). In Lambic, futures implicitly receive the results of asynchronous remote generic function invocations. Futures also allow dealing with the exceptions occurring during the execution of the asynchronous invocations (e.g. timeout exceptions). As in E and AmbientTalk, such exceptions implicitly ruin the corresponding futures. A ruined future can be handled with a dedicated :catch argument in the `when-resolved` forms, in case of Lambic's explicit syntax (as in E and AmbientTalk), and with standard `try-catch` forms in the case of Lambic's uniform syntax (as in ProActive and JCoBox). Thus, the future produced by a remote invocation is guaranteed to always be resolved with a result, or ruined with an exception. Of course, also as in those languages, the exceptions can still be overlooked if the future is not used in further computations.

Asynchrony Contagion. The asynchrony contagion is related to the degree of uniformity in the language support for local and remote interactions (cf. Section 3.1.2). Lambic's explicit syntax exhibits the same issues as AmbientTalk. The problem of asynchrony contagion of this syntax is related to the explicit support for asynchronous remote invocations and result handling. As we extensively discussed in Section 3.2.3, it is the explicit abstraction for handling asynchronous results the main problem for the programs' modularity.

The asynchrony contagion can be observed also in Lambic's uniform syntax. In this case, the asynchrony is propagated through the execution qualifiers for methods (as in Kilim and TaskJava). Not only the method containing remote invocations should provide such annotation, but also other methods invoking the annotated method. This way developers can still be aware of the part of the programs affected by asynchronous executions.

The Event Interleaving Hazard. To cope with event interleaving hazards, Lambic provides the *uninterruptible* execution qualifier for methods. This qualifier allows blocking semantics for claiming futures as in ProActive, JCoBox, Scala and TaskJava. However, in our case the blocking semantics are used only for methods annotated as uninterruptible. This is similar to JCoBox and Scala which allow blocking semantics only through the `get` and `receive` operations respectively. ProActive and TaskJava use blocking semantics as the default behaviour. Our approach is less expressive than Lua's synchronisation constraints which allow for selective reception of events.

Context Dependency

In Section 3.2, we focused our study of context dependency on the solutions of context-oriented programming (COP) language models, i.e. models with dedicated language support to express context-dependent behaviour. Lambic's support for context dependency can be compared to those approaches as follows.

Modularity. In COP approaches, we find three main concerns of context dependency which need to be modularised: *modular partial behaviour definitions*, *modular groups of partial behaviour definitions*, and *modular behaviour selection* (cf. Section 3.2.2). Lambic's units of partial behaviour definition correspond to the predicated methods. These methods can be specialised using programmer-defined context predicates, providing fine-grained control of method applicability, in a similar way to the Predicate Dispatch model. Additionally, method dispatch is driven by the context predicates' precedence order which is fully defined by the methods' generic function. Thus, context predicates can be cleanly associated with methods without getting entanglement, i.e. achieving *modular behaviour selection*. Finally, Lambic does not provide means to explicitly group partial behaviour definitions. Instead, partial definitions can be implicitly grouped by associating them with the same context predicate (as in the Ambience, Filtered Dispatch and Predicate Dispatch models). However, as we discussed in Section 7.2.7, one of the drawbacks of implicit groups of partial definitions is that it is less simple to model state that is shared by several context-dependent adaptations. This is unlike approaches such as ContextL, PyContext, and JCop, where the state can be explicitly associated to layers (those models' explicit groups of partial definitions).

Dynamic Selection. In Lambic, context-dependent adaptations are represented as predicated methods which are dynamically selected as a result of the method dispatch mechanism. This is similar to most of COP approaches. In particular, in Lambic the selection and combination of methods is computed in accordance to the predicates and their order of declaration in the generic functions. This solution is based on the Filtered Dispatch model which alleviates the limitation of the Predicate Dispatch model to include user-defined orderings of predicates.

Consistent Composition. Consistently composing context-dependent behaviour corresponds to ensuring an unambiguous combination between the different partial definitions required to process a method execution. In Lambic, generic function-based priorities between predicates ensure the consistent composition of methods, as in Filtered Dispatch. These priorities alleviating the issue of combinatorial explosion of context-dependent adaptations (cf. Section 2.3). Still, this mechanism is less expressive than ContextL's support for composition rules. Indeed, the priority order between predicates can be modelled as one composition rule in ContextL. Finally, the composed methods can access each other anonymously, by means of *super calls* (the `call-next-method` form), as in ContextL, PyContext, CDR, Ambience, JCop and Filtered Dispatch.

Restricted Scope. As in most models for context-oriented programming, Lambic delimits a context adaptation to the scope of a generic function invocation (similarly to the PyContext, CDR, Filtered Dispatch and Predicated Dispatch models). Additionally, as discussed in Section 7.3.1, this scope is unambiguous even in the presence of concurrency and event interleaving. To the best of our knowledge, Lambic is the only model for context dependency that ensures this second property.

Note that our approach shares several properties with existing models for context-oriented programming, especially with those based on CLOS (Filtered Dispatch and ContextL). Still, instead of extending such approaches we decided to create our own extension of CLOS. This extension gave us full control on the implementation which facilitated the integration between the support for context dependency and those required for ambient-oriented programming and group behaviour.

Group Behaviour

Lambic's support for group behaviour can be compared to existing approaches as follows.

Plurality Encapsulation. As explained in Section 3.3, the main purpose of plurality encapsulation is to access a remote service without regard for the number of objects that provide the service. In Lambic, group generic functions encapsulate the group behaviour for objects as such the group concern is abstracted from the communication. A client can interact uniformly with a service represented by one object or a group of objects. No special group identity, interface or communication style are required. Such an encapsulation is achieved only by the Gaggles, Mailer/Encapsulator and AWED models.

Group Protocols. In Lambic, group protocols for membership and communication are aligned with the requirements of the AmOP paradigm, as in the Ambient References model. As explained in Section 7.3.1, Lambic provides a decentralised group behaviour management, decoupled group communication, and connection-independent failure handling. Our model dos not provide special parameter passing semantics for group communication as that of the AWED and Typed Groups models. In Lambic, groups are not first class entities. Invocations are addressed always to one instance of a group class.

Modularity. Lambic separates the group behaviour from the base functionality of the group classes. Group behaviour is contained in group methods whereas the base functionality can still be defined in terms of standard (peer) methods. Group and peer methods are dynamically selected, sorted and combined according to the classes used as parameter specialisers, and according to the context predicates associated to the methods. Applicable group and peer methods can uniformly access the next most specific by means of the `call-next-method` and `call-peer-method` forms. As explained in Section 3.3.2, most of the approaches for group behaviour studied in this dissertation contain the group protocols in dedicated entities (except from Gaggles and Typed Groups). Such entities can be adapted without losing their modularity. A distinguishing characteristic of Lambic is that the group and base behaviour of programs can be defined uniformly in terms of classes and methods. Most of the support required for group classes and group methods is implicitly provided by Lambic's execution model. Only the propagation expression is added to group methods which enables declaratively specifying propagations of invocations to members of group classes. Still, this expression consists of a number of invocations to standard generic functions. This is unlike AWED where group behaviour is defined using aspects and Mailer/Encapsulator model which defines the group behaviour in meta classes.

Chapter 8

Conclusion

This dissertation has presented the results of our study on modularisation techniques for pervasive computing. In this chapter, we revisit the research goals as stated in the introduction (cf. Chapter 1). We then restate the contributions of the Lambic programming model. Finally, we discuss the limitations of our work and avenues for further research.

8.1 Research Goals Revisited

We briefly review the research goals stated in Section 1.3 and discuss the extent to which they have been achieved.

- It was our goal to study the effects of pervasive computing in the identity and behaviour of software services. In Chapter 2, we stated that in pervasive computing software entities providing similar functionality should abstract from specific locations and identities, and run in the environment as a single service. We called this property the *pervasive identity* of services. We classified the issues entailed by pervasive identities in three major concerns: *distribution*, *context-dependent behaviour*, and *group behaviour*. For each concern, we presented a list of requirements found in existing programming language research literature. We finally argued that the three concerns are tightly interconnected, and that the lack of integrated support to cope with them may lead to programs that are hard to maintain and extend. We referred to this problem as the need for modularity in pervasive computing.

- It was our goal to review the interactions between the requirements for event-driven distribution, context dependency and group behaviour. In Chapter 3, we reviewed the state of the art for each of the three concerns. We argued that the programming style required to support event-driven execution is the main source of conflicts for achieving modular programs. This claim was based on the analysis of four issues of the event-driven programming style: *inversion of control*, *lost continuations*, *asynchrony contagion* and *event interleaving*. We studied the implications of these issues for modularising the programs' context-dependent adaptations and group behaviour. We focused this study on the distribution model proposed by the ambient-oriented programming paradigm (AmOP) —the starting point of our research. We finally proposed a list of requirements for a unified programming model for event-driven distribution and modularity.

- It was our goal to extend the object-oriented programming paradigm with integrated support for distribution, context dependency and group behaviour. In Chapters 4, 5 and 6, we achieved this goal by means of our proof by construction, the *Lambic* object-oriented programming model. We highlight the properties of this model in the following section. Finally, in Chapter 7 we validated Lambic by means of two case studies of pervasive computing services.

8.2 Lambic in a Nutshell

Lambic is an object-oriented programming model for modularity in pervasive computing. It extends the generic function-based object model of Common Lisp with support for the concerns identified in Chapter 2. For concurrency and distribution, Lambic integrates the properties of the AmOP paradigm, in what we call *futurised generic functions* (cf. Chapter 4). For context dependency and group behaviour, Lambic provides two other extensions, called *predicated generic functions* (cf. Chapter 5) and *group generic functions* (cf. Chapter 6) respectively. Additionally, a underlying unified execution process ensures that these three features can be effectively used in combination.

Futurised Generic Functions. Futurised generic functions are Lambic's embodiment of the actor-based model proposed by the AmOP paradigm, called communicating event loops (cf. Section 3.1.1). In Lambic, actors define boundaries of concurrent execution for the objects; methods are specialised exclusively on objects (not on actors). Inter-actor computations are realised by means of asynchronous generic function invocations. And the results of functions evaluations are asynchronously returned to the actor from which the functions are invoked.

As in the original event loop model, Lambic provides explicit syntax for distribution. This includes dedicated abstractions for asynchronous generic function invocations and for asynchronous result handling (futures). Additionally, our model features an internal future-handling process that enables distributed computations to use the same syntax as for local computations, while internally still executing them in an asynchronous manner. This implicit syntactical support is a key property to achieve modularity in programs.

Predicated Generic Functions. Predicated generic functions are an extension of the multiple dispatching mechanism of CLOS. In Lambic, method definitions can be guarded by predicates which are used to decide on the applicability of the method for a list of actual arguments. If more than one predicated method is applicable, the order in which the predicates are declared in the corresponding generic function is used as a tiebreaker.

This allows for fine-grained control of applicability and specificity of methods according to the programs' execution context.

Group Generic Functions. Group generic functions are a novel approach to object-oriented group behaviour in pervasive computing. The key property of this model is to make the group behaviour part of the shared definition of services. We name this property *empathic group behaviour*. Group generic functions provide two main abstractions: group classes and group methods. A group class allows its instances to be aware of each other. Invocations addressed to one instance are implicitly propagated to the other instances of the group class. This way, the instances can cooperate in the execution of invocations, while the programs making the invocations can be oblivious to such a cooperation. Propagation of invocations are declaratively defined in group methods. As such, the group behaviour is decoupled from the base functionality of the group class which is still defined in standard CLOS methods.

8.3 Contributions

Having explained our approach in detail all through this dissertation, we now restate the high-level view of the contributions of our work introduced in Chapter 1.

Modularity in Ambient-oriented Programming. In the context of pervasive computing, we augment the ambient-oriented programming paradigm with support for software modularity. We identify a list of requirements for an integrated model for event-driven distribution, context dependency and group behaviour. We claim that language support required for asynchronous program execution and network failure handling should not hinder the semantics of dynamic method dispatch and inheritance-based composition. The scope for context-dependent adaptations of behaviour should be consistent even in the presence of concurrent and interleaved event-driven interactions. The distribution properties of the AmOP paradigm (decentralised discovery, decoupled communication, connection-independent failure handling) should also be preserved at the group level definition of the programs. Finally, the modularity required for context-dependent behaviour should not hamper the modularity required for group behaviour.

A Generic Function-based Model for Event-driven Distribution. Lambic's futurised generic functions gracefully align the multiple dispatch semantics of generic functions with event-driven distribution. Our model accomplishes the properties of ambient-oriented programming (cf. Section 2.2). Services are discovered using a decentralised publish/subscribe protocol. No address or location is required. Events for discovery, communication and failure handling are all represented as generic function invocations. Communication events correspond to asynchronous generic function invocations. Asynchronous generic function invocations are resilient to partial failures. When an actor that is supposed to process a remote invocation becomes disconnected, the invocation is buffered until the connection is restored.

Explicit and Uniform Language Support for Event-driven Distribution. Futurised generic functions provide explicit and uniform syntax for communication, while guaranteeing the event-driven execution of remote interactions. Chapter 4 presents an in-depth analysis of the benefits and limitations of either syntax with respect to the issues of event-driven distribution (cf. Chapter 3). We conclude that explicit syntax helps developers to better understand the effects of asynchronous remote interactions in the programs. However, it often entails more verbose code and less straightforward control flows. Uniform syntax, on the other hand, enables the programs to be less verbose and to keep control flows sequential but the support for dealing with remote interactions is more restricted, and in some cases more error-prone. Finally, although our aim is not to advocate the use of a particular kind of syntax, we do rely on the implicit handling of asynchronous results to modularise context-dependent adaptations and group behaviour, as we explain in the following contributions.

Generic Functions with Context Predicate Dispatch. Predicated generic functions allow for modular definition of context-dependent behaviour. Modularity is achieved by enabling context-dependent adaptations to be expressed as predicated methods. These methods can be specialised on programmer-defined context predicates, providing fine-grained control of method applicability. Method dispatch is driven by the context predicates' precedence order which is fully defined by the methods' generic function. These methods selected for an invocation can anonymously access each other only by means of "super calls" (using the `call-next-method` form). Lambic delimits a context-dependent adaptation to the scope of a generic function invocation. This means that context predicates are evaluated for every invocation, ensuring that the selected methods are always consistent with the execution context.

To cope with event-driven distribution, we integrate Lambic's internal future handling mechanism with the semantics for determining the effective method for invocations. This way, our model can implicitly handle possibly asynchronous results returned by context predicates and super calls. With respect to the propagation of network failures, Lambic allows timeout exceptions to be propagated through super calls (in the form of ruined futures). However, these exceptions must be explicitly handled if they are raised by the predicates. Finally, Lambic's event loop model preserves the per-invocation scope of context-dependent adaptations, even in case of concurrent interactions. This is because actors handle only one invocation at a time.

Generic Functions for Group Behaviour. Lambic's group generic functions allow for modular definition of group behaviour. Our model explicitly separates group behaviour (contained in group methods) from base functionality of the programs (contained in standard methods). When a group generic function is invoked, the group and peer methods are dynamically selected, sorted and combined. Additionally, group generic functions encapsulate group behaviour for objects. As such, the group concern is completely abstracted from the communication. Clients can interact uniformly with a service represented by one or a group of objects. Our model needs no special group identity, group interface, or group communication style.

Lambic preserves the properties of the AmOP paradigm also for the definition of group behaviour. Distributed group classes discover each other in a decentralised fashion. Both interactions with the group and interactions between group peers use a decoupled communication model. As in the case of predicated generic functions, our model handles the asynchronous remote results of the propagation implicitly. Yet, network failures (i.e. timeout exceptions) should be manually handled in the group methods. Finally, the group and base level behaviour of group generic functions can have context-dependent variations. These variations are modelled as independent base and group method definitions specialised on different context predicates.

Pervasive Identities. We validate Lambic by implementing two concrete pervasive computing services. These services have been conceived as pervasive identities as both required the natural integration of nearby services to provide ubiquitous access to their functionality. In the scenario of Kriek, we use a pervasive identity to enable a single-user service to dynamically distribute its functionality among the user's devices. In the scenario of Geuze, the pervasive identity allows for realtime group collaboration. It integrates the work of several users ensuring consistency between their concurrent interactions.

8.4 Work influenced by our Research

The direct results of our research have been presented in [VED+07, VCVD09, VGC+10]. Additionally, other research have been influenced by the work presented in this dissertation.

Context-dependent Composition of Web Services. In her master thesis [Hua09], Huang applies Lambic's concepts for dynamic compositions of web services (known as *mashups*). The main observation is that mashups rely on event-driven communication models to increase responsiveness in programs and to deal with eventual failures affecting remote requests. AJAX asynchronous requests [Gar05] are a prototypical example of this. The problem with such an approach is that it suffers from inversion of control (the control flow of the programs is broken up into several callbacks, cf. Section 3.1.2). Such fragmented control flow is particularly awkward for mashups as it obscures the tasks they accomplish.

Huang's work demonstrates that Lambic's implicit handling of event-driven distributed interactions enables the definition of mashups using the standard imperative and sequential object-oriented programming style. She develops a Lisp-based framework, called DYMAC, in which web services are accessed using the same syntax as local function invocations. Internally, the invocations are converted into asynchronous AJAX requests. DYMAC also handles the results of the asynchronous requests implicitly. This way, the mashups do not require to deal with continuation-passing schemes (e.g. AJAX callbacks). This significantly simplifies the mashups' control flows. The initial results of this research have been presented in [VHC+10].

Context-dependent Service Migration and Partitioning. The notion of pervasive identities put forward in this dissertation has been explored in the context of dynamic service partitioning in pervasive computing. In his master thesis [Bai08], Bainomugisha observes that a way to maximise the use of resources found in the user's surroundings, is by decomposing services into parts which can be dynamically distributed to different devices. He also observes that current service partitioning approaches are based mostly on low-level and static operations, and have no support for network failures. To solve this problem the Resilient Actor model has been proposed. In this model, applications are defined as a hierarchy of actors that are interconnected thorough elastic bindings. Such bindings are special kinds of remote references which can be *stretched*, i.e. enabling the actors to be moved to new locations, and *retracted*, i.e. to move back the actors to their original location. In case of disconnections, an automatic retraction mechanism is triggered which regenerates the affected part of the application as it was before the partitioning. Finally, this model proposes a set of resilient strategies with different implementations of the stretching and retraction operations.

This research has been developed in the AmbientTalk programming language and presented in [BCC$^+$10].

Self-adaptability in Context-aware Systems. In [CMV$^+$08], we integrate Lambic's ideas of context-dependent behaviour into a peer-to-peer adaptable topology for pervasive computing, called PALTA. PALTA allows the incremental construction of distributed networks according to the current network state. This model provides an algorithm that takes advantage of the best features of a fully connected network when the number of peers is small enough to allow the devices manage this kind of topology. When the network becomes too large to maintain a fully connected topology, the algorithm will automatically adapt the network configuration to become a Relaxed Ring [MR07]. This configuration can handle a large number of peers by executing more complex algorithms for self-managing the distributed network. At the moment, PALTA has been implemented in the Mozart-Oz programming system [VH04] which contains the support for the relaxed ring configuration. To provide this dynamic adaptability, PALTA borrows our model's ideas of having separate peer connectivity event handlers for different execution contexts (sizes of the peer group).

8.5 Limitations and Future Work

The assessment of Lambic presented in Chapter 7 leads us to believe that our model is well suited to the expression of context dependency and group behaviour in pervasive computing. However, a number of issues need to be further explored. In this section, we discuss the limitations and avenues for future work.

8.5.1 Allowing Efficient Predicate Evaluation

In Chapter 7, we explained that a major issue in our solution is that the intensive use of context and propagation predicates can make the evaluation of programs less efficient.

Moreover, by enabling predicates to include invocations of remote generic functions, we also increase the communication overhead and make predicate evaluation more vulnerable to network failures. In the past, efficient implementation techniques for predicate dispatch have been thoroughly investigated [CC99, Mil04]. Those techniques use static information about the programs to reduce the expected time for dynamic method dispatch. However, as we discuss in Section 3.2.1, such approaches require that predicates use only compile-time constant expressions. This contradicts Lambic's principle of enabling predicates to test runtime values of arbitrary expressions. Still, the possibility exists that programs in Lambic do not rely completely on dynamic context conditions. We should therefore carefully evaluate what part of that static analysis can be used in our setting.

Another solution is to use a caching mechanism that enables predicates to be re-evaluated only when it is strictly needed.[1] This solution implies deciding the extent for which the cached values are available in the program execution. For instance, such an extent could correspond to the execution of a generic function invocation. Thus, tests included in predicates would be evaluated only once, even if they are used in more than one predicate. Alternatively, the re-evaluation of tests could depend on specific changes on context conditions (time, location, etc.). As such, the cached values could remain available for several invocations. In Section 4.5.5, we explain this case when using a GeoIP service (a program needs to request its current location only if its IP address changes).

Dealing with cached values also implies deciding a protocol for updating them. This is especially relevant for caching results of remote tests, as discussed in Section A.7. The update of such a cached result can be triggered by the program invoking the remote test (the client program), or by the program processing it (the service program). Examples of the former case are the requests to the GeoIP service (the client program determines when to update the location). Examples of the latter case are drawing editors which proactively propagate the changes in their state to the session (so that predicates testing such state do not need make remote invocations).

8.5.2 Managing Advanced Context Dependencies

By enabling context predicates to have a per-generic function precedence order, Lambic gives developers fine control over the specificity of context-dependent adaptations. Still, additional support is needed to express further relationships among the adaptations. We have identified a number of those relationships in our work on requirement analysis for context-aware systems [DVC$^+$07, DVV$^+$07]. Apart from priority order, context-dependent adaptations can establish relationships such as *inclusion* (an adaptation requires the behaviour of another one), *exclusion* (an adaptation precludes the applicability of another one), and *conditional dependency* (the applicability of an adaptation depends on the return value of another adaptation). In the future, we plan to extend Lambic with language abstractions that support the definition of such relationships. In this endeavour, we should ensure that the specification of more advanced relationships between context-

[1]CLOS already uses a similar technique for optimising the computation of the effective method for an invocation of a generic function [BDG$^+$88].

dependent adaptations, does not undermine the understandability of programs promoted in this dissertation. A similar effort has been made by Costanza and Hirschfeld in [CD08] by extending ContextL with a high-level feature description language.

In Lambic, such relationships could correspond to dependencies between context predicates. Such dependencies could be defined by developers in the same way they define the predicates' priority order. An important issue we need to study carefully, though, is that the tests included in context predicates can also create implicit dependencies between the predicates.

8.5.3 Increasing Behaviour Reusability

In Lambic, the programs' base functionality, context-dependent adaptations and group behaviour, are uniformly modelled as methods. Reusability is achieved by enabling the dynamic composition of such methods for the execution of a generic function invocation. Methods are ordered from most to least specific, and less specific methods are accessed using the `call-next-method` form. The benefits of this uniform composition mechanism have been largely illustrated in the validation of this dissertation (cf. Chapter 7).

In the future, we will continue exploring the possibilities of reusability in our model. We are particularly interested on studying the reusability of behaviour spanning methods of several generic functions. Examples of such a behaviour are the coordination protocols for object groups, such as the one used by the Geuze drawing editor (cf. Section 7.2.4). While in that example we had to manually encode this protocol, in the future we envision that a library of object group protocols (as that of Mailer/Encapsulator and DAC models, cf. Section 3.3.1) will be provided.

Another alternative for increasing reusability in Lambic is also discussed in the context of the Geuze drawing editor. It consists of allowing two or more methods of possibly different generic functions to share a context-dependent adaptation or a group behaviour definition. In the drawing editor, such a shared definition is required for the `move-shape` and `paint-shape` group generic functions which have exactly the same group-level definition. Today, such support is found in meta-level programming (e.g. CLOS' metaobject protocol [BDG$^+$88]) and aspect-oriented programming models (e.g. the AWED model, cf. Section 3.3.1).

8.5.4 Modularising State

Thus far, Lambic does not provide any means to associate state with specific context-dependent adaptations or group behaviour definition. In the example of the Geuze drawing editor, this limitation led us to model all state (even temporary values) as fields of the classes of the system. Alternatively, the state could be contained in auxiliary variables or classes. However, in this case developers would still have to remember the variables and classes that concern each adaptation.

In the future, we plan to extend Lambic with support for context- and group-specific state. A similar approach is provided by ContextL, PyContext and JCop (cf. Section 3.2.1). This solution would not only support the modularity of both behaviour and state. This would also allow modelling variables that may have different values according to the context (known as *contextual values* [CH05, Tan08]).

8.5.5 Detecting Event Interleaving Hazards

Finally, in Section 4.5.4 we observe that the effects of event interleaving in programs are particularly difficult to detect. Lambic can avoid interleavings by providing blocking semantics for methods annotated as *uninterruptible* (cf. Section 4.4.2). Finer control can be found in other approaches which enable the use of blocking semantics for specific method invocations, as in the cases of Lua's synchronisation constraints [SRRB10] or JCoBox's blocking claims of futures [SPH10] (cf. Section 3.1.2). However, identifying which computations require such blocking semantics remains an issue.

In future, we plan to research tool support for detecting event interleaving hazards in programs. Among other information, such support would have to identify the points in the program execution where interleavings can occur and the state that can be compromised to each case. Additionally, to bear the asynchronous contagion problem (cf. Section 3.1.2), the tool support will also have to be able to trace chains of invocations until finding the one that is causing the asynchronous execution. In Section 7.2.6, we make this tracing manually for the example of the Geuze drawing editor.

Appendix A

Lambic in Common Lisp

This chapter presents the implementation of the Lambic model in the Common Lisp programming language. For this, we assume that the reader is aware of the semantics of CLOS, especially of its metaobject protocol. First, we provide a general overview of Lambic. We then describe the implementation of our model's main components: futurised generic functions, predicated generic functions and group generic functions. We explain generic function-based event loops independently from futurised functions, even though conceptually they are only one model of distribution (cf. Chapter 4). We discuss the main issues found when combining all these components. Finally, we introduce a prototypical caching mechanism built on top of Lambic to improve performance and resilience to network failures of predicate evaluation.

A.1 Overview of Lambic Programming Model

Figure A.1 shows an overview of Lambic. In this figure, the different components of our model are represented as layers. Such layers are piled up according to their dependencies:

- At the bottom of this pile there is the Common Lisp layer which contains the implementation of CLOS (LispWorks® [Ric90]) and the libraries used by Lambic.
- The next layer describes Lambic's generic function-based event loop model. We classify its constituents according to four main concerns: parameter passing, concurrency, discovery, communication, connectivity and conditions.
- On top of event loops we implement futurised generic functions. This layer includes the classes and operations that support Lambic's explicit and uniform programming style for event-driven distribution.
- Next, we present the layer denoting the implementation of predicated generic functions.
- Finally, at the top of the pile there is the layer for group generic functions.

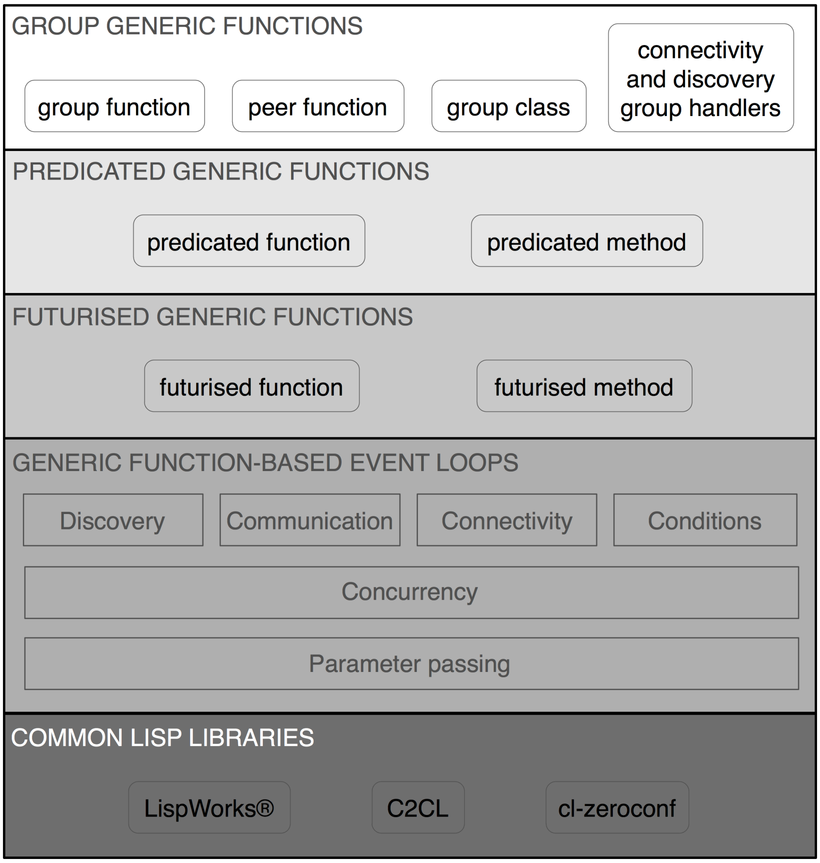


Figure A.1: Overview of implementation of Lambic.

A.2 Generic Function-based Event Loops

Lambic's implementation for event-driven distribution is an adaptation of the event loop model of the AmbientTalk programming language. It includes support for parameter passing, concurrency, communication and discovery.

A.2.1 Parameter Passing

In Section 4.3.1, we explain that when an object is passed as an argument or return value of a remote generic function invocation, it is automatically converted into a remote reference. Conversely, when a remote reference is passed to the actor of the referenced object, it is converted into a local reference to the object. To achieve this behaviour, Lambic provides two evaluation strategies which are denoted by the `value-object` and `distributable-`

object classes. The value-object class implements pass-by-value semantics and is used to define the far-reference class. The distributable-object class, on the other hand, implements pass-by-reference semantics. It is the root class in Lambic. When an instance of distributable-object is passed as a parameter in a remote generic function invocation, it is implicitly replaced by an instance of remote-reference.

Each actor has a list of the distributable objects it contains. This allows the actor to check whether a received remote reference designates one of its contained objects. In such a case, the instance of the remote-reference class is replaced by the instance of distributable-object class. This behaviour is defined in the print-object method of both classes. Such a method is implicitly called when objects are serialising over the network.

A.2.2 Concurrency

Lambic's event-driven concurrency model is implemented by two distributable classes: actor and future. The behaviour of the actor class consists mainly of three generic functions: send-actor-message, receive-actor-message and process-actor-message. The send-actor-message generic function is invoked by the in-actor-of macro. This generic function has two methods which handle local and remote asynchronous function invocations. These methods specialise a receiver argument on the distributable-object and far-reference classes respectively. In both cases, the asynchronous invocations are converted into messages. Remote asynchronous invocations are handled by the communication layer (to transmit the invocations over the network). Local asynchronous invocations are handled by invoking the receive-actor-message function which puts the message in the event queue of the actor. Both send-actor-message methods return a future as immediate result. It corresponds to an instance of a future class.

The event loop of an actor in Lambic is represented by a process which is provided by the Multi-Processing library of LispWorks. The actor's event queue corresponds to the mailbox of such a process. The spawn-actor macro creates an instance of the actor class and initiates a process which continuously waits for a message in its mailbox. Upon message reception, the process calls the process-actor-message function of the actor which converts the message into a generic function invocation, invokes the function, and sends the result to the actor that contains the future attached to the message (if any).

Asynchronous return values are achieved by means of the functions when-resolved and resolve-with-result defined for the future class. when-resolved is directly used in Lambic programs and receives a future and a function that is executed if the value field of the future is bound, or stored in the continuations list otherwise. The resolve-with-result function is asynchronously invoked when the actor that processes the asynchronous invocation for which the future serves as the return value's placeholder, produces the result.

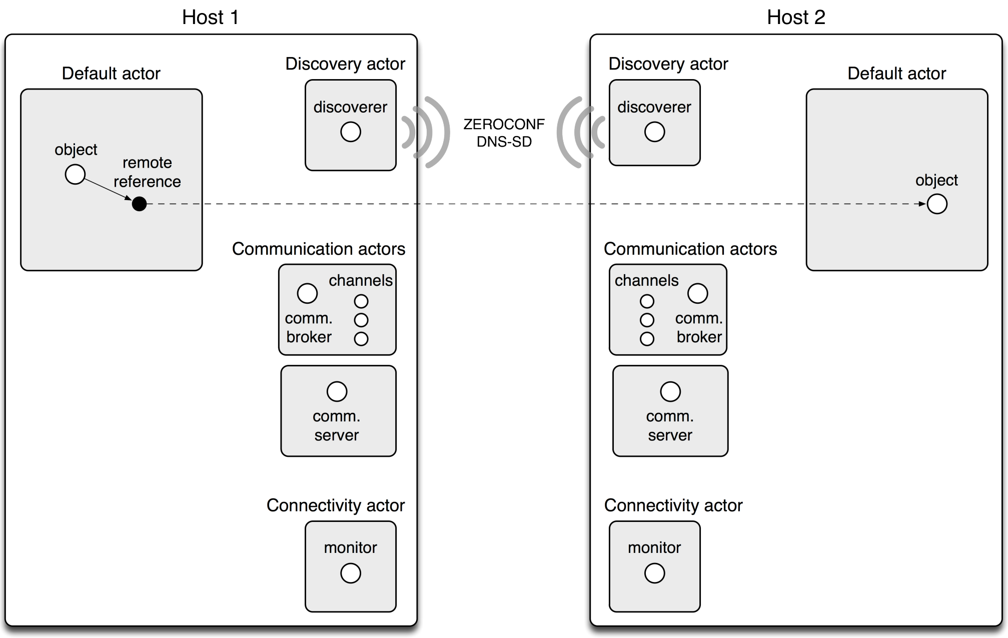


Figure A.2: Event-driven communication and discovery in Lambic.

A.2.3 Communication

Lambic's implementations of service discovery, communication and connectivity are contained in independent actors. Figure A.2 shows these actors. The support for communication is split into two actors. One actor is responsible for sending remote generic function invocations over the network. The other is in charge to receive them. In the behaviour of the actor sending remote invocations there are two classes: `communication-broker` and `channel`. The `communication-broker` class is responsible for handling all asynchronous invocations of remote generic functions made by any actor in its host. At each host there is only one instance of the `communication-broker` class (bound to the `*communication-broker*` variable). Each actor of the host has access to such a variable. The `channel` class contains the behaviour to transmit messages over the network. For this, it uses a TCP/IP socket provided by the Communication library of LispWorks. There is an instance of `channel` per remote object. As we explain later in this section, this granularity is important to handle connectivity events for each service.

The actor in charge of receiving remote invocations corresponds to a socket server provided by the Communication library of LispWorks. It has its own event loop. Communication with this actor is possible only by means of asynchronous generic function invocations.

Figure A.3 shows the sequence diagrams of the operation of handling asynchronous

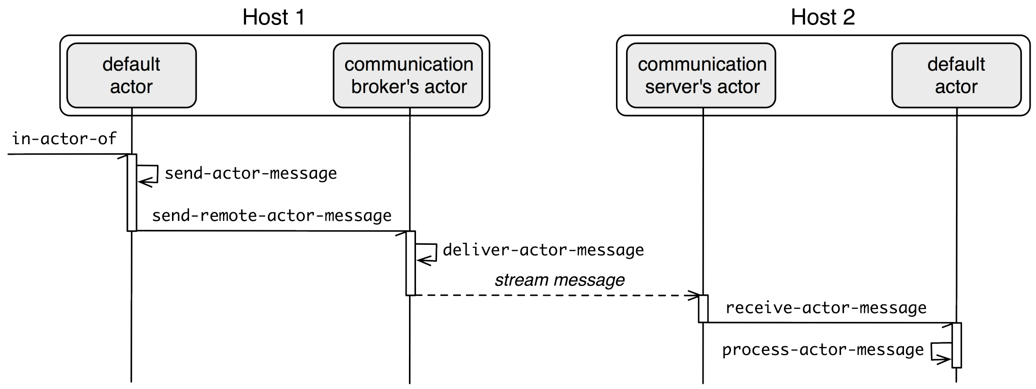


Figure A.3: Sequence diagram for remote communication.

invocations of remote generic functions. It consists of the following steps:

- An actor (the *default actor* at *Host 1* in the example) evaluates an `in-actor-of` expression. It is converted into an invocation of the `send-actor-message` generic function. Assuming that it corresponds to a remote invocation, the actor invokes the `send-remote-actor-message` generic function in the communication broker's actor. This generic function receives as arguments the broker (the `*communication-broker*` variable) and the message containing the remote invocation.

- The broker's actor processes the invocation of `send-remote-actor-message` by finding the channel to the remote object used as message receiver (and creating a new channel if it is not found). Then, the actor invokes the `deliver-actor-message` generic function which is defined for the `channel` class, and which eventually streams the message through the socket.

- The message is received by the socket server. It forwards the message by invoking the `receive-actor-message` generic function on the actor containing the message receiver (the *default actor* of *Host 2*).

- Finally, this actor handles the message (and thus the remote invocation) by calling the `process-actor-message` generic function.

A.2.4 Connectivity

Service connectivity is handled in an actor that contains an instance of the `monitor` class. This class' main function is to periodically check the status of the connection of remote services. This information is used to notify connectivity observers and to raise timeout exceptions.

Figure A.4 shows the sequence diagram of an interaction with the connectivity actor. This diagram includes the following steps:

- Observers for the connection status of a remote service are registered at the default actor by means of the `when-disconnected` and `when-reconnected` generic functions. These observers are managed by the communication broker's actor. This is done by invoking the `add-disconnected-observer` and `add-reconnected-observer` generic functions respectively.
- The broker's actor processes such invocations by associating the observers with the channel denoting the remote service.
- At the connectivity actor, the monitor checks the connection status of the remote references associated with each channel of the broker's actor. Disconnected references are notified by calling the `service-disconnected` generic function on the broker's actor.
- The broker's actor reacts to a disconnected service by disconnecting the channel that contains the reference, and notifying the disconnection to the corresponding observers (calling the `disconnect-channel` and `notify-disconnection` generic functions respectively).
- At this point, messages using the disconnected reference as receiver are queued in the reference's corresponding channel.
- Reconnected references are notified by the monitor, invoking the service-reconnected generic function on the broker's actor.
- The broker's actor reconnects the channel of the reference and transmit the queued messages. Finally, this actor notifies the reconnection to the observers.

A.2.5 Discovery

Lambic's support for service discovery relies on a library called CL-ZEROCONF [Wis05]. This library is a bridge to Apple's implementation of the ZEROCONF peer-to-peer protocol, called Bonjour [App11]. We use CL-ZEROCONF for publishing and subscribing to services. This behaviour is defined as part of the `discoverer` class. Each host has an instance of this class which is contained in the discovery actor. This instance is bound to the variable `*discoverer*` which is accessible at every actor of the host.

Figure A.5 shows a sequence diagram containing the two operations of discovery available in Lambic: exporting objects as services (using the `export-service` form), and adding discovery observers (using the `whenever-discovered` form). This diagram comprises the following steps:

- At *Host 1*, `whenever-discovered` expressions are processed by invoking the `add-discovery-observer` generic function on the discovery actor. This generic function receives the `*discoverer*` variable and a service description as arguments.

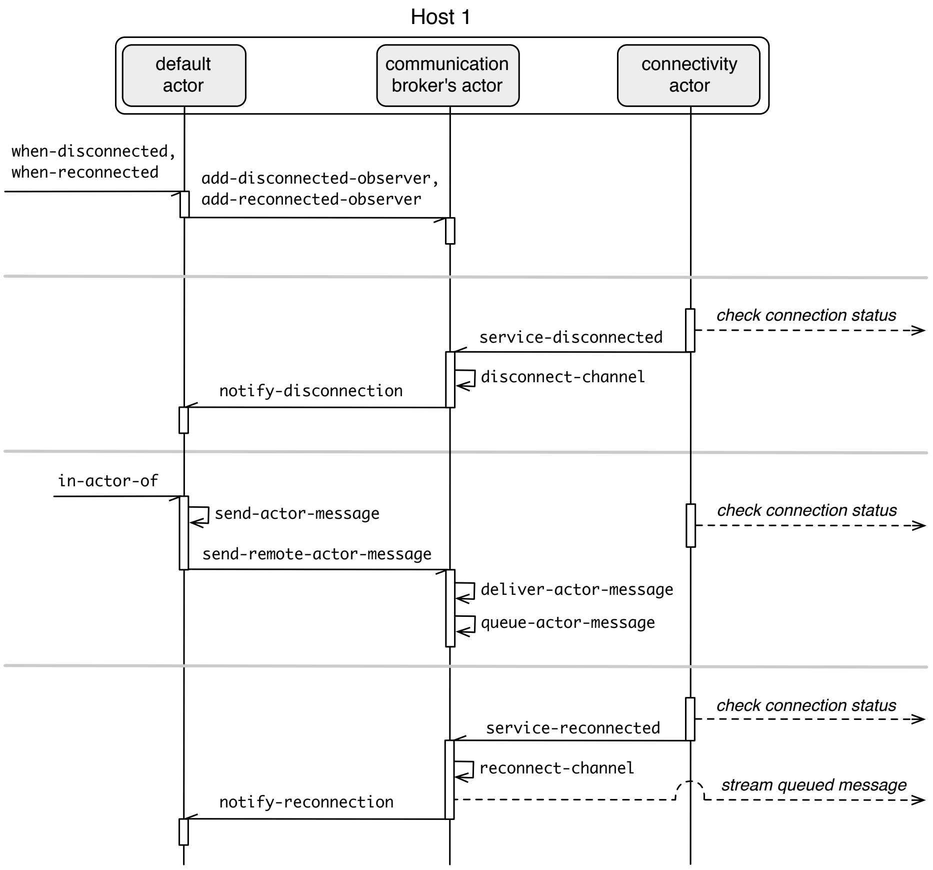


Figure A.4: Sequence diagram for connectivity management.

- The discovery actor handles such invocations by requesting CL-ZEROCONF to browse for every service accomplishing with the service description.

- At *Host 2*, an object with such a service description is exported using the `export-service` form. It invokes the `publish` generic function on the discovery actor, which in its turn asks CL-ZEROCONF to publish the service on the network.

- When the remote object is discovered by *Host 1*, it is informed to the communication actor (calling the `service-connected` generic functions) so that it adds a new channel for the discovered remote reference. The discovery is also notified to the observers of the service description provided by the remote object (calling the `notify-discovery` generic function).

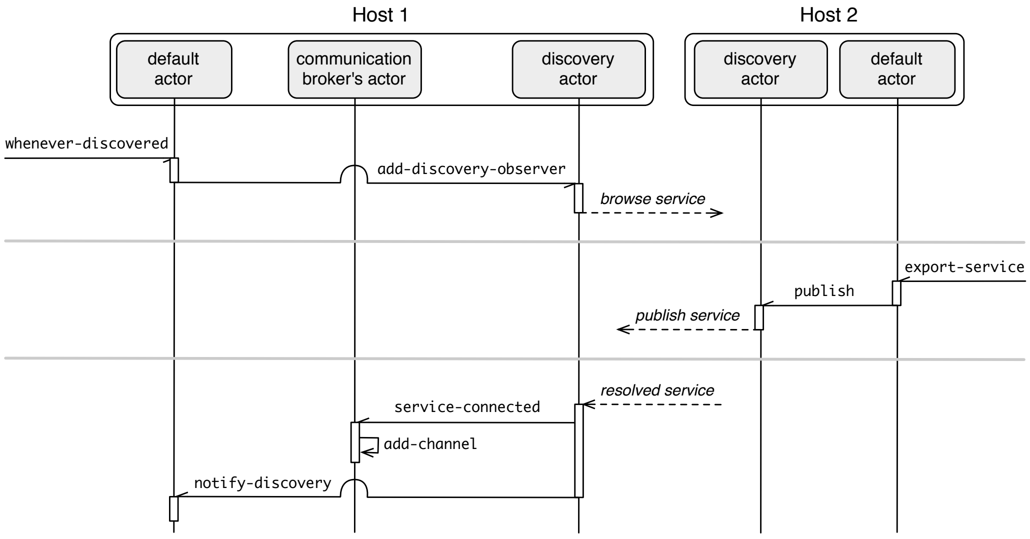


Figure A.5: Sequence diagram for discovery.

In the previous diagram, we show a particular combination of discovery operations, where a service description is requested before the object providing such description is published. However, Lambic's service discovery mechanism also covers the case where the object is first published and then requested. Additionally, this mechanism also discovers local services. Finally, our implementation will notify the discovery of every object fulfilling the service description.

A.2.6 Conditions

Lambic uses CLOS' standard exception handling mechanism based on *conditions* [Sei05].[1] For the case of timeout exceptions, we represent them by the `timeout-exception` condition. We create these conditions under two different circumstances:

- When the channel that sends a remote invocation is connected (to the corresponding remote actor), we attach the timeout to the socket created for such a send. When the timeout is reached, we capture the exception raised by the socket and create a `timeout-exception` condition. This condition is asynchronously notified to the future of the remote invocation by invoking a `ruin-with-condition` generic function. This generic function is defined as part of the behaviour of the `future` class.
- If the channel is disconnected, we set a dedicated *timer* process which checks the timeout of the invocation. When it is reached, the timer asynchronously invokes

[1]The differences between CLOS' conditions and other languages' exceptions are not relevant for the implementation of our model.

the `ruin-with-condition` on the corresponding future.

A.3 Futurised Generic Functions

The implementation of futurised generic functions is an extension of the metaobject protocol of CLOS. This extension enables generic functions to implicitly handle futures received as arguments, and to return a future as result. We define this behaviour for the `futurised-function` and `futurised-method` classes and make all generic functions and methods to be instances of such classes. We do this implicitly, by redefining the `defgeneric` and `defmethod` macros. Finally, we also enable Common Lisp native functions and special forms to work with futures. In this section, we review each of these extensions.

A.3.1 Futurised Function Class

The `futurised-function` class is a subclass of `standard-generic-function`. It uses the `funcallable-standard-class` as its metaclass. This allows the instances of `futurised-function` to be invoked as functions (i.e. known as *funcallable instances*). The listing below shows the definition of this class.

```
; Definition of the class for futurised generic functions.
(defclass futurised-function (standard-generic-function) ()
  (:metaclass funcallable-standard-class))
```

Listing A.1: Definition of `futurised-function` class.

In the CLOS metaobject protocol, invocations to *base-level* generic functions are handled by the `compute-discriminating-function` *meta-level* generic function. It receives the invoked generic function as an argument and returns a closure which is applied to the arguments of the invocation. We then define a method for this function which is specialised on `futurised-function`.

Listing A.2 presents part of the implementation of futurised generic functions. In particular, the listing shows the `compute-discriminating-function` method and the `proceed` function. The `compute-discriminating-function` method performs the following tasks:

- It creates a future (`result-future`) which is returned as the result of the method.
- It computes and stores the discriminating function as defined in CLOS (invoking `call-next-method`).
- It collects all the arguments corresponding to unresolved futures.
- If there are no unresolved futures, the method continues the execution by calling the `proceed` function. If there are unresolved futures, the method resgisters an observer for their resolution. This is done by using the `when-all-resolved` form which works

in a similar way as the `when-resolved` form (cf. Section 4.4.2). Once the futures are resolved, the method calls the `proceed` function.

```
; Enable generic functions to handle futures.
(defmethod compute-discriminating-function ((function futurised-function))
  (lambda (&rest args)
    (let ((result-future (make-instance 'future))
          (discriminating-function (call-next-method))
          (unresolved-futures (find-unresolved-futures args)))
      (if unresolved-futures
        (proceed function args result-future discriminating-function)
        (when-all-resolved unresolved-futures
          (lambda (useless-result)
            (proceed function args result-future discriminating-function) )))
        result-future)))

; Apply discriminating function.
(defun proceed (function args result-future discriminating-function)
  (let* ((unfolded-args (unfold args))
         (designator (first unfolded-args)))
    (if (in-current-actor designator)
      (setf result  (apply discriminating-function unfolded-args) )
      (let* ((function-name (generic-function-name function))
             (function-call (make-function-call :name function-name
                                                :arguments unfolded-args)))
        (setf result (send-actor-message *current-actor* designator
                                         function-call
                                         :due-in *response-timeout*) )))
    (resolve-with-result result-future result)))
```

Listing A.2: Enabling generic functions to handle futures.

The `proceed` function contains the continuation of the execution of the `compute-discrimating-function` method. This function is synchronously executed only when the invocation of the generic function does not receive unresolved futures as arguments. Otherwise, it is executed asynchronously. We highlight the two cases in Listing A.2. The `proceed` function performs the following tasks:

- It replaces the futures of the arguments by their results (using the `unfold` function).
- It checks whether the first argument —i.e. the *implicit actor designator* (cf. Section 4.4.2)— is contained in the actor executing the invocation. If this is the case, the discriminating function is applied to the arguments. Otherwise the invocation is turned into an asynchronous invocation to the corresponding remote actor (using

the `send-actor-message` generic function explained in Section A.2.2). Listing A.2 also highlights these two cases.

- Finally, the result of either case is used to resolve the future of the invocation (`result-future`).

A.3.2 Futurised Method Class

The `futurised-method` class is a subclass of `standard-method`. It has an extra field that stores the execution qualifier of the method (cf. Section 4.4.2). The listing below shows the definition of this method.

```
; Definition of the class for futurised methods.
(defclass futurised-method (standard-method) ()
  ((execution-qualifier :initarg :execution-qualifier :initform nil
                        :accessor execution-qualifier)))
```

Listing A.3: Definition of class for futurised methods.

The execution qualifier is checked when the method is evaluated. If the method body returns an unresolved future and the qualifier has not been indicated, the execution throws a warning. Also, when the execution of the method is suspended (because the handling of an asynchronous generic function invocation), it is checked whether the qualifier is `:uninterruptible`. In such a case, the actor blocks its event loop until the resolution of the future returned by the asynchronous invocation. We include this support for execution qualifiers in the body of the `defmethod` macro.

A.3.3 Futurised Library

The futurised generic function model also requires to enable Common Lisp native functions and special forms to work with futures. Because these functions and forms are not handled by `compute-discriminating-function`, we have to manually add the support for futures. For native functions we provide a `futurise-functions` macro. It receives as arguments a list of symbols bound to functions. For each of those functions, a Lambic method is generated with an invocation of the function as body. With regards to special forms (and also some native macros), because they have special rules to evaluate their parameters, we have to manually redefine them as macros. As an example, Listing A.4 shows the redefinition of the `if` special operator as a macro.

```
; Futurised if form
(defmacro futurised-if (&rest args)
  `(let ((fut (make-instance 'future))
         (condition ,(car args)))
     (when-resolved condition
       (lambda (condition-result)
         (resolve-with-result fut (if condition-result ,@(cdr args)))))
     fut))
```

Listing A.4: Redefinition of `if` special operator.

A.4 Predicated Generic Functions

The implementation of predicated functions is an extension of the metaobject protocol of CLOS. This extension enables generic functions to specialise methods on context predicates. We define this behaviour for the `predicated-function` and `predicated-method` classes. As in the case of futurised generic functions, this extension also requires the redefinition of the `defgeneric` and `defmethod` macros.[2]

To enable context predicate dispatch, we model the predicates as implicit arguments of the generic function. Such a support enables Lambic to determine the applicability of methods according to the evaluation of the context predicates, and to order the methods according to the priority order of the predicates declared in the generic function definitions. We explain this extension in the following subsections.

A.4.1 Predicated Function Class

The `predicated-function` class is a subclass of `standard-generic-function` and uses the `funcallable-standard-class` as its metaclass. The `predicated-function` class has a `predicates` field which stores the list of predicates used by the generic function's methods (cf. Section 5.1.1). The listing below shows the definition of the `predicated-function` class.

```
; Definition of the class for predicated generic functions.
(defclass predicated-function (standard-generic-function)
  ((predicates :initarg :predicates :initform nil :accessor function-predicates))
  (:metaclass funcallable-standard-class))
```

Listing A.5: Definition of `predicated-function` class.

[2] In Section A.6, we review how these extensions are combined.

To enable context predicate dispatch, we extend the generic functions' parameter lists with a set of implicit *predicate parameters*. This is done in the redefinition of the `defgeneric` macro. In the current implementation, we allow methods to use up to ten predicates. Thus, by default we extend the parameter list of a generic function with ten predicate parameters. As an example, consider the automatic extension of the following generic function definition presented in the validation chapter (Section 7.2.3):

```
; Original definition
(defgeneric mouse-move (editor shape x y)
  (:predicates moving? drawing? drawing-selection?))

; Implicitly extended version
(defgeneric mouse-move (editor shape x y predpar-1 predpar-2 predpar-3
                        predpar-4 predpar-5 predpar-6 predpar-7 predpar-8
                        predpar-9 predpar-10)
  (:predicates moving? drawing? drawing-selection?))
```

Listing A.6: Implicit addition of predicate parameters.

Such predicate parameters are implicitly bound to values in a `compute-discriminating-function` method specialised on the `predicate-function` class. We defer the explanation of this method to Section A.4.3, when discussing the evaluation of context predicates.

A.4.2 Predicated Method Class

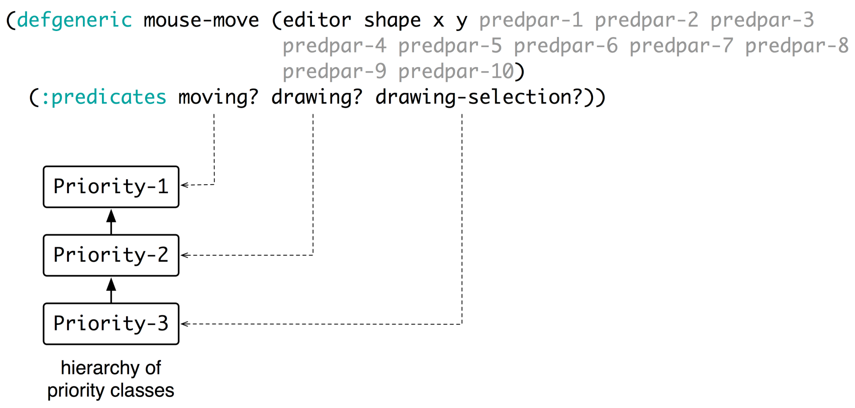
The `predicated-method` class is a subclass of `standard-method`. It has an extra field containing a predicate expression (cf. Section 5.1.1). The listing below shows the definition of the `predicated-method` class.

```
(defclass predicated-method (standard-method)
  ((predicates :initarg :predicates :initform nil :accessor method-predicates)))
```

Listing A.7: Definition of `predicated-method` class.

As in the case of predicated generic functions, we also extend the parameter list of predicated methods with ten extra predicated parameters. This is implicitly done in the redefinition of the `defmethod` macro. In this case such parameters are specialised on classes representing the priorities of the predicates (called *priority classes*). These classes determine the order of evaluation of the methods.

Priority classes are structured in an inheritance chain. The root class of this chain represents the least priority. Determining which classes specialise the predicate parameters of a method, depends on the specificity of the predicates used by the method. As



```
(defgeneric mouse-move (editor shape x y predpar-1 predpar-2 predpar-3
                          predpar-4 predpar-5 predpar-6 predpar-7 predpar-8
                          predpar-9 predpar-10)
  (:predicates moving? drawing? drawing-selection?))
```

Figure A.6: Mapping predicates to priority classes.

explained in Section 5.1.1, such a specificity is given by the order in which predicates are declared in the method's generic function. The more specific the predicates used by a method, the more priority has the method. Thus, associating priority classes with predicate parameters implies the following steps:

- The predicates used by a method are sorted from least to most specific.
- The sorted list of predicates is mapped to the predicate parameters of the method. This mapping is done from left to right. The least specific predicate corresponds to the first parameter.
- Each predicate parameter uses as a specialiser the priority class representing the specificity order of the predicate. For a predicate which appears in the first position in the predicate list of the generic function, the priority class is `priority-1`. For a predicate appearing in the second position, the priority class is `priority-2` (where `priority-1` is the parent class of `priority-2`), and so on.

Figure A.6 illustrates the mapping of priority classes to the predicates of the `mouse-move` generic function presented in Listing A.6. The `moving?`, `drawing?` and `drawing-selection?` predicates are associated with the `priority-1`, `priority-2` and `priority-3` classes respectively. As such the methods of this generic function are augmented with the list of specialised predicate parameters shown in Listing A.8. The `mouse-move` method using the `moving?` context predicate has the first predicate parameter specialised on the `priority-1` class. As such, it is the least specific method. The most specific method is the that specialised on `drawing-selection?` and whose first predicate parameter is specialised on `priority-3`.

```
; mouse-move when moving
(defmethod mouse-move ((editor geuze-editor) shape x y (predpar-1 priority-1)
                       predpar-2 predpar-3 predpar-4 predpar-5 predpar-6
                       predpar-7 predpar-8 predpar-9 predpar-10)
  (:when (moving? shape editor))
  ...)

; mouse-move when drawing
(defmethod mouse-move ((editor geuze-editor) shape x y (predpar-1 priority-2)
                       predpar-2 predpar-3 predpar-4 predpar-5 predpar-6
                       predpar-7 predpar-8 predpar-9 predpar-10)
  (:when (drawing? shape editor))
  ...)

; mouse-move when drawing selection
(defmethod mouse-move ((editor geuze-editor) shape x y (predpar-1 priority-3)
                       predpar-2 predpar-3 predpar-4 predpar-5 predpar-6
                       predpar-7 predpar-8 predpar-9 predpar-10)
  (:when (drawing-selection? shape editor))
  ...)
```

Listing A.8: Implicit addition of specialised predicate parameters.

A.4.3 Context-dependent Predicate Dispatch

Listing A.9 shows the definition of the `compute-discriminating-function` method specialised on the `predicated-function` class.

```
; Enable generic functions to dispatch on context predicates.
(defmethod compute-discriminating-function ((function predicated-function))
  (lambda (&rest args)
    (let* ((extended-args (add-predicate-arguments function args))
           (preselected-methods (compute-applicable-methods function
                                                            extended-args))
           (selected-methods (loop for method in preselected-methods
                                   if (eval-method-predicates method args)
                                   collect method)))
      (apply-methods selected-methods function extended-args))))
```

Listing A.9: Enabling context predicate method dispatch.

This method performs the following tasks:

- It extends the list of arguments received in the invocation with ten predicate arguments. This way they match the extended lists of parameters of the predicated generic function and methods. The predicate arguments correspond to objects of the priority class mapping the most specific context predicate. For instance, in the example of Listing A.8, the most specific predicate (`drawing-selection?`) is mapped to the `priority-3` class. Thus, an invocation to `mouse-move` can be implicitly converted as follows:

```
; Original invocation
(mouse-move editor-1 circle-2 30 40)

; Implicitly extended version
(mouse-move editor-1 circle-2 30 40 object-p3 object-p3 object-p3
          object-p3 object-p3 object-p3 object-p3 object-p3
          object-p3 object-p3)
```

  In the listing above, `object-p3` is an instance of the `priority-3` class. Note that this instance is used only for computing the right order of the methods according to their specialising context predicates. As such, we can use the same instance for all the predicate arguments of the invocation.

- Then, the applicable methods are computed using the standard CLOS semantics (method selection according to the types of the arguments).

- For each method resulting from the previous step, we evaluate its predicates. The final list of applicable methods includes only the methods whose predicates are satisfied.

- Finally, the selected methods are applied to the extended list of arguments.

A.5 Group Generic Functions

The implementation of group generic functions is an extension of the metaobject protocol of CLOS. This extension enables the definition of group behaviour for objects. To model such a group behaviour independent from the standard (peer) behaviour of objects, we use two generic functions. These generic functions are instances of the `group-function` and `peer-function` classes respectively. Group generic functions require the redefinition of the `defgeneric` macro. In addition, group generic functions introduce the `defgroupgeneric` and `defgroupmethod` macros. Group behaviour can de defined only for instances of a `group-class` class. A `defgroupclass` macro enables a new class to implicitly become a subclass of `group-class`.

A.5.1 Group Class

The `group-class` class is a subclass of `standard-class`. It has a `peers` field to store the instances of a same group class. In addition, we define an *after* `initialize-instance` method specialised on this class. This method registers the observers that allow `group-class` instances to discover each other and be aware of their changes of connectivity. Listing A.10 shows these two definitions.

```
; Definition of group-class class
(defclass group-class ()
  ((peers :initform nil :accessor peers))

; Enable objects of a group class to find each other.
(defmethod initialize-instance :after ((this-object group-class)
                                        &key &allow-other-keys)
  (let* ((classname (class-name (class-of this-object)))
         (description (format nil "~d" classname)))
    (push this-object (peers this-object))
    (export-service this-object description)
    (whenever-discovered description
      (lambda (peer-object)
        (if (new-peer? this-object peer-object)
          (push peer-object (peers this-object))
          (peer-discovered this-object peer-object)
          (when-disconnected peer-object
            (lambda (peer-object)
              (peer-disconnected this-object peer-object)))
          (when-reconnected peer-object
            (lambda (peer-object)
              (peer-reconnected this-object peer-object)))))))
```

Listing A.10: Definition of `group-class` class.

In the initialize-instance method, a new instance adds itself to the list stored in its `peers` field. Then the instance is published by means of the `export-service` form (highlighted in the listing above). To install the observers we use the `whenever-discovered`, `when-disconnected` and `when-reconnected` forms for this purpose (also highlighted in the listing). Discovered instances are added to the peer list only if they have not been already registered. Then, the corresponding generic functions, `peer-discovered`, `peer-disconnected` and `peer-reconnected` are invoked. Default methods are provided for these generic functions which are specialised on `group-class`. These methods contain in their body only an invocation to print a message in the console.

A.5.2 Group Function Class

The group-function class is a subclass of standard-generic-function and uses as metaclass the funcallable-standard-class class. Instances of this class contain the group methods of group generic functions. It has a peer-function field bound to the peer-level definition of the group generic functions (an instance of the peer-function class). The listing below shows the implementation of the group-function class.

```
; Definition of the class for group generic functions.
(defclass group-function (standard-generic-function)
  ((peer-function :accessor peer-function))
  (:metaclass funcallable-standard-class))
```

Listing A.11: Definition of group-function class.

Instances of group-function class are defined by the defgeneric macro. The name of these instances add group- a prefix to name given to the group generic function. For instance, the definition of a move-shape group generic function will implicitly creates an instance of group-function called group-move-shape. Then, all group methods of move-shape will also be renamed group-move-shape.

A.5.3 Peer Function Class

Listing A.12 shows the implementation of the peer-function class. Its instances contain the peer methods of group generic functions. It is also a subclass of standard-generic-function and uses the funcallable-standard-class as its metaclass. This class has a group-function field bound to the corresponding group-function instance.

```
; Definition of the class for peer generic functions.
(defclass peer-function (standard-generic-function)
  ((group-function :initarg :group-function :initform nil
                   :accessor group-function))
  (:metaclass funcallable-standard-class))
```

Listing A.12: Definition of peer-function class.

Instances of peer-function (and thus their containing peer methods) preserve the name given originally to the group generic functions.

A.5.4 Peer and Group Methods

In this implementation, we do not use any special metaclass for peer or group methods. Both kinds of methods correspond to instances of standard-method. The implementation

```
(DEFMETHOD GROUP-PAINT-SHAPE ((SHAPE GROUP-SHAPE) COLOR)
  (LET* ((LAMBIC::ALL-COMBINATIONS
         (LAMBIC::FIND-ARGUMENT-COMBINATIONS (LIST SHAPE COLOR)
                                             '(#<STANDARD-CLASS GROUP-SHAPE 21C23CAB>
                                               #<BUILT-IN-CLASS T 20950CAF>)))
         (*ORIGINAL-ARGUMENTS* (LIST SHAPE COLOR))
         (LAMBIC::SELECTED-COMBINATIONS (LAMBIC::APPLY-PROPAGATION-CONDITIONS ((SELECTED? SHAPE)) (SHAPE COLOR) LAMBIC::ALL-COMBINATIONS))
         (LAMBIC::CALL-NEXT-METHOD-FUTURE (MAKE-INSTANCE 'FUTURE)))
    (WHEN-ALL-RESOLVED (LIST (FUTURISED-LAMBDA (LAMBIC::ARGUMENT-COMBINATIONS)
                           (LET* ((LAMBIC::PARALLEL-FUTURE (MAKE-INSTANCE 'FUTURE))
                                  (LAMBIC::*STANDARD-TIMEOUT* 10)
                                  (LAMBIC::INTERMEDIATE-RESULTS
                                    (LOOP LAMBIC::FOR LAMBIC::COMBINATION
                                          LAMBIC::IN LAMBIC::ARGUMENT-COMBINATIONS
                                          LAMBIC::COLLECT (HANDLER-CASE
                                                            (COMMON-LISP:IF (LAMBIC::IN-CURRENT-ACTOR (CAR LAMBIC::COMBINATION))
                                                              (CALL-NEXT-METHOD (COMMON-LISP:NTH 0 LAMBIC::COMBINATION)
                                                                                (COMMON-LISP:NTH 1 LAMBIC::COMBINATION))
                                                              (COMMON-LISP:APPLY 'PAINT-SHAPE
                                                                                 (APPEND LAMBIC::COMBINATION
                                                                                         (APPEND (LIST :METHOD-SPECIALISERS)
                                                                                                 (LIST '(GROUP-SHAPE T))))))
                                                            (LAMBIC::TIMEOUT-CONDITION NIL NIL)))))
                             (WHEN-RESOLVED LAMBIC::INTERMEDIATE-RESULTS
                                            (COMMON-LISP:LAMBDA (LAMBIC::RESULTS)
                                              (LET ((*ORIGINAL-ARGUMENTS* (LIST SHAPE COLOR)))
                                                (RESOLVE-WITH-RESULT LAMBIC::PARALLEL-FUTURE
                                                                     (COMMON-LISP:FUNCALL #<LAMBIC::FUTURISED-FUNCTION DEFAULT-REDUCE-RESULTS-TO 21AAD35A>
                                                                                           LAMBIC::RESULTS)))))
                             LAMBIC::PARALLEL-FUTURE))
                                               LAMBIC::SELECTED-COMBINATIONS)
      (COMMON-LISP:LAMBDA (LAMBIC::RESULTS)
         (RESOLVE-WITH-RESULT LAMBIC::CALL-NEXT-METHOD-FUTURE (COMMON-LISP:FUNCALL (FIRST LAMBIC::RESULTS) (SECOND LAMBIC::RESULTS)))))
    LAMBIC::CALL-NEXT-METHOD-FUTURE))
```

Table A.1: Expansion of `defgroupmethod` macro.

of group methods is contained mostly in the `defgroupmethod` macro. In this macro, the propagation expression is converted into a function which is bound to the `call-next-method` form. This function includes the following tasks:

- To find the *most general propagation* for an invocation (cf. Section 6.2.4). It consists of all the possible argument lists resulting from combining the instances of the group classes used as parameters.
- To obtain the *effective propagation* for an invocation (cf. Section 6.2.4). This means applying the list of propagation predicates (received in the `:propagate` argument of the propagation expression) to the different argument lists obtained in the step above.
- To propagate the invocations. This implies invoking the next applicable method with each argument list included in the effective propagation. These invocations are performed sequentially.
- To control exceptions raised by each invocation using the handlers passed in the `:catch` argument of the propagation expression.
- To deal with the return values of the invocations using the result function passed in the `:return` argument of the propagation expression.

As example, Figure A.1 shows the result of expanding the `defgroupmethod` macro for the following definition:

```
(defgroupmethod paint-shape ((shape group-shape) color)
  (:propagate ((selected? shape))
   :return nil)
  (call-next-method))
```

The body of the resulting `group-paint-shape` method corresponds to the code replacing the invocation of `call-next-method` in the `paint-shape` group method.

Because the propagation can target different hosts, the invocations included in the propagation contain an extra argument denoted with the `:previous-method-specialisers` keyword. This argument indicates the list of specialisers of the method that has triggered the propagation. As we explain later in the next section, this information is required to ensure that the execution of the group generic function continues with the next most specific peer or group method (thus, avoiding re-applying the same group method at different hosts).

Finally, in Section 6.2.4 we also introduce the `call-peer-method` form. Using this form in a group method will skip any other applicable group method, and execute the peer methods of the group generic function. For this, we also use an additional `:next-peer-methodp` argument. We further discuss this argument in the next section.

A.5.5 Applying Peer and Group Generic Functions

Invocations of group generic functions are received by a `compute-discriminating-function` method specialised on the `peer-function` class, shown in Listing A.13. This method is also called during the execution of an invocation, from the last applicable group methods of the group generic function (via the `call-next-method` or `call-peer-method` forms). We refer to the first case as a standard invocation, and the second case as invoking the *peer-level continuation*. We distinguish these two cases by means of a `*peer-level-continuationp*` variable. By default, this variable is `nil`. A standard invocation is handled by forwarding it to the group level of the group generic function. This implies invoking the `group-function` instance associated with the group generic function. This invocation is received by a `compute-discriminating-function` method specialised on the `group-function` class.

```
; Handling group generic function invocations at the peer level.
(defmethod compute-discriminating-function ((function peer-function))
  (lambda (&rest args)
    (let ((group-function (group-function function)))
      (if (or *peer-level-continuationp* (not group-function))
        (apply (call-next-method) args)
        (apply group-function args)))))
```

Listing A.13: Peer-level discriminating function.

An invocation of the peer-level continuation of a group generic function is handled by applying the original discriminating function. It is obtained by invoking the `call-next-method` form in the method specialised on `peer-function`.

Listing A.14 shows the definition of the `compute-discriminating-function` method specialised on `group-function`. This method performs the following tasks:

- It extracts the internal arguments (denoted with the `:previous-method-specialisers` and `:next-peer-methodp` keywords) from the argument list of the invocation. The values of these arguments are stored in the `previous-specialisers` and `next-peer-methodp` variables respectively.

- If `next-peer-methodp` is true then the `proceed-with-peer-function` method is called. This method invokes the peer-level definition of the group generic function (by setting the `*peer-level-continuationp*` variable to true and applying the corresponding peer generic function). If `next-peer-methodp` is `nil`, the `previous-specialisers` variable is checked.

- If `previous-specialisers` is `nil`, the original group-level discriminating function is applied. Otherwise, the list of selected group methods is computed. The group methods which have been already processed are removed from the list (i.e. the methods whose specialisers are more specific than those passed in `previous-specialisers`).

The remaining selected methods are applied. In case that there are no remaining methods, the `proceed-with-peer-function` method is called.

```
; Handling group generic function invocations at the group level.
(defmethod compute-discriminating-function ((function group-function))
  (lambda (&rest args)
    (multiple-value-bind (original-args previous-specialisers next-peer-methodp)
        (extract-internal-arguments args)
      (if next-peer-methodp
        (proceed-with-peer-function function original-args)
        (if  (not previous-specialisers)
          (apply (call-next-method) original-args)
          (let* ((selected-methods (compute-applicable-methods function
                                                               original-args))
                 (remaining-methods (remove-applied-methods selected-methods
                                                            previous-specialisers)))
            (if remaining-methods
              (apply-methods remaining-methods function original-args)
              (proceed-with-peer-function function original-args))))))))

; Invoking peer-level continuation.
(defmethod proceed-with-peer-function (function args)
  (let ((peer-function (peer-function function))
        (*peer-level-continuationp* t))
   (apply peer-function args)))
```

Listing A.14: Group-level discriminating function.

A.6 Integration

We now review Lambic's support for composing futurised, predicate and group generic functions. This support comprises two tasks: composing the behaviour provided by each feature, and combining the features' syntactic abstractions. In this section, we focus on the composition of behaviour. The main challenge of this task is to keep the modularity of the three features. In this section, we discuss the issues we encounter to ensure such a modularity.

A.6.1 Composition of Behaviour

Lambic composes the behaviour of the features' generic function classes. This behaviour is defined in the `compute-discriminating-function` methods. We organise the generic function classes in the hierarchy presented in Figure A.7. In this hierarchy, `predicate-function` is the parent class of the `peer-function` and `group-function` classes, and `peer-function` is the parent class of the `futurised-function` class. Therefore, we achieve the following sequence of tasks:

- Generic function invocations are handled by the `compute-discriminating-function` method specialised on the `futurised-function` class. As such, the futures passed as arguments of the invocations are handled as a first step. This ensures that no futures are passed to the other `compute-discriminating-function` methods (invoked using the `call-next-method` form).
- The next method applied is the one specialised on the `peer-function` class. This method forwards the invocations to the `compute-discriminating-function` method specialised on `group-function` class. This method executes the group methods of the generic functions. After this execution, the control is given back to `compute-discriminating-function` method specialised on `peer-function`.
- Finally, the method specialised on `predicate-function` is processed.

Although the above precedence order between the methods works properly for most cases, the three features require further interactions with each other. These extra interactions are the following:

- The `compute-discriminating-function` method specialised on `predicate-function` has to deal with futures resulting from the evaluation of context predicates. We cope with this case by using Lambic's explicit future handlers in that method (the `when-resolve` form, cf. Section A.2.2). The same support is added to the `compute-discriminating-function` method defined for the `group-function` class, which has to deal with futures resulting from the evaluation of the propagation predicates.
- In Listing A.14, we show that the `compute-discriminating-function` method specialised on the `group-function` class invokes the `compute-applicable-methods` function of CLOS' metaobject protocol. However, the possibility exists that the group methods have context predicates. In such a case, the applicability of the methods should be evaluated according to those predicates. Thus, we have to create a new definition for `compute-applicable-methods` which includes such a predicate-based method dispatch. However, in the implementation of Common Lisp we use in our work (LispWorks), it is not possible to define methods for such a generic function (`compute-applicable-methods` can be invoked but not redefined). This forces us to create a `compute-applicable-predicated-methods` which is directly invoked within the `compute-discriminating-function` method specialised on the `group-function` class.
- Finally, the invocation of `compute-applicable-predicated-methods` can lead to a future (resulting from the evaluation of the context predicates). As such, they also have to be handled explicitly by means of `when-resolve` forms.

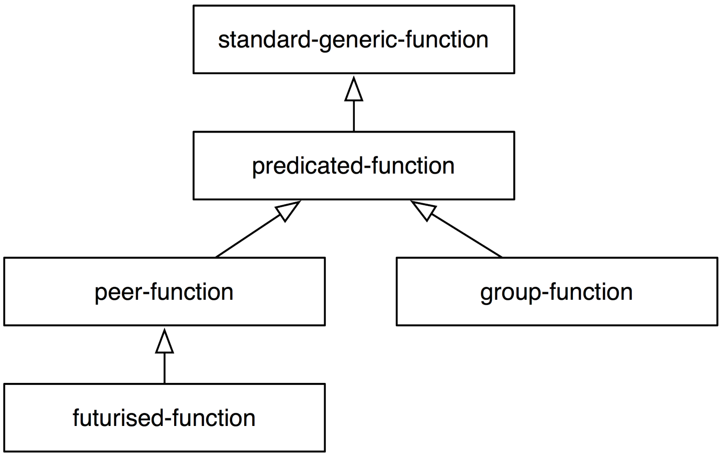


Figure A.7: Hierarchy of generic function metaclasses in Lambic.

A.7 A Push-based Cache Algorithm

An important issue with Lambic is that the intensive use of context and propagation predicates can make the evaluation less efficient and more prone to network failures. To tackle this issue we include in our current implementation a preliminary cache algorithm. We use this algorithm for the group behaviour of group classes. Such behaviour typically requires recurrent accesses to the fields of the peer objects. We then enable the peers to cache the values of each other's fields. We do this in a *push-based* manner: the cache of a field is updated whenever the field changes its value. The reason for this is that fields are accessed more frequently than that they change their values. We then let each `group-class` instance inform its changes of state to all its peers.

A.7.1 Example of Use

Figure A.8 illustrates such a push-based caching mechanism in the context of the scenario of the collaborative drawing editor (cf. Section 7.2). In this figure, there is an actor containing an editor object and another containing the cache for the editor's `zoom-level` field. Accessing this field for the first time results in an asynchronous remote invocation of the `zoom-level` reader function. This invocation returns an unresolved future as immediate return value (Case #1 in the figure). When the future receives the value of the invocation, it is cached for new requests (Case #2). Then, if the `zoom-level` field changes its value it is notified to the cache's actor. As such, further accesses to the field get the updated value (Case #3).

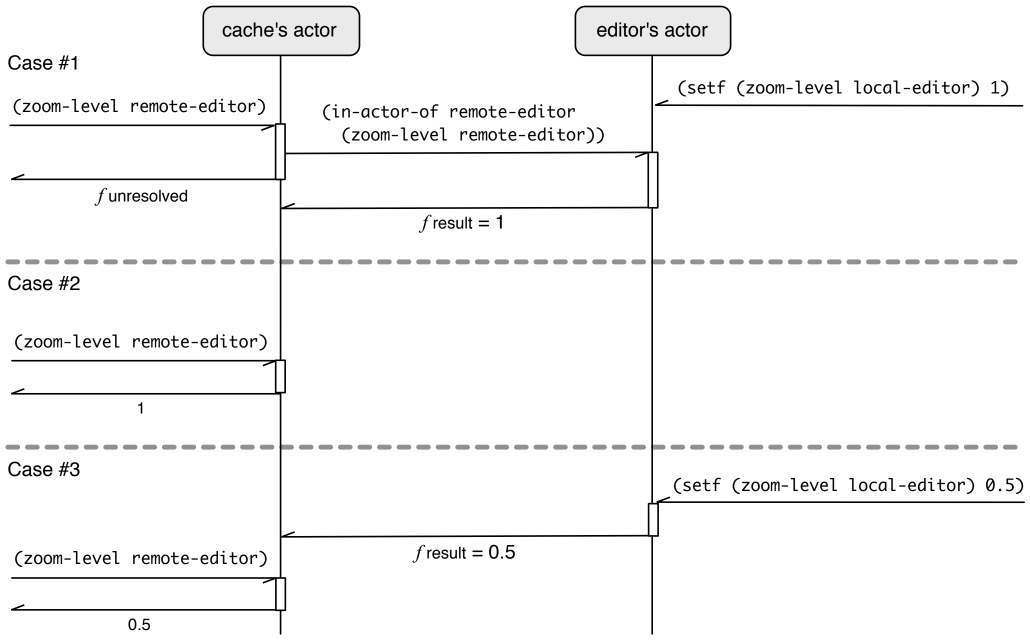

cache's actor
editor's actor
Case #1
(zoom-level remote-editor)
(in-actor-of remote-editor
(zoom-level remote-editor))
(setf (zoom-level local-editor) 1)
f unresolved
f result = 1
Case #2
(zoom-level remote-editor)
1
Case #3
(setf (zoom-level local-editor) 0.5)
f result = 0.5
(zoom-level remote-editor)
0.5

Figure A.8: Push-based cache for group class fields.

A.7.2 Implementation of Push-based Cache Algorithm

Before showing the implementation of the cache algorithm we briefly discuss the implicit definition of accessor functions for class fields. As in CLOS, when a field is defined using the `:accessor` argument, our model implicitly creates two methods for reading and writing the value from/to the field. Figure A.15 shows these automatically generated methods for the `zoom-level` field of the `geuze-editor` class. The reader method uses the value received in the `:accessor` argument as a name (e.g. `zoom-level`). The writer method's name is a two-element list composed of the `setf` symbol and the value of the `:accessor` argument (e.g. `(setf zoom-level)`). The two methods use the `slot-value` low-level function to access and modify the field respectively.

```
; Group class definition.
(defgroupclass geuze-editor ()
  ((zoom-level :accessor zoom-level)
   ...))

; Automatically generated reader method.
(defmethod zoom-level ((editor geuze-editor))
  (slot-value editor 'zoom-level))
```

```
; Automatically generated writer method.
(defmethod (setf zoom-level) ((editor geuze-editor) value)
  (setf (slot-value editor 'zoom-level) value))
```

Listing A.15: Automatically generated accessor functions.

Implementing push-based caching of group class fields involves the following tasks:

- Creating a list of cached values. We include this list in a `*cached-items*` variable. Each item consists of three values: the peer object whose field is accessed, the name of the reader function of that field, and the future that will cache the value of the field.

- Defining an additional method for the reader function, which is specialised on the `remote-reference` class. This implies that an invocation to a remote reader function will now be locally handled by this method definition. This method first checks whether there is a cached item containing the result of the invocation (i.e. for the peer object of the group class passed as argument). If this is not the case, the remote reader function is invoked and the resulting future is stored in a new cached item.

- When a cached item is generated, it is stored not only in the actor invoking the remote reader function. It is also communicated and stored in the actor containing the requested peer object. This way this actor can notify the changes of value in that object's field. This is done in a new *after* method for the field's writer function. This method iterates over the list of cached items searching for those matching the peer object and the field's reader function. Then the method invokes the `resolve-with-result` function on the future of each matched cached item.

Listing A.16 shows the definition of the new reader method specialised on remote references, the *after* writer method, and the auxiliary functions these methods require.

```
; Current actor's cached items.
(defvar *cached-items* nil)

; Reader method specialised on a remote reference.
(defmethod zoom-level  ((remote-editor remote-reference))
  (let ((future (get-cache-future remote-editor 'zoom-level )))
    (if (is-resolved future)
      (future-result future)
      (in-actor-of remote-editor (zoom-level  remote-editor)
                   :with-future future))))
```

```
; After writer method.
(defmethod (setf zoom-level ) :after ((local-editor geuze-editor) value)
  (foreach (cached-item *cached-items*)
    (if (match cached-item local-editor' zoom-level )
      (let ((future (third cached-item)))
        (in-actor-of future (resolve-with-result future value))))))

; Get cache future.
(defun get-cache-future (remote-object function-name)
  (let* ((cached-item (find-if #'(lambda (cached-item)
                                   (match cached-item remote-object
                                          function-name))
                              *cached-items*))
         (future (third cached-item)))
    (if (not future)
      (begin
        (setf future (make-instance 'future))
        (add-cache-future remote-object function-name new-future)
        (in-actor-of remote-object
          (add-cache-future remote-object function-name new-future))))
   future))

; Add a new cache future.
(defun add-cache-future (local-object function-name future)
  (push (list remote-object function-name future) *cached-items*))

; Match cache item.
(defun match (cached-item object function-name)
  (and (equal (first cached-item) remote-object)
       (equal (second cached-item) function-name)))
```

Listing A.16: Implementing push-based caches.

In the listing above, we highlight the references to specific function names in the reader and writer methods. In our implementation, we create a macro that receives the name and creates the two methods implicitly. Yet, this only works for accessor functions. A more general solution is part of our future work.

Appendix B

Lambic Syntax and Libraries

This chapter summarises the syntax of Lambic. Additionally, we present a library of functions we develop to implement the scenarios presented in the validation chapter of this thesis.

B.1 Syntax

To present the syntax of Lambic, we use an adaptation of the EBNF notation. Due to the central role of parentheses in Lambic's syntax, we have substituted this symbol in the EBNF specification (used for grouping terms), with angle brackets. The expressions after the arrows indicate the return values.

B.1.1 Classes

```
Class definition ::=
(defclass class-name (parent-class)({field}))                    → new-class

field ::= (field-name field-option)
field-option ::= [:initarg initarg-name]
                 [:initform form]
                 [:reader reader-function-name] ...

Group class definition ::=
(defgroupclass class-name (parent-class)({field}))               → new-class

Class instantiation ::=
(make-instance class-name [init-values])                         → new-object
```

B.1.2 Generic Functions

Generic function definition ::=
(`defgeneric` *function-name parameter-list*
  [(`:predicates` {*pred-name*})]) → *new-generic*

parameter-list ::= ({*parameter*})

Group generic function definition ::=
(`defgroupgeneric` *function-name parameter-list*
  [(`:group-predicates` {*pred-name*})]
  [(`:predicates` {*pred-name*})]) → *new-groupgeneric*

parameter-list ::= ({*parameter*})

Generic function invocation ::=
(*function-name* {*argument*}) → *result*

B.1.3 Methods

(Peer) method definition ::=
(`defmethod` *method-name* [*execution-qualifier*] [*combination-qualifier*]
  specialised-parameter-list
  [*predicate-expression*]
  body) → *new-method*

Group method definition ::=
(`defgroupmethod` *method-name* [*execution-qualifier*] [*combination-qualifier*]
  specialised-parameter-list
  [*predicate-expression*]
  propagation-expression
  body) → *new-groupmethod*

execution-qualifier ::= `:interruptible` | `:uninterruptible`
combination-qualifier ::= `:before` | `:around` | `:after`
specialised-parameter-list ::= ({*parameter* | (*parameter specialiser*)})
predicate-expression ::= (`:when` {(*pred-name arguments*)})

Continued on next page

Continued from previous page

propagation-expression ::= (`:propagate` ({(*pred-name arguments*)})
    [*in-actor-of*] [*due-in*] [*catch*] [*return*])

in-actor-of ::= `:in-actor-of` *actor-designator*
due-in ::= `:due-in` ⟨`nil` | *seconds*⟩
catch ::= `:catch` {(*exception-name* ([*argument*])*body*)}
return ::= `:return` ⟨`nil` | *user-defined-function*⟩

Next method call ::= (`call-next-method`) → *result*

Next peer method call ::= (`call-peer-method`) → *result*

B.1.4 Explicit Syntax for Event-driven Distribution

Export service ::=
(`export-service` *object service-description*) → *nil*

Import service ::=
(`import-service` *service-description*) → *unbound-reference*

Discovery event handler ::=
(`whenever-discovered` *service-description lambda*) → *nil*

Actor definition ::=
(`spawn-actor` *name* {*object*}) → *actor*

Remote generic function invocation ::=
(`in-actor-of` *actor-designator function-invocation*
    [`:with-future` ⟨*future* | `nil`⟩]
    [`:due-in` *seconds*]) → *future* | *nil*

Remote result handling ::=
(`when-resolved` *future function*
    [`:catch` *exception-handler*]) → *nil*

Continued on next page

Continued from previous page

Discovery event handler ::=
(`whenever-disconnected` *remote-reference lambda*) → *nil*

Discovery event handler ::=
(`whenever-reconnected` *remote-reference lambda*) → *nil*

B.2 Lambic Library

This library comprises a number of additional generic functions and macros included in the current implementation of the Lambic programming language model.

B.2.1 Library for Basic Functionality

(`try-catch` *form* {*exception-handler*})
Macro for handling exceptions (similar to Common Lisp `handler-case` form).

(`begin` *forms*)
Macro for evaluating forms in the order in which they are given (similar to Common Lisp `progn` form).

(`foreach` (*var list*) *body*)
Macro for iterating over the elements of a list (similar to Common Lisp `dolist` form).

(`display` *forms*)
Generic function to print representations of objects in the standard output.

(`sethash` *hash-table key value*)
Generic function to set key-value pair to a hash table.

B.2.2 Library Supporting Furturised Generic Functions

(`make-class-tag` *class-name*)
Generic function to create a text-based description of a class.

(`futurise-functions` *function-names*)
Generic function that enables existing Common Lisp functions to implicitly handle futures.

Continued on next page

Continued from previous page

`*response-timeout*`
Variable storing the default timeout for remote generic function invocations.

B.2.3 Library Supporting Group Generic Functions

(`peer-discovered` *local-peer discovered-peer*)
Generic function to handle peer discovery event.

(`peer-disconnected` *local-peer disconnected-peer*),
(`peer-reconnected` *local-peer reconnected-peer*)
Generic functions to handle peer connectivity events.

(`each` *object*), (`original` *object*), (`co-located?` *object-1 object-2*)
Generic functions representing propagation predicates for group methods.

(`first` *results*), (`all` *results*)
Generic functions representing multi-result handlers (returned by the propagation of a group method).

`no-applicable-propagation`
Error thrown when the evaluation of the propagation expression of a group method results in an empty list.

Appendix C

Lambic Kriek

Kriek is implemented using the `lambic-user` package. This package integrates the functionalities of the `cl-user`, `capi`, `color` and `gp` LispWorks packages. We decompose the Kriek program in four parts: classes, operations (generic functions), discovery and connectivity events, and user interface. Figure C.1 shows the graphic user interface of Kriek.

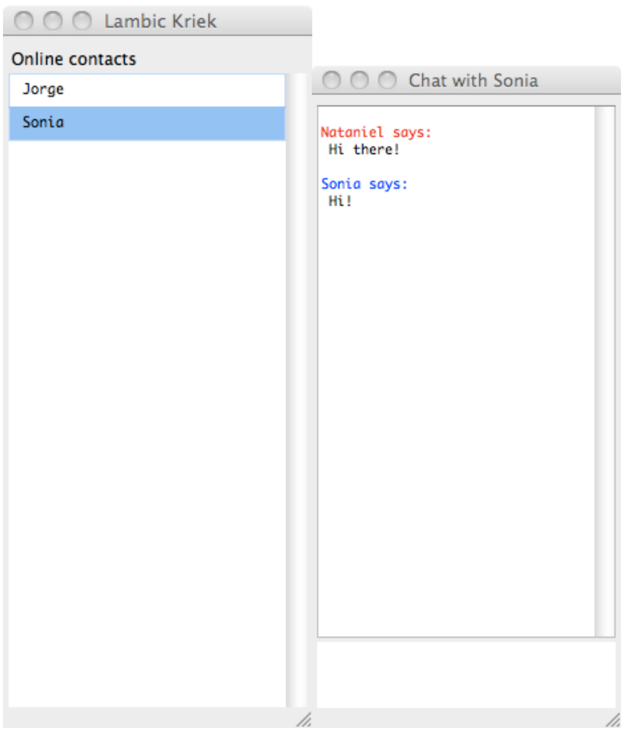
C.1 Kriek Group Class and Structs

```lisp
; Definition of Kriek chat service as a group class
(defgroupclass kriek-chat ()
  ((username :initarg :username
             :accessor chat-username)
   (address-book :initform (make-hash-table :test 'equal)
                 :accessor chat-address-book)
   (history :initform '()
            :accessor chat-history)
   (gui :initarg :gui :accessor chat-gui)
   (role :initform nil :accessor chat-role)))

; Definition of contact struct
(defstruct contact username chat)

; Definition of text message struct
(defstruct text-message sender receiver text)
```

Listing C.1: Kriek class and structs.



```
Nataniel says:
 Hi there!

Sonia says:
 Hi!
```

(a) Kriek's main GUI. (b) Kriek's chat GUI.

Figure C.1: Kriek's GUI

C.2 Kriek Operations

C.2.1 Sending and Receiving Text Messages

```
; Send text to contact
(defmethod send-text ((chat kriek-chat) remote-chat text)
  (let ((message (make-text-message :sender chat
                                    :receiver remote-chat
                                    :text text)))
    (put-text-message chat message)))

; Put text message
(defgroupgeneric put-text-message chat text-message)

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Put text message in contact's and sender's chat
(defgroupmethod put-text-message ((chat kriek-chat) text-message)
  (:propagate ((in-text-message? chat text-message)) :due-in nil
   :return nil)
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Put text message in chat. Display and store it.
(defmethod put-text-message ((chat kriek-chat) text-message)
  (display-text-message chat text-message)
  (store-text-message chat text-message))
```

Listing C.2: Sending and receiving text messages.

C.2.2 Displaying Text Messages

```
; Display text message
(defgroupgeneric display-text-message (chat text-message)
  (:predicates text-chat-peer?))
```

```
; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Propagate to chats of same user
(defgroupmethod display-text-message ((chat kriek-chat) text-message)
  (:propagate ((same-user? chat (original chat)))
   :return nil)
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; When chat plays the text-chat-peer role
(defmethod display-text-message ((chat kriek-chat) text-message)
  (:when (text-chat-peer? chat))
  (display-text-message-in-window chat text-message)
  (call-next-method))

; Default behaviour for displaying text message
(defmethod display-text-message ((chat kriek-chat) text-message)
  (notify-text-message chat text-message))

; Notify text message using Growl
(defmethod notify-text-message ((chat kriek-chat) text-message)
  (growl-text-message text-message))
```

Listing C.3: Displaying text messages.

C.2.3 Storing and Retrieving Text Messages

```
; Store text message
(defgroupgeneric store-text-message (chat text-message))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Propagate to chat playing storage-peer role
(defgroupmethod store-text-message ((chat kriek-chat) text-message)
  (:propagate ((storage-peer? chat)) :due-in nil
   :return nil)
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Record text message in chat's history
(defmethod store-text-message ((chat kriek-chat) text-message)
  (push text-message (chat-history chat)))
```

```
; Get messages from history
(defmethod get-history ((chat kriek-chat) contact)
  (loop for text-message in (chat-history chat)
        when (or (equal (text-message-receiver message) (contact-username contact))
                 (equal (text-message-sender message) (contact-username contact)))
        collect text-message))
```

Listing C.4: Storing and retrieving text messages in/from history.

C.2.4 Predicates for Kriek Operations

```
; Check if chat is the text message's sender or receiver
(defmethod in-text-message? ((chat kriek-chat) text-message)
  (or (equal chat (text-message-sender text-message))
      (equal chat (text-message-receiver text-message))))

; Check if chat is the storage-peer
(defmethod storage-peer? :interruptible (chat)
  (chat-role-equal chat "storage-peer"))

; Check if chat is the text-chat-peer
(defmethod text-chat-peer? :interruptible (chat)
  (chat-role-equal chat "text-chat-peer"))

; Check if chat's role is the role passed as argument
(defmethod chat-role-equal :interruptible (chat role)
  (let ((chat-role (try-catch (chat-role chat)
                   (timeout-exception () nil))))
    (equal chat-role role)))

; Check if chat's username equals the username of the chat
; passed originally as argument to the group generic function
(defmethod same-user? ((chat kriek-chat) original-chat)
  (let* ((address-book (chat-address-book original-chat))
         (contact (get-contact address-book chat))
         (original-username (chat-username original-chat)))
    (equal (contact-username contact) original-username)))
```

```
; Get contact corresponding to chat
(defmethod get-contact (address-book chat)
  (with-hash-table-iterator (next-contact address-book)
    (loop (multiple-value-bind (more name contact) (next-contact)
            (unless more (return nil))
            (when (eql (contact-chat contact) chat)
              (return contact))))))
```

Listing C.5: Context and propagation predicates for Kriek operations.

C.3 Handling Discovery and Connectivity Events

```
; Handling discovery of contacts
(defmethod peer-discovered :interruptible ((chat kriek-chat) remote-chat)
  (let ((remote-username (try-catch (chat-username remote-chat)
                            (timeout-exception () nil))))
    (if remote-username
      (begin
        (add-contact chat remote-username remote-chat)
        (display-contact chat remote-chat)))))

; Handling disconnection of contacts
(defmethod peer-disconnected ((chat kriek-chat) remote-chat))
  (hide-contact chat remote-chat))

; Handling reconnection of contacts
(defmethod peer-reconnected ((chat kriek-chat) remote-chat))
  (display-contact chat remote-chat))

; Add contact to local chat's address book
(defmethod add-contact ((chat kriek-chat) remote-username remote-chat)
  (let ((new-contact (make-contact :username remote-username
                                   :chat remote-chat)))
    (sethash contact-username (chat-address-book chat) new-contact)))
```

Listing C.6: Handling discovery and connectivity of contacts.

C.4 Graphical User Interface

C.4.1 GUI Classes

```
; Main pane of Kriek chat. Show list of contacts.
(defclass kriek-contacts-pane (pinboard-layout)
  ((draggingp :initform nil :accessor pane-draggingp)
   (kriek-chat :accessor pane-kriek-chat)))

; Box displaying contact’s name
(defclass kriek-contact-box (pinboard-object)
  ((contact :initarg :contact :accessor box-contact)
   (window :initform nil :accessor box-window)
   (selectedp :initform nil :accessor box-selectedp)))
```

Listing C.7: GUI classes.

C.4.2 GUI Operations

```
; Complete initialisation of kriek-contact-window object
(defmethod initialize-instance :after ((contact-window kriek-contact-window))
  (let ((title (format nil “Chat with  d”
                       (contact-username (window-contact contact-window)))))
    (setf (interface-title contact-window) title)
    (load-history contact-window)))

; Display contact in chat’s contacts pane
(defmethod display-contact ((chat kriek-chat) remote-chat)
  (let* ((pane (chat-gui chat))
         (contact (get-contact chat remote-chat))
         (contact-box (make-instance ’kriek-contact-box :contact contact
                                     :width 200 :height 25))
         (contact-windows (collect-interfaces ’kriek-contact-window))
         (window (find-if (lambda (window)
                            (equal (contact-username contact)
                                   (contact-username (window-contact window))))
                          contact-windows)))
    (manipulate-pinboard pane contact-box :add-top)
    (reorder-pane pane)
    (when window
      (setf (box-window contact-box) window)
      (when (interface-visible-p window)
        (notify-contact-status window “online”)))))
```

```
; Conceal contact from chat's contacts pane
(defmethod hide-contact ((chat kriek-chat) remote-chat)
  (let* ((pane (chat-gui chat))
         (contact (get-contact chat remote-chat))
         (contact-box (get-contact-box pane contact)))
    (when contact-box
      (setf (layout-description pane)
            (remove contact-box (layout-description pane)))
      (reorder-pane pane)
      (let ((window (box-window contact-box)))
        (when (and window (interface-visible-p window))
          (notify-contact-status window "offline"))))))

; Display text in contact window
(defmethod display-text-message-in-window ((chat kriek-chat) text-message)
  (let* ((pane (chat-gui chat))
         (remote-chat (find-remote-chat text-message))
         (contact (get-contact chat remote-chat))
         (contact-username (text-message-sender text-message))
         (contact-windows (collect-interfaces 'kriek-contact-window))
         (window (find-if (lambda (window)
                            (equal contact-username
                                   (contact-username (window-contact window))))
                          contact-windows))
         (contact-box (get-contact-box pane contact)))
    (unless window
      (setf window (capi:display (make-instance 'kriek-contact-window
                                                :contact contact
                                                :kriek-chat (pane-kriek-chat pane))))
      (setf (box-window contact-box) window))
      (unless (interface-visible-p window)
        (capi:display window)
        (load-history window))
    (set-in-window window text-message)))

; Return the remote chat used as receiver or sender of the text message
(defmethod find-remote-chat (text-message)
  (if (in-current-actor (text-message-sender text-message))
    (text-message-receiver text-message)
    (text-message-sender text-message)))

; Load history in contact's window
(defmethod load-history (contact-window)
  (let* ((chat (kriek-chat contact-window))
         (contact (contact contact-window))
         (text-messages (reverse (get-history chat contact))))
    (foreach (text-message text-messages)
      (set-in-window contact-window text-message))))
```

```
; Load text message in history window
(defmethod set-in-window (contact-window text-message)
  (let* ((sender (text-message-sender text-message))
         (header-text (format nil “~%~d says:” sender))
         (text (format nil “~%~d” (text-message-text text-message)))
         (format-face *contact-face*)
         (history-pane (history-pane contact-window)))
    (when (equal sender (contact-username (contact contact-window)))
      (setf format-face *own-face*))
    (add-text-to-history-pane history-pane header-text format-face)
    (add-text-to-history-pane history-pane text)))

; Add text to history pane
(defun add-text-to-history-pane (history-pane text
                                 &optional (face *text-face* face-supplied-p))
  (let* ((buffer (editor-pane-buffer history-pane))
         (end (editor:buffers-end buffer)))
    (editor:insert-string end text)
    (editor:with-point ((start end))
        (editor:character-offset start (- (length text)))
        (editor:put-text-property start end ’editor:face face))))

; Notify contact’s change of status
(defmethod notify-contact-status (contact-window status)
  (let* ((name (contact-name (contact contact-window)))
         (status-text (format nil “~%~d is ~%~d%” name status)))
    (add-text-to-history-pane (history-pane contact-window)
                              status-text *contact-face*)))

; Draw contact box in pane
(defmethod draw-pinboard-object ((pane kriek-contacts-pane)
                                 (contact-box kriek-contact-box)
                                 &key &allow-other-keys)
  (with-geometry contact-box
    (let* ((color :white))
      (if (box-selectedp contact-box)
        (setf color :selection-color))
      (draw-rectangle pane %x% %y% %width% %height%
                      :foreground :selection-color :filled t)
      (draw-rectangle pane (1+ %x%) (1+ %y%) (- %width% 2) (- %height% 2)
                      :foreground color :filled t)
      (draw-string pane (contact-username (box-contact contact-box))
                   (+ %x% 10) (+ %y% 15) :foreground :black))))
```

```
; Get contact box from contact
(defmethod get-contact-box ((pane kriek-contacts-pane) contact)
  (find-if (lambda (contact-box)
             (eql (box-contact contact-box) contact))
           (layout-description pane)))

; Select contact box
(defmethod select-contact-box ((contact-box kriek-contact-box))
  (let ((pane (pinboard-object-pinboard contact-box)))
    (deselect-contact-boxes pane)
    (setf (box-selectedp contact-box) t)
    (invalidate-rectangle pane)))

; Deselect contact boxes
(defmethod deselect-contact-boxes ((pane kriek-contacts-pane))
  (let ((contact-boxes (layout-description pane)))
    (foreach (contact-box contact-boxes)
      (deselect-contact-box contact-box))))

; Deselect contact box
(defmethod deselect-contact-box ((box kriek-contact-box))
  (setf (box-selectedp box) nil)
  (invalidate-rectangle (pinboard-object-pinboard box)))

; Reorder contacts pane
(defmethod reorder-pane ((pane kriek-contacts-pane))
  (let* ((contact-boxes (layout-description pane))
         (ordered-contact-boxes (sort-contacts contact-boxes))
         (x 0)
         (y 0))
    (foreach (contact-box ordered-contact-boxes)
      (setf (pinboard-pane-position contact-box)
            (values x y))
      (if (evenp (position contact-box ordered-contact-boxes))
        (setf (getf (pinboard-object-graphics-args contact-box) :foreground)
              :selection-color))
      (setf y (+ y 24)))
    (invalidate-rectangle pane)))

; Rearrange contacts (based on alphabetic order)
(defun sort-contacts (contact-boxes)
  (sort contact-boxes (lambda (b1 b2)
                        (string< (contact-username (box-contact b1))
                                 (contact-username (box-contact b2))))))
```

```
; Basic support for Growl notifications
(defmethod growl-text-message (text-message)
  (let* ((sender (text-message-sender text-message))
         (text (text-message-text text-message))
         (title (format nil “Message from ~a:” sender)))
    (growl “Message Received” title text “Lambic Kriek”)))
```

Listing C.8: GUI operations.

C.4.3 User Interface of Contact Window

```
; Contact window. It contains chat transcripts.
(define-interface kriek-contact-window ()
  ((contact :initarg :contact :accessor window-contact)
   (kriek-chat :initarg :kriek-chat :accessor kriek-chat))
  (:panes
   (history-pane
    editor-pane
    :enabled nil :accessor history-pane :flag :output
    :visible-min-width 200 :visible-min-height 400)
   (editor-pane
    editor-pane
    :accessor editor-pane :flag :output
    :after-input-callback 'text-composed
    :vertical-scroll nil :visible-min-width 200
    :visible-min-height 50)))
```

Listing C.9: Interface of contact window.

C.4.4 Contact Window’s Event handler

```
; Process text typed in contact window
(defun text-composed (editor-pane command)
  (when (and (system:gesture-spec-p command)
             (eql (system:gesture-spec-to-character command)
                  #\return))
    (let* ((contact-window (top-level-interface editor-pane))
           (contact (contact contact-window))
           (remote-chat (contact-chat contact))
           (local-chat (kriek-chat contact-window))
```

```
         (text-message (make-text-message :sender local-chat
                                          :receiver remote-chat
                                          :text (editor-pane-text
                                                 editor-pane))))
    (send-text local-chat remote-chat text-message)
    (set-in-window contact-window text-message)
    (apply-in-pane-process editor-pane
                           #'(setf editor-pane-text)
                           “” editor-pane))))
```

Listing C.10: Contact window’s event handler.

C.4.5 Main Window

```
; Main interface
(define-interface kriek-chat-interface ()
  ()
  (:panes
   (contacts-pane
    kriek-contacts-pane
    :accessor contacts-pane
    :description nil
    :input-model ‘((((:button-1 :press) mouse-clicked)
                   ((:button-1 :second-press) mouse-doubly-clicked))
    :title “Online contacts”
    :background :white :vertical-scroll t
    :scroll-width 150 :scroll-height 200))
  (:layouts
   (main-layout
    row-layout
    '(contacts-pane)))
  (:default-initargs
   :title “Lambic Kriek”
   :max-width 215
   :best-height 500))
```

Listing C.11: Contact window’s event handler.

C.4.6 Main Window’s Event Handlers

```
; Handle mouse-clicked event.
(defun mouse-clicked (pane x y)
  (let ((contact-box (pinboard-object-at-position pane x y)))
    (if contact-box
        (select-contact-box contact-box)
      (unselect-contact-boxes pane))))

; Handle mouse-doubly-clicked event.
(defun mouse-doubly-clicked (pane x y)
  (let ((contact-box (pinboard-object-at-position pane x y)))
    (when contact-box
      (let ((window (window contact-box)))
        (unless window
          (setf window
                (capi:display (make-instance 'kriek-chat-contact-window
                                             :contact (box-contact contact-box)
                                             :kriek-chat (pane-kriek-chat pane))))
          (setf (box-window contact-box) window))
        (unless (interface-visible-p window)
          (capi:display window)
          (load-history window))))))
```

Listing C.12: Main window’s event handler.

C.4.7 Auxiliary Parameters

```
; Definition of selection color
(define-color-alias :selection-color
                    (make-rgb 0.15s0 0.51s0 0.93s0 0.3s0))

; Set graphic style for typed text
(defparameter *text-face* (editor:make-face :black :foreground :black :bold-p t
                                            :italic-p t :if-exists t))

; Set graphic style for text from contact
(defparameter *contact-face* (editor:make-face :red :foreground :red :bold-p t
                                               :italic-p t :if-exists t))
```

```
; Set graphic style for text from local chat
(defparameter *own-face* (editor:make-face :blue :foreground :blue :bold-p t
                                           :italic-p t :if-exists t))
```

Listing C.13: Auxiliary parameters.

C.4.8 Main Function

```
; Kriek chat's main function
(defun make-lambic-kriek (username)
  (let* ((kriek-chat (make-instance 'kriek-chat :username username))
         (kriek-chat-interface (make-instance 'kriek-chat-interface))
         (contacts-pane (contacts-pane kriek-chat-interface)))
    (setf (chat-gui kriek-chat) contacts-pane)
    (setf (pane-kriek-chat contacts-pane) kriek-chat)
    (capi:display kriek-chat-interface)))
```

Listing C.14: Function to create Kriek chat.

Appendix D

Lambic Geuze

Geuze is implemented using the `lambic-user` package. This package integrates the functionalities of the `cl-user`, `capi`, `color` and `gp` LispWorks packages. We decompose the Geuze program in four parts: definition of classes, graphical operations, GUI events and main user interface. Figure D.1 shows the graphic user interface of Geuze.

D.1 Geuze Classes

D.1.1 Group Classes

```
; Definition of geuze-editor group class
(defgroupclass geuze-editor (pinboard-layout)
  ((username :initarg :username :accessor editor-username)
   (shapes :initform nil :accessor editor-shapes)
   (canvas :accessor editor-canvas)
   (brush-active :initform nil :accessor brush-active)
   (selected-color :initform :black :accessor selected-color)
   (zoom-level :initform 1 :accessor zoom-level)
   (window :accessor editor-window)
   ; GUI temporary state
   (drag-status :initform nil :accessor drag-status)
   (line-status :initform nil :accessor line-status)
   (selection-rectangle :initform nil :accessor selection-rectangle)
   ; Group management
   (in-session :initform nil :accessor in-session?)
   (traffic :initform “normal” :accessor editor-traffic))
```

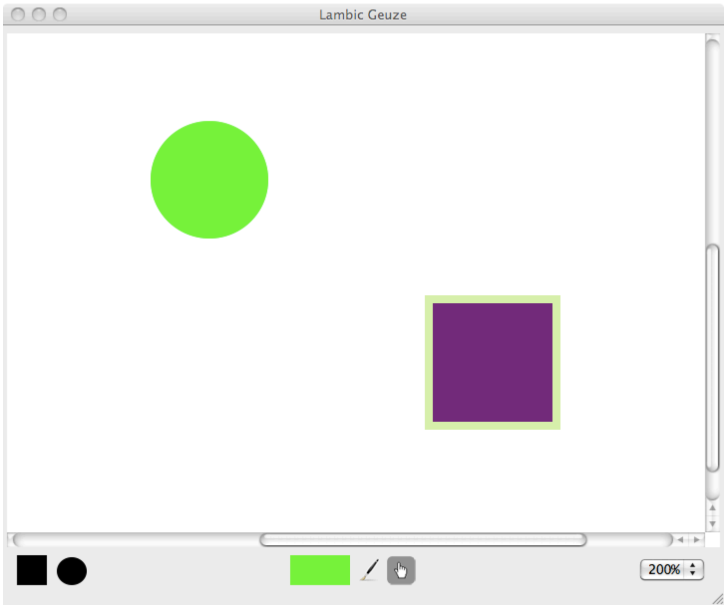


Figure D.1: Geuze's GUI.

```
; Initialise an geuze-editor instance by adding the canvas.
(defmethod initialize-instance :after ((editor geuze-editor))
  (let ((canvas (make-instance 'rectangle
                               :width 700
                               :height 500
                               :graphics-args '(:foreground :white)
                               :filled t)))
    (manipulate-pinboard editor canvas :add-top)
    (setf (editor-canvas editor) canvas)
    (setf (editor-window editor) (list 0 0 700 500))))

; Definition of geuze-shape group class
(defgroupclass geuze-shape (pinboard-object)
  ((name :initarg :name :accessor shape-name)
   (user-editor :initform nil :accessor shape-user)
   ; Group management
   (owner :initarg :owner :accessor shape-owner)))

; Definition of geuze-rectangle group class
(defgroupclass geuze-rectangle (geuze-shape) ())

; Definition of geuze-circle group class
(defgroupclass geuze-circle (geuze-shape) ())
```

```
; Definition of geuze-drawn-shape group class
(defgroupclass geuze-drawn-shape (geuze-shape)
  ((line :initform nil :initarg :line :accessor line)
   (status :initform nil :accessor status)))
```

Listing D.1: Geuze's group class definitions.

D.1.2 Auxiliary Structs

```
; Definition of drag-status struct
(defstruct drag-status x y)

; Definition of shape-record struct
(defstruct shape-record class name owner user x y w h color)

; Definition of drawn-shape-record struct
(defstruct (drawn-shape-record (:include shape-record)) line)
```

Listing D.2: Geuze's struct definitions.

D.2 Graphical Operations

D.2.1 Adding a Shape

```
; Definition of add-shape group generic function
(defgroupgeneric add-shape (editor shape-record)
  (:group-predicates in-session?))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Group method for shape's owner
; Propagate addition to all editors
(defgroupmethod add-shape :uninterruptible ((editor geuze-editor) shape-record)
  (:when (in-session? editor))
  (:propagate ((each editor))
   :return #'local-shape
   :catch (timeout-exception () nil))
  (call-next-method))
```

```
; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Add a shape to editor
(defmethod add-shape ((editor geuze-editor) shape-record)
  (with-slots (class name owner x y w h color)
      shape-record
    (let* ((zoom-level (zoom-level editor))
           (visible-w (* w zoom-level))
           (visible-h (* h zoom-level))
           (shape (make-instance class :owner owner :name name :x x :y y
                                 :width visible-w :height visible-h
                                 :graphics-args (list :background :transparent
                                                      :foreground color))))
      (apply-in-pane-process editor
                             (lambda (editor shape)
                               (manipulate-pinboard editor shape :add-top))
                             editor shape)
      (push shape (editor-shapes editor))
      shape)))

; Add a drawn shape to editor
(defmethod add-shape ((editor geuze-editor) (shape-record drawn-shape-record))
  (with-slots (owner name line color)
      shape-record
    (multiple-value-bind (tl-x tl-y br-x br-y)
        (get-enclosing-rectangle line)
      (let* ((x (- tl-x 5))
             (y (- tl-y 5))
             (width (+ 10 (- br-x tl-x)))
             (height (+ 10 (- br-y tl-y)))
             (shape (make-instance 'geuze-drawn-shape :owner owner :name name :x x
                                   :y y :line line :width width :height height
                                   :graphics-args (list :background :transparent
                                                        :foreground color))))
        (push shape (editor-shapes editor))
        (apply-in-pane-process editor
                               (lambda (editor shape)
                                 (manipulate-pinboard editor shape :add-top))
                               editor shape)
        shape))))
```

Listing D.3: The `add-shape` group generic function.

D.2.2 Deleting a Shape

```
; Definition of delete-shape group generic function
(defgroupgeneric delete-shape (editor shape)
  (:group-predicates in-session? shape-owner? shape-client?))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Group method for shape's owner
; Propagate deletion to all editors
(defgroupmethod delete-shape ((editor geuze-editor) (shape geuze-shape))
  (:when (in-session? editor)
         (shape-owner? shape editor))
  (:propagate ((each editor) (co-located editor shape))
   :return nil)
  (call-next-method))

; Group method for shape's client
; Propagate deletion to shape's owner
(defgroupmethod delete-shape ((editor geuze-editor) (shape geuze-shape))
  (:when (in-session? editor)
         (shape-client? shape editor))
  (:propagate ((shape-owner? shape editor)
               (same-shape-name? editor shape (original shape)))
   :return nil)
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Delete shape from editor
(defmethod delete-shape ((editor geuze-editor) (shape geuze-shape))
  (setf (editor-shapes editor) (remove shape (editor-shapes editor)))
  (apply-in-pane-process editor
    (lambda (editor shape)
      (manipulate-pinboard editor shape :delete))
    editor shape))

; Delete multiple shapes from editor
(defmethod delete-shapes (local-editor remote-editor)
  (loop for shape in (editor-shapes local-editor)
        if (eql (shape-user shape) remote-editor)
        do (delete-shape local-editor shape)))
```

Listing D.4: The `delete-shape` group generic function.

D.2.3 Painting a Shape

```
; Definition of paint-shape group generic function
(defgroupgeneric paint-shape (editor shape color)
  (:group-predicates in-session?))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Propagate the paint to all the shapes selected by the editor
; performing the paint
(defgroupmethod paint-shape ((editor geuze-editor) (shape geuze-shape) color)
  (:propagate ((same-selection? shape (original shape)))
   :return nil)
  (call-next-method))

; Propagate the paint to all editors
(defgroupmethod paint-shape ((editor geuze-editor) (shape geuze-shape) color)
  (:when (in-session? editor))
  (:propagate ((same-shape-name? editor shape (original shape)))
   :return nil)
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Change shape's color
(defmethod paint-shape ((editor geuze-editor) (shape geuze-shape) color)
  (apply-in-pane-process editor
    (lambda (shape color)
      (setf (getf (pinboard-object-graphics-args shape) :foreground) color))
    shape color))
```

Listing D.5: The `paint-shape` group generic function.

D.2.4 Moving a Shape

```
; Definition of move-shape group generic function
(defgroupgeneric move-shape (editor shape x y final-position)
  (:group-predicates in-session?))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Propagate the move to all the shapes selected by the editor
; performing the move
(defgroupmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y
                            final-position)
  (:propagate ((same-selection? shape (original shape)))
   :return nil)
  (call-next-method))

; Propagate the move to all editors
(defgroupmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y
                            final-position)
  (:when (in-session? editor))
  (:propagate ((relevant-position? editor shape final-position)
               (same-shape-name? editor shape (original shape)))
   :return nil)
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Set and display new position for shape
(defmethod move-shape ((editor geuze-editor) (shape geuze-shape) x y
                       final-position)
  (apply-in-pane-process editor
    (lambda (shape x y)
      (with-geometry shape
        (setf (pinboard-pane-position shape)
              (values (+ %x% x) (+ %y% y)))))
      shape x y))
```

Listing D.6: The move-shape group generic function.

D.2.5 Drawing a Shape

```
; Draw circle
(defmethod draw-pinboard-object ((editor geuze-editor) (shape geuze-circle))
  (with-geometry rectangle
    (let* ((args (pinboard-object-graphics-args rectangle))
           (level (zoom-level editor))
           (position-offset (* 4 level))
           (size-offset (* 8 level)))
      (if (shape-user rectangle)
        (draw-rectangle editor %x% %y% %width% %height%
                        :foreground (find-selection-color rectangle editor)
                        :filled t))
      (draw-rectangle editor (+ %x% position-offset) (+ %y% position-offset)
                      (- %width% size-offset) (- %height% size-offset)
                      :foreground (getf args :foreground) :filled t))))

; Draw rectangle
(defmethod draw-pinboard-object ((editor geuze-editor) (shape geuze-rectangle))
  (with-geometry ellipse
    (let* ((args (pinboard-object-graphics-args ellipse))
           (half-width  (floor (1- %width%)  2))
           (half-height (floor (1- %height%) 2))
           (circle-x (+ %x% half-width))
           (circle-y (+ %y% half-height))
           (level (zoom-level editor))
           (size-offset (* 4 level)))
      (if (shape-user ellipse)
        (draw-ellipse editor circle-x circle-y half-width half-height
                      :foreground (find-selection-color ellipse editor)
                      :filled t))
      (draw-ellipse editor circle-x circle-y
                    (- half-width size-offset) (- half-height size-offset)
                    :foreground (getf args :foreground) :filled t))))

; Draw drawn shape
(defmethod draw-pinboard-object ((editor geuze-editor)
                                 (shape geuze-drawn-shape)
                                 &key &allow-other-keys)
  (let* ((line (line shape))
         (args (pinboard-object-graphics-args shape))
         (level (zoom-level editor))
         (shadow-thickness (* 10 level))
         (line-thickness (* 4 level)))
    (if (shape-user shape)
      (draw-lines editor line
                  :foreground (find-selection-color shape editor)
                  :thickness shadow-thickness
                  :line-end-style :round :line-joint-style :round))
    (draw-lines editor line :foreground (getf args :foreground)
                :line-end-style :round :line-joint-style :round
                :thickness line-thickness)))

; Draw temporary line
(defun draw-temporary-line (editor x y)
  (let* ((line-status (line-status editor))
         (old-x (car line-status))
         (old-y (cdr line-status)))
    (draw-line editor old-x old-y x y :foreground (selected-color editor)
               :line-end-style :round :line-joint-style :round :thickness 4)))

; Create shape out of drawn line
(defun create-drawn-shape (editor)
  (let* ((line-status (line-status editor))
         (shape-record (make-instance 'drawn-shape-record
                                      :name (generate-shape-name
                                             'drawn-shape-record)
                                      :owner editor :line line-status
                                      :color (selected-color editor))))
    (add-shape editor shape-record)))
```

Listing D.7: The `draw-pinboard-object` CAPI generic function.

D.2.6 Selecting a Shape

```
; Definition of select-shape group generic function
(defgroupgeneric select-shape (editor shape user-editor)
  (:group-predicates in-session? shape-owner? shape-client?))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Group method for shape's owner
; Propagate selection to all editors
(defgroupmethod select-shape ((editor geuze-editor) (shape geuze-shape)
                              user-editor)
  (:when (in-session? editor)
         (shape-owner? shape editor))
  (:propagate ((same-shape-name? editor shape (original shape)))
   :return nil)
  (if (not (shape-user shape))
    (begin
      (call-next-method)
      shape)))

; Group method for shape's client
; Propagate selection to shape's owner
(defgroupmethod select-shape ((editor geuze-editor) (shape geuze-shape)
                              user-editor)
  (:when (in-session? editor)
         (shape-client? shape editor))
  (:propagate ((shape-owner? shape editor)
               (same-shape-name? editor shape (original shape)))
   :catch (timeout-exception () nil))
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Set editor as shape's user and display selection effect
(defmethod select-shape ((editor geuze-editor) (shape geuze-shape) user-editor)
  (setf (shape-user shape) user-editor)
  (apply-in-pane-process editor
    (lambda (shape)
      (invalidate-rectangle (pinboard-object-pinboard shape)))
    shape))
```

Listing D.8: The `select-shape` group generic function.

D.2.7 Drawing Selection

```
; Draw selection rectangle. Select all deselected shapes,
; deselect all selected shapes.
(defun draw-selection-rectangle (editor x y)
  (let ((selected-rectangle (selection-rectangle editor)))
    (if selected-rectangle
        (multiple-value-bind (ox oy)
            (pinboard-pane-position selected-rectangle)
          (setf (pinboard-pane-size selected-rectangle)
                (values (- x ox) (- y oy)))
          (foreach (shape (editor-shapes editor))
            (let ((coords (get-coords selected-rectangle)))
              (with-geometry shape
                (if (or (inside-rectangle coords %x% %y%)
                        (inside-rectangle coords %x%
                                          (+ %y% %height%)))
                    (if (not (shape-user shape))
                      (select-shape editor shape editor))
                  (if (shape-user shape)
                    (deselect-shape editor shape)))))))
      (begin
        (setf selected-rectangle
              (make-instance 'rectangle
                             :x x :y y :external-min-width 0 :external-min-height 0
                             :visible-min-width 0 :visible-min-height 0
                             :internal-min-width 0 :internal-min-width 0
                             :graphics-args '(:foreground :selection-color)
                             :filled nil))
        (setf (selection-rectangle editor) selected-rectangle)
        (manipulate-pinboard editor selected-rectangle :add-top)))))
```

Listing D.9: The draw-selection-shape group generic function.

D.2.8 Deselecting a Shape

```
; Definition of deselect-shape group generic function
(defgroupgeneric deselect-shape (editor shape)
  (:group-predicates in-session? shape-owner? shape-client?))

; GROUP LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Group method for shape's owner
; Propagate deselection to all editors
(defgroupmethod deselect-shape ((editor geuze-editor) (shape geuze-shape))
  (:when (in-session? editor)
         (shape-owner? shape editor))
  (:propagate ((same-shape-name? editor shape (original shape)))
   :return nil)
  (call-next-method))

; Group method for shape's client
; Propagate deselection to shape's owner
(defgroupmethod deselect-shape ((editor geuze-editor) (shape geuze-shape))
  (:when (in-session? editor)
         (shape-client? shape editor))
  (:propagate ((shape-owner? shape editor)
               (same-shape-name? editor shape (original shape)))
   :return nil))
  (call-next-method))

; PEER LEVEL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Set nil as shape's user and remove selection effect
(defmethod deselect-shape ((editor geuze-editor) (shape geuze-shape))
  (setf (shape-user shape) nil)
  (apply-in-pane-process editor
    (lambda (shape)
      (invalidate-rectangle (pinboard-object-pinboard shape)))
    shape))

; Deselect editor's shapes used by editor given as parameter.
(defmethod deselect-shapes ((editor geuze-editor) user-editor)
  (foreach (shape (editor-shapes editor))
    (if (eql (shape-user shape) user-editor)
        (deselect-shape editor shape))))
```

Listing D.10: The `deselect-shape` group generic function.

D.2.9 Updating a Shape

```
; Update shapes based on records
(defmethod update-shapes ((editor geuze-editor) shape-records)
  (foreach (shape-record shape-records)
    (update-shape editor shape-record)))

; Update shape
(defmethod update-shape ((editor geuze-editor) (shape-record drawn-shape-record))
  (let ((shape (find-shape-with-record editor shape-record)))
    (if shape
      (apply-in-pane-process editor
        (lambda (shape shape-record editor)
          (with-slots (line version color)
              shape-record
            (setf (getf (pinboard-object-graphics-args shape) :foreground) color)
            (setf (line shape) line)
            (manipulate-pinboard editor shape :add-top)))
        shape shape-record editor))))

; Update drawn shape
(defmethod update-shape ((editor geuze-editor) (shape-record shape-record))
  (let ((shape (find-shape-with-record editor shape-record)))
    (if shape
      (apply-in-pane-process editor
        (lambda (shape shape-record editor)
          (with-slots (x y w h version color)
              shape-record
            (setf (getf (pinboard-object-graphics-args shape) :foreground) color)
            (set-pinboard-object-geometry shape :x x :y y :width w :height h)
            (manipulate-pinboard editor shape :add-top)))
        shape shape-record editor))))

; Find hidden shape
(defun find-shape-with-record (editor shape-record)
  (loop for shape in (editor-shapes editor)
        when (equal (shape-name shape) (shape-record-name shape-record))
        return shape))
```

Listing D.11: The `update-shapes` generic function.

D.2.10 Predicates and Return Functions for Graphical Operations

```
; Check whether new shape's position should be
; propagated to the session.
(defgeneric relevant-position? (editor shape final-position)
  (:predicates medium-traffic? low-traffic?))

; Always propagate the new positions (normal traffic)
(defmethod relevant-position? (editor shape final-position)
  t)

; Propagate the new positions only to the editors
; that are currently displaying the shape. Additionally, propagate
; the final position to everyone.
(defmethod relevant-position? (editor shape final-position)
  (:when (medium-traffic? local-editor))
  (or final-position
      (on-focus? editor shape))

; Propagate only the final position.
(defmethod relevant-position? (editor shape final-position)
  (:when (low-traffic? local-editor))
  final-position)

; Check whether editor's traffic rate is set to medium
(defmethod low-traffic? (editor)
  (equal (editor-traffic editor) "medium"))

; Check whether editor's traffic rate is set to low
(defmethod low-traffic? (editor)
  (equal (editor-traffic editor) "low"))

; Check whether shape is being displayed by editor
(defmethod on-focus? :uninterruptible (editor shape)
  (let ((shapes (try-catch (shapes-on-focus editor)
                 (timeout-exception () nil))))
    (and (co-located editor shape)
         (find shape shapes))))

; Check whether editor owns shape
(defmethod shape-owner? :uninterruptible (shape editor)
  (let ((owner (try-catch (shape-owner shape)
                (timeout-exception () nil))))
    (equal owner editor)))
```

```
; Check whether editor does not own shape
(defmethod shape-client? (shape editor)
  (not (shape-owner? shape editor)))

; Check whether shape has the same name as the shape originally
; passed as argument to the group generic function
(defmethod same-shape-name? :uninterruptible (editor shape original-shape)
  (let ((shape-name (try-catch (shape-name shape)
                      (timeout-exception () nil))))
      (and (equal shape-name (shape-name original-shape))
           (co-located? editor shape))))

; Check whether shape has the same user as the shape originally
; passed as argument to the group generic function
(defmethod same-selection? ((shape geuze-shape) (original-shape geuze-shape))
  (equal (shape-user shape) (shape-user original-shape)))

; Return nil for arguments that are not local instances of
; geuze-shape
(defmethod same-selection? (shape original-shape)
  nil)

; Return local shape
(defmethod local-shape (shapes)
  (find-if #'in-current-actor shapes))
```

Listing D.12: Context and propagation predicates for Geuze's graphical operations.

D.3 Handling Discovery and Connectivity Events

```
; Handle peer discovery
(defmethod peer-discovered ((local-editor geuze-editor) remote-editor)
  (send-owned-shapes local-editor remote-editor))

; Handle peer disconnections
(defmethod peer-disconnected ((local-editor geuze-editor) remote-editor)
  (hide-disconnected-shapes local-editor remote-editor)
  (deselect-shapes local-editor remote-editor))
```

```
; Handling peer reconnection
(defmethod peer-reconnected ((local-editor geuze-editor) remote-editor)
  (send-owned-shapes local-editor remote-editor))

(defmethod hide-disconnected-shapes (local-editor remote-editor)
  (foreach (shape (editor-shapes local-editor))
    (if (eql (shape-owner shape) remote-editor)
      (apply-in-pane-process local-editor
        (lambda (shape editor)
          (manipulate-pinboard editor shape :delete))
        shape local-editor))))

; Send local editor's shapes to remote editor
(defmethod send-owned-shapes (local-editor remote-editor)
  (let ((shape-records (generate-shape-records local-editor)))
    (in-actor-of remote-editor
      (update-shapes remote-editor shape-records)
      :with-future nil)))

; Get a record with the information of each shape
(defun generate-shape-records (editor)
  (loop for shape in (editor-shapes editor)
        when (eql editor (shape-owner shape))
        collect (generate-shape-record shape)))

; Generate a shape-record from a geuze-shape object
(defmethod generate-shape-record ((shape geuze-shape))
  (with-geometry shape
    (make-instance 'shape-record
                   :class (class-of shape) :name (shape-name shape)
                   :owner (shape-owner shape) :user (shape-user shape)
                   :x %x% :y %y% :w %width% :h %height%
                   :color (getf (pinboard-object-graphics-args
                                 shape) :foreground))))

; Generate a drawn-shape-record from a geuze-drawn-shape object
(defmethod generate-shape-record ((shape geuze-drawn-shape))
  (make-instance 'drawn-shape-record
                 :name (shape-name shape) :owner (shape-owner shape)
                 :user (shape-user shape) :line (line shape)
                 :color (getf (pinboard-object-graphics-args
                               shape) :foreground)))
```

Listing D.13: Handling peer discovery and connectivity events.

D.4 User Interface Events

D.4.1 Handling Mouse Events

Handling *mouse-down* Event

```
; The mouse-down generic function
(defgeneric mouse-down (editor shape x y)
  (:predicates painting? moving? drawing? drawing-selection?
               selecting? deselecting?))

; mouse-down when painting shape
(defmethod mouse-down (editor shape x y)
  (:when (painting? shape editor))
  (paint-shape editor shape (selected-color editor)))

; mouse-down when moving shape
(defmethod mouse-down ((editor geuze-editor) shape x y)
  (:when (moving? shape editor))
  (setf (drag-status editor) (make-drag-status :x x :y y)))

; mouse-down when drawing shape
(defmethod mouse-down (editor shape x y)
  (:when (drawing? shape editor))
  (setf (line-status editor) (cons x y)))

; mouse-down when selecting shape
(defmethod mouse-down ((editor geuze-editor) shape x y)
  (:when (selecting? shape editor))
  (if (select-shape editor shape editor)
    (call-next-method)
    (display-message “Shape already in use.”)))

; mouse-down when deselecting shape
(defmethod mouse-down (editor shape x y)
  (:when (deselecting? shape editor))
  (deselect-shapes editor editor))

; GUI predicates
; Enters into moving mode if a shape is passed as argument
; and the editor’s brush is not selected
(defmethod moving? ((editor geuze-editor) shape)
  (and shape (not (brush-active editor))))
```

```
; Enters into selecting mode if a shape is passed as argument
; and it is not in use already
(defmethod selecting? ((editor geuze-editor))
  (and shape (not (shape-user shape))))
```

Listing D.14: The `mouse-down` generic function.

Handling *mouse-move* Event

```
; The mouse-move generic function
(defgeneric mouse-move (editor shape x y)
  (:predicates moving? drawing? drawing-selection?))

; mouse-move when moving
(defmethod mouse-move ((editor geuze-editor) shape x y)
  (:when (moving? shape editor))
  (move-shape editor shape x y nil)
  (let ((status (drag-status editor)))
    (setf (drag-status-x status) x (drag-status-y status) y)))

; mouse-move when drawing
(defmethod mouse-move (editor shape x y)
  (:when (drawing? shape editor))
  (draw-temporary-line editor x y)
  (setf line-status `(,x ,y ,@line-status)))

; mouse-move when drawing selection
(defmethod mouse-move (editor shape x y)
  (:when (drawing-selection? editor))
  (draw-selection-rectangle editor x y))
```

Listing D.15: The `mouse-move` generic function.

Handling *mouse-up* Event

```
; The mouse-up generic function
(defgeneric mouse-up (editor shape x y)
  (:predicates moving? drawing? drawing-selection?))
```

```
; mouse-up when moving
(defmethod mouse-up (editor shape x y)
  (:when (moving? shape editor))
  (move-shape editor shape x y t)
  (setf (drag-status editor) nil)
  (invalidate-rectangle editor))

; mouse-up when drawing
(defmethod mouse-up (editor shape x y)
  (:when (drawing? shape editor))
  (create-drawn-shape editor)
  (setf (line-status editor) nil)

; mouse-up when drawing selection
(defmethod mouse-up (editor shape x y)
  (:when (drawing-selection? editor))
  (manipulate-pinboard editor (selection-rectangle editor) :delete)
  (setf (selection-rectangle editor) nil))
```

Listing D.16: The `mouse-up` generic function.

Predicates for Mouse Events

```
; Check painting-shape mode
(defmethod painting? (editor shape)
  (and shape (brush-active editor)))

; Check moving-shape mode
(defmethod moving? (editor shape)
  (and shape (not (brush-active editor))))

; Check drawing-shape mode
(defmethod drawing? (editor shape)
  (and (not shape) (brush-active editor)))

; Check drawing-selection mode
(defmethod drawing-selection? (editor shape)
  (and (not shape) (not (brush-active editor))))

; Check selecting-shape mode
(defmethod selecting? (editor shape)
  (and shape (not (shape-user shape))))
```

```
; Check deselecting-shape mode
(defmethod deselecting? (editor shape)
  (not shape))
```

Listing D.17: The `mouse-up` generic function.

D.4.2 Other Event Handlers

```
; Window position changed
(defun scroll-editor-callback (editor-pane scroll-dimension scroll-operation
                               scroll-value &key interactive &allow-other-keys)
  (let ((new-x (get-horizontal-scroll-parameters editor-pane :slug-position))
        (new-y (get-vertical-scroll-parameters editor-pane :slug-position)))
    (if (eq scroll-operation :move)
      (setf (editor-window editor-pane) (list new-x new-y 700 500)))))

; Color button selected
(defun option-pane-set-color (pane x y)
  (let ((object (pinboard-object-at-position pane x y)))
    (if object
      (let ((current-color (getf (pinboard-object-graphics-args object)
                                 :foreground)))
        (multiple-value-bind (new-color successp)
            (prompt-for-color "Colors" :color current-color)
          (when successp
            (setf (getf (pinboard-object-graphics-args object) :foreground)
                  new-color)
            (let ((editor (editor (top-level-interface pane))))
              (setf (selected-color editor) new-color))))))))

; Finger button selected
(defun set-cursor-mode (interface)
  (setf (simple-pane-cursor (editor interface)) *br-finger-cursor*)
  (setf (brush-active (editor interface)) nil))

; Brush button selected
(defun set-painting-mode (interface)
  (setf (simple-pane-cursor (editor interface)) *brush-cursor*)
  (setf (brush-active (editor interface)) t))
```

```
; Zoom option button changed
(defun change-zoom (zoom-percentage interface)
  (let* ((editor (editor interface))
         (all-shapes (reverse (layout-description editor)))
         (new-zoom-level (/ zoom-percentage 100))
         (delta (/ new-zoom-level (zoom-level editor))))
    (when (and all-shapes (not (eql delta 1)))
      (foreach (shape all-shapes)
        (with-geometry shape
          (set-pinboard-object-geometry shape
                                        :x (* %x% delta) :y (* %y% delta)
                                        :width (* %width% delta)
                                        :height (* %height% delta))
          (when (eql shape (editor-canvas editor))
            (set-horizontal-scroll-parameters editor :max-range %width%)
            (set-vertical-scroll-parameters editor :max-range %height%))))
      (setf (zoom-level editor) new-zoom-level))))

; Toolbar shape selected
(defun drag-from (pane x y)
  (let ((object (pinboard-object-at-position pane x y)))
    (if object
      (let ((value (capi-object-name object)))
        (drag-pane-object pane value)))))

; Toolbar shape dropped
(defun drop-shape-callback (editor drop-object stage)
  (case stage
    (:formats
     (set-drop-object-supported-formats drop-object '(:value)))
    (:enter
     (if (and (drop-object-provides-format drop-object :value)
              (drop-object-allows-drop-effect-p drop-object :copy))
       (setf (drop-object-drop-effect drop-object) :copy)))
    (:drag
     (if (and (drop-object-provides-format drop-object :value)
              (drop-object-allows-drop-effect-p drop-object :copy))
       (setf (drop-object-drop-effect drop-object) :copy)))
    (:drop
     (if (and (drop-object-provides-format drop-object :value)
              (drop-object-allows-drop-effect-p drop-object :copy))
       (set-pane-focus editor)
       (deselect-shapes editor editor)
       (let* ((value (drop-object-get-object drop-object editor :value))
              (shape-record (make-instance 'geuze-shape-record :w 45 :h 45
                                           :name (generate-shape-name value)
                                           :class value :owner editor
                                           :x (drop-object-pane-x drop-object)
```

```
                                   :y (drop-object-pane-y drop-object)
                                   :color (selected-color editor))))
          (add-shape editor shape-record)
          (setf (drop-object-drop-effect drop-object) :copy))))))

; Deletion key pressed
; Key: Delete
(defun deletion-key-pressed (editor x y key)
  (delete-shapes editor editor))

; Zoom option in percentage
(defun option-in-percentage (option)
  (format nil "~a%" option))

; Get shape at a given position.
(defun get-shape-at-position (editor x y)
  (let ((shape (pinboard-object-at-position editor x y)))
    (unless (eql (editor-canvas editor) shape)
      shape)))

; Generate a shape name
(defun generate-shape-name (name)
  (read-from-string (format nil "~d-fig-~d" name (gensym))))

; Find selection color
(defmethod find-selection-color (shape editor)
  (if (in-current-actor editor)
    :selection-color
    (let* ((user (shape-user shape))
           (pos (position user (peers editor)))
           (colors (get-chromatic-color-names)))
      (nth pos colors))))

; Return list of shapes currently being displayed in
; editor's window
(defmethod shapes-on-focus ((editor geuze-editor))
  (loop for shape in (editor-shapes editor)
        when (inside-window editor shape)
        collect shape))

; Check whether shape is contained within window geometry
(defmethod inside-window (window shape)
  (let ((window (editor-window editor))
        (zoom (editor-zoom-level editor)))
    (with-geometry shape
      (let ((shape-x (* %x% zoom))
            (shape-y (* %y% zoom))
            (shape-xw (* (+ %x% %width%) zoom))
            (shape-yh (* (+ %y% %height%) zoom)))
        (or (inside-rectangle window shape-x shape-y)
            (inside-rectangle window shape-x shape-yh)
            (inside-rectangle window shape-xw shape-y)
            (inside-rectangle window shape-xw shape-yh))))))
```

Listing D.18: Other user interface events.

D.5 Main User Interface

```
; Definition of main interface
(define-interface editor-interface ()
  ()
  (:panes
   (drag-layout
    pinboard-layout
    :description *toolbar-shapes*
    :background :transparent
    :input-model '(((:button-1 :press) drag-from))
    :visible-min-width 280)
   (color-pane
    pinboard-layout
    :description (list (make-instance 'rectangle
                                      :filled t :x 0 :y 5 :width 60 :height 30
                                      :graphics-args '(:foreground :black)))
    :input-model '(((:button-1 :press) option-pane-set-color))
    :background :transparent)
   (buttons-pane
    button-pinboard-pane
    :description *toolbar-buttons*
    :selected-button-position 0
    :background :transparent :visible-min-width 280)
   (zoom-pane
    option-pane
    :accessor color-chooser :items '(25 50 100 150 200 400 800)
    :selected-item 100 :print-function 'option-in-percentage
    :selection-callback 'change-zoom :visible-max-width 70)
   (editor
    geuze-editor
    :scroll-callback 'scroll-editor-callback
    :drop-callback 'drop-shape-callback :accessor editor
    :draw-pinboard-objects :local-buffer
    :background :gray :cursor *br-finger-cursor* :horizontal-scroll t
    :vertical-scroll t :scroll-width 700 :scroll-height 500
    :fit-size-to-children nil :input-model *editor-input-model*))
  (:layouts
   (bottom-layout
    row-layout
    '(drag-layout color-pane buttons-pane zoom-pane)
    :adjust :center :uniform-size-p nil :visible-max-height 40
    :ratios '(nil nil nil nil) :y-adjust :center)
   (main-layout
    column-layout
    '(editor bottom-layout)))
  (:default-initargs
   :layout 'main-layout :title "Lambic Geuze"
   :best-width 723 :best-height 582))
```

Listing D.19: Main user interface.

D.5.1 Auxiliary Parameters

```
; Cursor icon
(defparameter *brush-cursor*
  (load-cursor
    '((:cocoa
       #.(current-pathname "images/brush-cursor.tif")
       :x-hot 2 :y-hot 40))))

; Finger icon
(defparameter *br-finger-cursor*
  (load-cursor
    '((:cocoa
       #.(current-pathname "images/br-finger-cursor.tif")
       :x-hot 17 :y-hot 11))))

; Default local selection color
(define-color-alias :selection-color
                    (make-rgb 0.15s0 0.51s0 0.93s0 0.4s0))

; Shapes in the toolbar
(defparameter *toolbar-shapes*
  (list (make-instance 'rectangle
                       :filled t :x 10 :y 5 :width 30 :height 30
                       :graphics-args '(:foreground :black)
                       :name 'geuze-rectangle)
        (make-instance 'ellipse
                       :filled t :x 50 :y 5 :width 30 :height 30
                       :graphics-args '(:foreground :black)
                       :name 'geuze-ellipse)))

; Buttons in the toolbar
(defparameter *toolbar-buttons*
  (list (make-instance 'checkable-button
                       :x 0 :y 5
                       :checked-image-location
                       #.(current-pathname
                          "images/selected-brush.png")
                       :unchecked-image-location
                       #.(current-pathname
                          "images/unselected-brush.png")
                       :when-checked 'set-painting-mode)
        (make-instance 'checkable-button
                       :x 32 :y 5
                       :checked-image-location
                       #.(current-pathname
                          "images/selected-finger.png")
                       :unchecked-image-location
                       #.(current-pathname
                          "images/unselected-finger.png")
                       :when-checked 'set-cursor-mode
                       :checkedp t)))

; Input model
(defparameter *editor-input-model*
        `(((:button-1 :press)
            ,(lambda (editor x y)
               (let ((pressed-shape (get-shape-at-position editor x y)))
                 (mouse-down editor pressed-shape x y))))
          ((:button-1 :motion)
            ,(lambda (editor x y)
               (let ((pressed-shape (get-shape-at-position editor x y)))
                 (mouse-move editor pressed-shape x y))))
          ((:button-1 :release)
            ,(lambda (editor x y)
               (let ((pressed-shape (get-shape-at-position editor x y)))
                 (mouse-up editor pressed-shape x y))))
          ((#\del :press) deletion-key-pressed)))
```

Listing D.20: Auxiliary parameters for user interface.

D.5.2 Main Function

```
; Geuze editor's main function
(defun make-lambic-geuze (username)
  (let* ((interface (make-instance 'editor-interface))
         (editor (editor interface)))
    (setf (editor-username editor) username)
    (capi:display interface)))
```

Listing D.21: Function to create Geuze editor.

Bibliography

[AFK$^+$93] Gul Agha, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel Distrib. Technol.*, 1(2):3–14, 1993.

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AHH$^+$09] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-oriented Programming Languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM.

[AHM$^+$10] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific Software Composition in Context-oriented Programming. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 50–65, Berlin, Heidelberg, 2010. Springer-Verlag.

[AHT$^+$02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[Ame91] Pierre America. POOL: design and experience. *ACM SIGPLAN OOPS Messenger*, 2:16–20, April 1991.

[App11] Apple Inc. Networking Bonjour Protocol, 2006-2011.

[Bai08] Engineer Bainomugisha. Resilient Service Partitioning for Pervasive Computing Services. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 2008.

[Bar05] Jakob Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In Hans Gellersen, Roy Want, and Albrecht Schmidt, editors, *Pervasive Computing*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115. Springer Berlin / Heidelberg, 2005.

[BBC02] Laurent Baduel, Francoise Baude, and Denis Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 28–36, New York, NY, USA, 2002. ACM.

[BBC$^+$06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BC06] Paolo Bellavista and Antonio Corradi. *The Handbook of Mobile Middleware.* Auerbach Publications, Boston, MA, USA, 2006.

[BCC$^+$10] Engineer Bainomugisha, Alfredo Cádiz, Pascal Costanza, Wolfgang De Meuter, Sebastián González, Kim Mens, Jorge Vallejos, and Tom Van Cutsem. *Language Engineering for Mobile Software*, chapter Language Engineering for Mobile Software. IGI Global, 2010.

[BCH$^+$96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. *ACM SIGPLAN Notices*, 31(10):69–82, 1996.

[BD96] Daniel Bardou and Christophe Dony. Split Objects: A Disciplined Use of Delegation within Objects. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 122–137. ACM Press, 1996.

[BD07] Genevieve Bell and Paul Dourish. Back to the Shed: Gendered Visions of Technology and Domesticity. *Personal Ubiquitous Computing*, 11:373–381, June 2007.

[BDG$^+$88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Not.*, 23(SI):1–143, 1988.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[BI93] Andrew P. Black and Mark P. Immel. Encapsulating Plurality. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 57–79, London, UK, 1993. Springer-Verlag.

[BKZD04] Michael Beigl, Albert Krohn, Tobias Zimmer, and Christian Decker. Typical Sensors needed in Ubiquitous and Pervasive Computing. *Economic Affairs*, 4(Figure 1):153–158, 2004.

[CC99] Craig Chambers and Weimin Chen. Efficient Multiple and Predicated Dispatching. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 238–255, New York, NY, USA, 1999. ACM.

[CD08] Pascal Costanza and Theo D'Hondt. Feature Descriptions for Context-oriented Programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08), co-located with Software Product Line Conference 2008 (SPLC2008)*, pages 9–14, September 2008.

[CH05] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-Oriented Programming - An overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[Cha93] Craig Chambers. Predicate Classes. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 268–296, London, UK, 1993. Springer-Verlag.

[CHVD08] Pascal Costanza, Charlotte Herzeel, Jorge Vallejos, and Theo D'Hondt. Filtered Dispatch. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2008. ACM.

[CK06] Gregory Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin / Heidelberg, 2006.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[CMV$^+$08] Alfredo Cádiz, Boris Mejías, Jorge Vallejos, Kim Mens, Peter Van Roy, and Wolfgang de Meuter. PALTA: Peer-to-peer AdaptabLe Topology for Ambient intelligence. In *Proceedings of the 2008 International Conference of the Chilean Computer Science Society*, pages 100–109, Washington, DC, USA, 2008. IEEE Computer Society.

[CSJR02] K. Chandra Sekharaiah and D. Janaki Ram. Object Schizophrenia Problem in Object Role System Design. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Object-Oriented Information Systems*, volume 2425 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin / Heidelberg, 2002.

[Ded06] Jessie Dedecker. *Ambient-Oriented Programming.* PhD thesis, Vrije Universiteit Brussel, 2006.

[Der99] Michael L. Dertouzos. The Future of Computing. *Scientific American*, pages 52–55, august 1999.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:107–113, January 2008.

[Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: a Personal Perspective.* Springer-Verlag New York, Inc., New York, NY, USA, 1982.

[DL05] Pierre-Charles David and Thomas Ledoux. WildCAT: A Generic Framework for Context-aware Applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[DVC+07] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-Oriented Domain Analysis. In *6th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT 2007)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, August 2007.

[DVCM+06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming in AmbientTalk. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer Berlin / Heidelberg, 2006.

[DVV+07] Brecht Desmet, Kristof Vanhaesebrouck, Jorge Vallejos, Pascal Costanza, and Wolfgang De Meuter. The Puzzle Approach for Designing Context-Enabled Applications. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 23–29, Washington, DC, USA, 2007. IEEE Computer Society.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EGS00] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 252–276, London, UK, 2000. Springer-Verlag.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054092.

[Eri11] Ericsson. The Erlang Programming Language. http://www.erlang.org, 1986-2011.

[Eug07] Patrick Eugster. Type-based Publish/Subscribe: Concepts and Experiences. *ACM Transactions on Programming Languages and Systems*, 29(1):6, 2007.

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language Support for Event-driven Programming. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 134–143, New York, NY, USA, 2007. ACM.

[Fou11] Python Software Foundation. The Python Programming Language. http://www.python.org, 1991-2011.

[Frø92] Svend Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In Ole Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 185–196. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0053037.

[Gar05] Jesse James Garrett. Ajax: A New Approach to Web Applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005.

[GBCV$^+$09] Elisa Gonzalez Boix, Tom Cutsem, Jorge Vallejos, Wolfgang Meuter, and Theo D'Hondt. A Leasing Model to Deal with Partial Failures in Mobile Ad Hoc Networks. In *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 231–251. Springer Berlin Heidelberg, 2009.

[GDL$^+$04] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System Support for Pervasive Applications. *ACM Transactions on Computer Systems*, 22:421–486, November 2004.

[Ger05] Dirk Gerrits. Erlisp, Common Lisp Library. http://common-lisp.net/project/erlisp, 2005.

[GF99] Rachid Guerraoui and Mohamed E. Fayad. OO Distributed Programming is *Not* Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, 1999.

[GFGM98] Rachid Guerraoui, Pascal Felber, Benoît Garbinato, and Karim Mazouni. System Support for Object Groups. *SIGPLAN Not.*, 33(10):244–258, 1998.

[GG97] Benoît Garbinato and Rachid Guerraoui. Using the Strategy Design Pattern to Compose Reliable Distributed Protocols. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 17–17, Berkeley, CA, USA, 1997. USENIX Association.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GMH07] Sebastián González, Kim Mens, and Patrick Heymans. Highly Dynamic Behaviour Adaptability Through Prototypes with Subjective Multimethods. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 77–88, New York, NY, USA, 2007. ACM.

[Goo09] Google Inc. Google Wave. http://wave.google.com, 2009.

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GSW$^+$02] Luke Gorrie, Vladimir Sekissov, David Wallin, Mats Cronqvist, and Martin Björklund. Distel: Distributed Emacs Lisp (for Erlang). http://fresh.homeunix.net/ luke/distel, 2002.

[Hal85] Robert H. Halstead, Jr. MULTILISP: a Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Har08] Klaus Harbo. CL-MUPROC: Erlang-inspired Multiprocessing in Common Lisp. http://common-lisp.net/project/cl-muproc, 2008.

[HCD08] Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. Reflection for the Masses. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 87–122. Springer Berlin / Heidelberg, 2008.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*. `http://www.jot.fm`, 7(3), March-April 2008.

[HO06] Philipp Haller and Martin Odersky. Event-Based Programming without Inversion of Control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[HO09] Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science - Elsevier*, 410(2-3):202–220, 2009.

[Hoh06] Gregor Hohpe. Programming Without a Call Stack – Event-driven Architectures. *OBJEKTspektrum*, 02:18–24, February 2006.

[HRB$^+$91] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language. Technical report, Dept. of Computer Science, University of British Columbia, Vancouver, Canada, october 1991.

[HT99] Lothar Hotz and Michael Trowe. NetCLOS - Parallel Programming in Common Lisp. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2034–2040, 1999.

[Hua09] Jianyi Huang. Language Support For Dynamic Mashups In the Internet of Things. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 2009.

[Kin05] Ken Kinder. Event-driven programming with twisted and python. *Linux Journal*, 2005(131):6, 2005.

[KRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.

[Lie87] Henry Lieberman. Concurrent Object-Oriented Programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[MDC96] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A Semantics of Introspection in a Reflective Prototype-Based Language. In *LISP AND SYMBOLIC COMPUTATION*, pages 153–180, 1996.

[MES05] Mark S. Miller, Dean E. Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

[MFRW09] Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and Modular Predicate Dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 31(2):1–54, 2009.

[MGB$^+$09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 1–20, New York, NY, USA, 2009. ACM.

[Mil04] Todd Millstein. Practical Predicate Dispatch. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2004. ACM Press.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[MPR06] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology*, 15:279–328, July 2006.

[MR07] Boris Mejías and Peter Van Roy. A Relaxed-Ring for Self-Organising and Fault-Tolerant Peer-to-Peer Networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[MS04] Giuseppe Milicia and Vladimiro Sassone. The Inheritance Anomaly: Ten Years After. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274. ACM Press, 2004.

[MV10] Boris Mejías and Peter Van Roy. Beernet: Building Self-Managing Decentralized Systems with Replicated Transactional Storage. *IJARAS: International Journal of Adaptive, Resilient, and Autonomic Systems*, 1(3):1–24, July - September 2010.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages. In *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press, 1993.

[NCT04] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, New York, NY, USA, 2004. ACM.

[NR08] Jim Newton and Christophe Rhodes. Custom Specializers in Object-Oriented Lisp. *Journal of Universal Computer Science*, 14(20):3370–3388, 2008.

[NSV$^+$06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly Distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM.

[Oli11] Oliver Widder. Geek and Poke. http://geekandpoke.typepad.com, 2011.

[Pro10] Programming Methods Laboratory of EPFL. The Scala Programming Language. http://www.scala-lang.org, 2003-2010.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, London, UK, 2001. Springer-Verlag.

[PVW$^+$04] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an Extensible Context Ontology for Ambient Intelligence. In *Ambient Intelligence*, pages 148–159, 2004.

[RC04] Giacomo Rizzolatti and Laila Craighero. The Mirror-Neuron System. *Annual Review of Neuroscience*, 27:169–192, 2004.

[Ric90] C. Richardson. LispWorks: A Common Lisp Programming Environment for Unix Workstations. pages 127–134, 1990.

[Rob10] Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. The Lua Programming Language. http://www.lua.org, 1993-2010.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-enabled Applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[Sei05] Peter Seibel. *Practical Common Lisp*. Apress Series. Apress, 2005.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer Berlin / Heidelberg, 2008.

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer Berlin / Heidelberg, 2010.

[SRRB10] Bruno Silvestre, Silvana Rossetto, Noemi Rodriguez, and Jean-Pierre Briot. Flexibility and Coordination in Event-based, Loosely coupled, Distributed Systems. *Computer Languages, Systems and Structures - Elsevier*, 36(2):142–157, 2010.

[Tai93] Antero Taivalsaari. Object-oriented Programming with Modes. *Journal of Object-Oriented Programming*, 6(3):25–32, 1993.

[Tan08] Éric Tanter. Contextual Values. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2008. ACM.

[Tea11] Growl Team. Growl: A Notification System for Mac OS X. http://growl.info, 2004 - 2011.

[TGDB06] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-Aware Aspects. In Welf Löwe and Mario Südholt, editors, *5th International Symposium on Software Composition (SC 2006) Software Composition*, volume 4089 of *LNCS*, Vienna Autriche, 2006. Springer.

[VA01] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks.* PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[VCDDM07] Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Object-Oriented Coordination in Mobile Ad Hoc Networks. In Amy Murphy and Jan Vitek, editors, *Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*, pages 231–248. Springer Berlin / Heidelberg, 2007.

[VCVD09] Jorge Vallejos, Pascal Costanza, Tom Van Cutsem, and Wolfgang De Meuter. Reconciling Generic Functions with Actors. In *ACM SIGPLAN International Lisp Conference, Cambridge, Massachusetts*, 2009.

[VDM07] Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Object-Oriented Coordination in Mobile Ad Hoc Networks. In *COORDINATION*, 2007.

[VED$^+$07] Jorge Vallejos, Peter Ebraert, Brecht Desmet, Tom Van Cutsem, Stijn Mostinckx, and Pascal Costanza. The Context-Dependent Role Model. In J. Indulska and K. Raymond, editors, *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2007), Paphos, Cyprus*, LNCS 4531. Springer, 2007.

[VGC$^+$10] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated Generic Functions: Enabling Context-dependent Method Dispatch. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 66–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[VHC$^+$10] Jorge Vallejos, Jianyi Huang, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. A Programming Language Approach for Context-aware Mashups. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 4:1–4:5, New York, NY, USA, 2010. ACM.

[vLDN07] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented Programming: Beyond Layers. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 143–156, New York, NY, USA, 2007. ACM.

[VME$^+$07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. AmbientTalk: Object-Oriented Event-driven Programming in Mobile Ad hoc Networks. In *XXVI International Conference of the Chilean Computer Science Society (SCCC)*. IEEE Computer Society, 2007.

[VME$^+$11] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jorge Vallejos, and Jessie Dedecker. Ambient-Oriented Programming website. http://prog.vub.ac.be/amop, 2008-2011.

[Wal99] Jim Waldo. The Jini Architecture for Network-centric Computing. *Commun. ACM*, 42(7):76–82, 1999.

[Wei91] Mark Weiser. The Computer for the Twenty-first Century. *Scientific American*, pages 94–100, september 1991.

[Wik11a] Wikipedia. “Geolocation” — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Geolocation, April 2011.

[Wik11b] Wikipedia. “Notification System” — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/notification_system, June 2011.

[Wis05] John Wiseman. The CL-ZEROCONF Library. https://github.com/wiseman/cl-zeroconf, 2005.

[WWWK96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1996.

[YB05] Eiko Yoneki and Jean Bacon. Ubiquitous Computing: Challenges in Flexible Data Aggregation. In *EUC*, pages 1189–1200, 2005.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented Concurrent Programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.