Vrije Universiteit Brussel
Laboratorium voor Programmeerkunde
Faculteit Wetenschappen - Vakgroep Informatica

Move Considered Harmful:

A Language Design Approach to Mobility and Distribution for Open Networks

Wolfgang De Meuter

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

Promotor: Prof. Theo D'Hondt - Vrije Universiteit Brussel
Co-Promotor: Prof. Roel Wuyts - Université Libre de Bruxelles

A language that doesn't affect the way you think about programming, is not worth knowing (Alan Perlis, 1982)

Samenvatting (Summary in Dutch)

Het onderzoek voorgesteld in dit proefschrift handelt in de context van zogenoemde “Ambient Intelligence”, een recent gepostuleerde visie die voorziet dat mensen in de nabije toekomst omringd zullen zijn door een open Personal Area Network bestaande uit draagbare computers, GSM’s, muziekspelers, domoticasystemen, PDA’s, wagens met boordcomputer, huishoudapparatuur, enz. De basisdoelstelling is dan van al deze toestelletjes vlot te laten samenwerken op alle niveau’s, gaande van technische netwerkinfrastructuur tot en met de logica die huist in de toepassingen die erop draaien. Ons onderzoek gaat uit van de veronderstelling dat dit zeer moeilijk, zoniet onmogelijk, zal zijn m.b.v. de huidige generatie van gangbare programmeertalen, en dat een nieuwe stroming van programmeertalen nodig zal zijn die speciale voorzieningen hebben om met de dynamiciteit van zulke netwerken om te gaan.

In de plaats van het kind met het badwater weg te gooien, zijn we vertrokken van het object-georiënteerde model aangezien de notie van ingekapselde objecten haar nut in de context van distributie reeds bewezen heeft. We analyseren eerst welke de gevolgen zijn van de dynamiciteit en openheid van de bedoelde netwerken op de toepasbaarheid van gangbare klassegebaseerde en prototypegebaseerde object-georiënteerde talen. Onze analyse toont dat beide taalfamilies totaal ontoereikend zijn. Het dooddoend argument tegen klassegebaseerde talen is dat hun klassen een verborgen relatie spannen tussen de objecten die ze genereren. Deze relatie geeft aanleiding tot immense problemen als de objecten mobiel worden. Klassieke prototypegebaseerde talen, langs de andere kant, hebben te lijden onder inherente veiligheidsproblemen. Het probleem is dat ze verschillende taaloperatoren aanbieden die “meer” met objecten doen dan het sturen van berichten (bvb. kloning en overerving), maar dat deze operatoren dan ook meer vrijgeven van de interne (ingekapselde) staat van objecten. Dit is extreem problematisch voor de veiligheid in de context van open netwerken die bestaan uit mobiele gebruikers en dito software.

We stellen een “derde weg” voor welke een doorsnede is van klassegebaseerde en prototypegebaseerde talen in de zin dat het vele eigenschappen van klassen overneemt zonder ze te introduceren. Ons model is louter gebaseerd op objecten en berichten, een eigenschap die we *extreme inkapseling* noemen. Alhoewel men zou kunnen besluiten dat dit enkel tot oninteressante talen kan leiden, tonen we aan dat dit niet noodzakelijk het geval is. In ons model kunnen objecten speciale soorten van methoden bevatten, wier aanroep resulteert in het uitvoeren van de methode *nadat* een primaire actie zoals klonen, uitbreiden of verplaatsen over een netwerk werd ondernomen. Op deze manier kunnen “ordinary methods”, “cloning methods”, “view methods”, “mixin methods”, “move methods” en “visit methods” objecten de mogelijkheid geven tot het creëeren van kopies en uitbreidingen van zichzelf, verplaatsen van zichzelf, enz. Dit model geeft toegang tot zeer krachtige prototypegebaseerde technieken zonder veiligheidsproblemen. Ons onderzoek toont vervolgens eveneens aan dat het model makkelijk in lijn is te brengen met concurrent, gedistribueerd en mobiel programmeren.

Ons concurrentiemodel en distributiemodel is gebaseerd op actieve objecten

die met mekaar communiceren door het asynchroon versturen van berichten. De idee is dat berichten die bij objecten aankomen meteen in de wachtrij worden geplaatst en dat objecten slechts één bericht tegelijk kunnen verwerken. Mogelijke synchronisatie tussen zender en ontvanger van een bericht wordt geregeld door een systeem van transparante “promises”, die dienen als plaatsvervanger voor de waarde die de ontvanger uiteindelijk zal opleveren. Zowel de zender als de ontvanger zetten hun uitvoering concurrent verder. Pas als de zender de waarde van de plaatsvervanger nodig heeft, zal hij wachten op de ontvanger.

Dit model wordt vervolgens uitgebreid met object-gebaseerde overerving. De idee is dat een speciaal type van methoden (“active view methods”) actieve objecten kunnen genereren die het object waarin ze ondergebracht zijn uitvoeren. Het resultaat is dan een hiërarchie van actieve objecten. Om geen concurrentieproblemen te krijgen werd de zichtbaarheid van afstammelingen echter herleid tot nul. Speciale zichtbaarheidsfuncties laten echter toe dat de vader en zijn afstammelingen op gecontroleerde wijze code in mekaars context kunnen uitvoeren, en dit op atomaire wijze. Dat laatste garandeert dat er geen twee afstammelingen tegelijk met gedeelde staat kunnen werken zodat er geen consistentieproblemen opduiken.

Het model is ook de blauwdruk voor een distributiemodel vermits actieve objecten zich op verschillende machines kunnen bevinden, en geografisch verpreide objecten berichten met mekaar kunnen uitwisselen. Bovendien kunnen, door het feit dat afstammelingen gegenereerd worden door methoden, deze laatsten er voor zorgen dat de afstammeling terecht komt op de machine vanwaaruit die methode werd geactiveerd. Dit laat ons toe om object-hierarchieën te bouwen waarbij objecten en hun vader zich op verschillende toestellen kunnen bevinden. Dit model is een veralgemening van de proxy-idee die meestal in de context van distributie gebruikt wordt: in ons model zijn proxies niets anders dan lokale afstammelingen van objecten op een ander toestel. Maar ons model is expressiever aangezien proxies methoden lokaal kunnen overschrijven. Standaard proxies delegeren alles over het netwerk. Inderdaad, men kan een lokale afstammeling van een object op een andere machine zien als een applet die verbonden is met de server die het afgeleverd heeft. We noemen ze “geconnecteerde applets”.

Tenslotte hebben we dit model verder uitgebreid met sterke mobiliteit. Ons mobiliteitsmechanisme wordt naar voor geschoven als een eerste voorbeeld van wat we *gestructureerde mobiliteit* noemen. Dit is de tegenpool van de huidige mobiliteitsmechanismen die allen gebaseerd zijn op een *move* instructie die een object en een locatie verwacht en dat object boudweg op die locatie zal zetten. Ons pleidooi schakelt deze *move* instructie gelijk met de *goto* instructie uit de jaren zestig aangezien er met *move* geen zinnig woord valt af te leiden over de uiteindelijke locatie van objecten in het netwerk. Vandaar de nood aan gestructureerde mobiliteit. Ons voorbeeld hiervan bestaat uit zogenoemde “move methods”, een nieuw type van methoden die ervoor zorgen dat de uitvoerder ervan verplaatst wordt over het netwerk. De idee is dat de ontvanger van een bericht verplaatst wordt naar de locatie van de zender. We tonen het nut van dit mechanisme aan door de distillatie van een reeks elegante programmeeridiomen die hoog niveau mobiliteitspatronen belichamen.

Abstract

Our research context is Ambient Intelligence, a research vision that postulates that people will soon be surrounded by an open Personal Area Network consisting of laptops, mobile phones, music players, household equipment, domotic units, PDAs, cars etc. The idea is to make the devices cooperate smoothly at all levels,varying from the lower technical levels (i.e., networks) to the higher logical levels (i.e., applications). Our research starts from the conjecture that it will be hard, if not impossible to accomplish this using current mainstream languages. A new generation of programming languages will be needed that have special features to deal with the dynamics of such networks.

Instead of throwing away the baby with the bath water, we started out from object-orientation because the notions of encapsulated objects have already shown to be beneficial for distributed programming. The consequences of the dynamics and openness of the target networks are analyzed for both class-based and prototype-based languages. Our analysis reveals that none of these language families are adequate for application in our context. A knockdown argument for class-based languages is that classes impose an implicit relationship between the objects they create. This relationship poses enormous problems for mobile objects and mobile devices. Classical prototype-based languages on the other hand suffer from inherent security problems because they feature several operators on objects that do "more" with objects than just message passing (such as inheritance and cloning). They therefore also reveal more than just the interface of the objects. This is extremely problematic in the context of open networks with mobile users and computations because of security reasons.

We present a new "third paradigm" of object-orientation that is shown to intersect class-based and prototype-based languages in the sense that it adopts class-based characteristics without introducing classes. The model is based on objects and message passing alone, a property we call *extreme encapsulation*. Extremely encapsulated objects can contain special types of methods whose body is run after accomplishing some primary task such as cloning, extending or moving the object. This way, upon invoking "ordinary methods", "cloning methods", "view methods", "mixin methods", "move methods" and "visit methods" objects can spawn clones and extensions of themselves, move themselves and so on. This model allows one to have most of the powerful prototype-based features without suffering from their security drawbacks. Our research shows how this strictly encapsulated object model aligns well with concurrency, distribution and strong mobility.

The concurrency and distribution model is based on active objects that communicate through asynchronous message passing. Messages sent to an object are immediately queued and objects can process only one request at a time. For possible synchronization of the sender and the receiver, a transparent promise is returned that is a placeholder for a potential return value of the method. The sender and the receiver can concurrently continue their execution. However, upon trying to do something with the value represented by the promise, the sender will be blocked until the receiver is ready.

This model of active objects is subsequently extended with object inheritance. A special type of methods ("active view methods") can spawn active objects that extend the active object in which they execute. This results in an active object hierarchy. The philosophy of the model is to use shared parents to contain the state shared by two or more active objects. To make this practicable, the descendant objects cannot — in contrast to common practice — automatically and freely access the slots of their parent object. Scoping is strictly restricted by active object boundaries. Nevertheless, the super-this relationship *can* be used in a beneficial way: special scope functions allow descendants to execute code atomically in the context of their parent, rendering the scoping between a descendant and its parent highly controlled. Because the scoping functions guarantee atomicity, managing shared data without race conditions in a very simple way.

This concurrency model is also the blueprint for a simple distribution model because active objects can reside on different machines. Distribution transparency allows for remote method invocation by means of asynchronous message passing. Furthermore, because descendants are always created in response of a message send, the special methods that spawn descendants can create these descendants on the machine from where that message was sent. This allows for networked delegation structures because an active object and its parent can reside on a different machine. This hierarchical model of distribution is actually a generalisation of the proxy-notion that is heavily used in conventional approaches to distribution: in our model, proxies are nothing but networked descendants of objects that do not override or add any method, but forward everything (because of the networked delegation mechanism) to the object they represent. Our model is more expressive because proxies can locally override some of the slots. The model can also be seen as a generalisation of the applet model. A local object is created by sending a message to a remote object which spawns a descendant object on the machine on which the message was sent. This new object can be seen as a local applet that extends the server that spawned it. We call such applets "connected applets".

Finally the model is extended with strong mobility. This strong mobility mechanism is argued to be a first proposal of what we call *structured mobility*. Structured mobility is the antidote for today's strong mobility mechanisms which are all based on a *move* instruction that expects an object and a location and that puts the object on that location. We argue that this *move* instruction has practically the same implications as the *goto* instruction in the sixties. Analogously to the arguments against *goto*, we develop an argument against *move* and formulate the need for structural mobility mechanisms in order for human readers to be able to track the location of objects. Our model presents a first example of such a mechanism: by adding yet another type of methods (called "move methods") objects that receive a message can be moved around. The idea is that the object is moved towards the machine of the object that sent the message corresponding to a move method. We show the adequacy of the model by presenting a number of elegant programming idioms that allow for the construction of some extremely high level mobility patterns.

Acknowledgements

My deepest gratitude goes to my advisor Theo D'Hondt. Establishing an exhaustive list of what I have learnt from him would take several pages. I have deep respect for the way he manages the Programming Technology Lab and for the way he inspires young people to do research in areas that are off the beaten track. I thank him for his belief in me and for supporting me in all scientific capers I have cut during the past decade. Thanks Theo!

My co-promotor Roel Wuyts joined this project much later but still provided me with valuable comments. He pushed me to reshuffle the structure of the dissertation significantly which, in the end, improved the quality of the text drastically. Roel spent a lot of time meticulously proof-reading everything.

I thank the members of my jury, Prof. Eric Jul, Prof. Pierre Cointe, Prof. Dirk Vermeir, Prof. Viviane Jonckers and Prof. Bernard Manderick for comments on the first version of the text.

Jessie Dedecker, Tom Van Cutsem and Stijn Mostinckx were my partners in crime during the last year and have often played the role of a mirror upon which I could reflect my ideas. I look forward to continue working with them in the future. An exceptionally big thank you goes to Jessie for helping me with the implementation of my exotic language features.

I thank the other members of our lab for supporting me in the final stressful period and for enduring me at our weekly research meetings, especially at those moments when I knew that I was the only one in the room who was right (again!). They all deserve a thank you: Johan Brichau, Thomas Cleenewerck, Linda Dasseville, Dirk Deridder, Christian Devalez, Peter Ebraert, Johan Fabry, Sofie Goderis, Kris Gybels, Andy Kellens, Isabel Michiels, Adriaan Peeters, Lucas Stoops, Dirk Van Deun and Ellen Van Paesschen.

Special thanks to Dirk Deridder, Isabel Michiels and Linda Dasseville without whom I would have missed several paychecks. They managed the research projects on which we have been working in the past years. I also want to mention our secretaries Lydie Seghers, Brigitte Beyens and Simonne De Schrijver for taking care of my administration and for a friendly chat every now and then.

The lab's "old" generation, Carine Lucas, Wim Lybaert and Patrick Steyaert in particular, have shown me that participating in academic life is more than publishing papers. They have introduced me in the enigmatic world of scientific project management, industrial contacts and research proposal writing.

My friends Ellie D'Hondt, Maja D'Hondt, Sven Casteleyn, Johan Brepoels, Peter Verbrugghe and Yann Leus who were there for me during the toughest weeks of the writing period, entertaining me with both virtual and real-life chat, and taking me for beers whenever I got in overdrive and needed a break. Special thanks to Maja for transporting that bulk of breakfast cereals to my house that prevented me from starving in the final writing phase.

Last but not least, I am grateful to my mother Anni Nicol. Without her devotion and determination I would never have finished high school. In our socio-economic situation starting and completing university studies was all but trivial. I owe all of my current professional status to her. Thanks mum!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of new programming languages has arguably been one of the most driving factors in the progress made in six decades of computer science. Algorithms, patterns, software engineering principles and methodologies developed in over half a century have very often seen a new impulse right after the introduction of a new language. New languages have helped programmers express solutions in more abstract ways and have lifted academic solutions to levels that generated a need for technological, methodological and managerial support. But how and why do programming languages progress? Why do people and organisations decide to move from their familiar programming language to a new one? In other words, what are the parameters that determine the demand/supply curve that governs the "programming language market"? In his essay on Post-Javaism [Bla04], Andrew Black tries to come up with an answer by drawing a parallel between programming languages and 20th century architectural styles. His conclusion is that there are four factors that make or break new languages: technology, economics, function and fashion.

Unfortunately, there is not much computer scientists can change to the way economics and fashion determine the (non) acceptance of new languages. The software industry and tool supplier industry (e.g. the compiler industry) are certainly among the most important factors in the spread of new languages. Practically all "major" programming languages have been designed or at least heavily supported by corporate organisations: from Fortran to Java via Basic, Cobol, C and C++, all have been heavily supported by corporations such as Sun, AT&T, IBM and Microsoft. Some of these languages were indeed designed by academia but it is certainly not their academic quality (or lack thereof) that has been the reason for their worldwide acceptance. This economy factor, together with the fashion factor is something individual scientists cannot control.

But the aforementioned languages *do* recover features and characteristics from languages designed by academics. These scientific predecessors *are* shaped by academics and their development *is* purely influenced by technology and function. Indeed, technological developments such as the mouse, a hard disk, nearly free memory, the internet, garbage collection, peephole optimisers, just-in-time compilers, branch-prediction and v-tables have had a huge impact on language design. The programming languages we currently use hardly have anything in common with those that were the basis of software development in times of expensive batches intended for large mainframes with little memory, slow processors, modem-based network connections, and whose major input-output converters consisted of amber terminals, tapes and line printers.

The conjecture of this dissertation is that we are on the eve of a new technological and functional revolution and that this will have its impact on programming languages all over again.

A “processor count” in an average meeting room or lecture theatre shows us that pervasive computing and wireless network technologies are now swiftly penetrating our society, transforming visionary ideas postulated by artificial intelligence researchers into massively available technology in less than a decade. Devices such as domotic units, mobile phones, PDAs and digital music players are getting ever easily accessible at reasonable prices. The average home nowadays already contains over a dozen micro processors, both visible in the form of (miniaturised) computers, as well as hidden in consumer electronics and household equipment. Moreover, the advent of interconnection technology such as Bluetooth and the next generation of mobile phones will connect these devices, turning us into individuals that are constantly accompanied by a dynamically defined cloud of cooperating devices that interact with their environment at all times. This cloud is referred to as a Personal Area Network or PAN [Wik04]. At the time of writing, these developments are already generating new functions as well. It suffices to mention applications of GPS-technology, digital money and internet music stores that are marketed along with new hardware to see that this is just a glimpse of what we can expect in the following decades.

Writing applications for devices that participate in the kind of dynamically defined networks mentioned above will be easier said than done. Indeed, a few years of experience with some disappointingly simplistic applications already shows that the construction of software that supports (both user and software) mobility is far from easy to accomplish. It is not an exaggeration to say that constructing distributed and certainly mobile systems is currently nothing more than sheer handicraft. One of the reasons is that current day mobile applications are mainly built by trying to deploy technology that was originally not conceived for this purpose. This is often called the middleware solution. Middleware is a layer of software that was written in an “ordinary” programming language following “ordinary” software engineering methodologies and the goal of which it is to map the concepts needed in the construction of distributed and mobile applications onto concepts existing in mainstream programming and software engineering technologies. Unfortunately, this turns out to be about as easy as trying to express the subtleties of chinese opera in terms of primitive percussion chords. It turns out that the dynamics arising in mobile computing is extremely hard to realise and keep track of in conventional programming languages. Everyone who actually ever built even a small application that is partitioned on a (statically defined) network topology will confirm this.

While recently a lot of effort has been spent on technology to support mobility and distribution in the context of middleware research, the status of programming languages is deplorable, especially in the field of mobility. And even in the case of distribution, most existing languages focus on networks whose topology is known in advance. We practically have no knowledge about programming language features and programming techniques (algorithms, patterns,...) to conquer the task of implementing clouds of processors whose identity is not known upfront and whose relative geographical position might change during the execution of a given program.

1.1 Research Context & Motivation

Before we delve into the actual technical problem addressed by our research, we want to clarify our vision of the future by giving a small scenario. This will help us to formulate our research context in section 1.1.2 and to formulate our technical goals in section 1.3 clearly. The scenario is presented in section 1.1.1. Although it may seem a bit far-fetched at first, our experience with undergraduate students taught us that it sets the right frame of mind to get a gist of the kind of technology we are after. The scenario is a variant of the "Ambient Intelligence" vision recently put forward as one of the strategic research directions in software engineering by the European Council's IST Advisory Group [IST03].

1.1.1 A Futuristic Scenario: The Kitchen without Buttons

Harry is single. His friend Gina is visiting him. They plan to prepare dinner together for old times' sake. Harry has an ultra modern kitchen of the famous ZanuKnecht brand with all amenities: a gammaray oven, cooker, coffee machine, blender, an old microwave oven, fridge, freezer and a cooker hood. Needless to say, all equipment is 100 per cent free of buttons. All operation is done by means of the latest handheld Gizmo model Harry always has at hand. These days, such a device is as common as the good old European mobiles 20 years ago. Gina also has one, albeit slightly obsolete, but it does what it is supposed to do because it is good enough to run Utopia2 scripts. Gina's old Baunussi kitchen is completely different. But both run their own Utopia2 scripts. Harry those of Zanuknecht. Gina the ones released by Baunussi.

They proceed. While Harry is chopping vegetables Gina sets about preparing the broth. Harry sends her (via the Gizmo2Gizmo-standard) his cooker objects (kitchen-click-control-click-share-click-address book-Gina-click) and off she goes. Although her Gizmo still works with old fashioned pixel technology, she is perfectly able to use these objects because of the Gizmo-classic-pixels-converter she installed. It ensures a best-possible projection onto pixel displays of those modern liquid morph interfaces everyone is running these days.

While Harry is busy cutting veggies, Gina asks him from within the kitchen if he is willing to run over to WallSmart to get some butter. Normally, Harry always does his shopping at 7-isnot-11 around the corner, but Gina insists on the "Le Chat Qui Pleure" brand that is being sold exclusively at WallSmart. So he has to upload the gamma of WallSmart into his Gizmo. He points the Gizmo to the Gizmo-dock next to the cable-tv-plug and selects www.wallsmart.supermarkets, clicks "dairy produce" and sets off. While walking on the pavement, Harry runs through the dairy produce assortment of WallSmart. After a while he finds Le Chat Qui Pleure butter and buys a pack. The i-ticket on the Gizmo screen notifies him that the pack is indeed bought and that the two euros are withdrawn from his bank account. Phone! Gina phones to ask whether he can send her his cooker-hood-objects. The broth is boiling and the kitchen windows are steamed up. Kitchen-click-cooker hood-click-share-click-click-ok. He quickly sends her his apartment-environment-objects in the same way, for otherwise, Gina wouldn't even be able to switch on the radio or turn on the light. But before doing so, he excludes the safety-box-objects and the body-objects. After all, what he has seen on the scales this morning is none of her business. House-click-domotics-click-share-click-excludefrom-click-sca... scales-click-saf... safety box-click-click-click. Upon arrival at WallSmart, he beams the i-ticket to the door. The Gizmo displays "counter 5". He walks to the counter, says hello to the friendly lady, beams her the i-ticket and receives his butter. Of course, the i-ticket is now erased from the Gizmo. If not he could be eternally collecting butter with the same ticket. Back home, the broth is simmering and the cooker hood is humming softly. Suddenly, Harry wonders what those blinking red letters are doing on the microwave's display. Gina replies that she downloaded the new Utopia2 software for the device because Harry's scripts already missed two new releases. A little agonized by the freedom she permitted herself, Harry thinks that he should have exluded the script-installation-objects from the apartment-environment-objects he just sent her. Nevertheless, he points his Gizmo towards the microwave to get the new controllers on his Gizmo as well. The handshake between the devices also automatically notifies the recipes known by the kitchen such that they will henceforth work together with the updated microwave.

Let's stop here! Imagine this being programmed in a mainstream object-oriented programming language. Which objects reside on which machine? Which are shared? What is replicated? How is the i-ticket's integrity ensured? It is clear that it will not be an easy task to express this kind of complexity in today's mainstream procedural and object-oriented languages. It is our conjecture that this kind of applications will require a new generation or programming languages that have built in provisions to deal with this kind of complexity.

1.1.2 PAN - Personal Area Networks

In brief, the idea of Ambient Intelligence, or AmI for short, is that every individual will be (or better: is) surrounded by a dynamically shaped processor cloud consisting of devices such as digital cameras, mp3 players, cellular phones, PDAs, laptops, microchip-enabled credit cards and wearable hardware (e.g. clothes with built-in chips), *and*, that these devices will intelligently and smoothly cooperate with each other and with the processors that unexpectedly enter the cloud as a direct or indirect consequence of the fact that people and devices move around. Examples of this phenomenon might include the spontaneous reaction of a domotics control system when one arrives at home, a car that wakes up as one approaches it, traffic control sensors that one encounters while driving, public amenities such as a railway station that beams its train schedule to whomever approaches it, advertisements and special offers as one moves about in a supermarket and so on. Even with the current state of hardware developments these applications are not far-fetched. A lot of the consumer electronics that we see around us today are already equipped with wireless network facilities such as WiFi, Bluetooth or infrared. Furthermore, the fact that all but lightweight technology such as JVMs with their garbage collector and fairly extended collection of standard libraries is currently running complex middleware on such devices shows that it gets less and less utopian to think of these devices as "real" computers that will be programmed in high-level languages. However, as argued by the following section, putting a JVM on these devices is not enough!

1.1.3 Why "Java" is Not Enough

We have deliberately put the name "Java" between quotes in the title of this section because we want to stress that this section is not just about Java but about the role of current mainstream languages in the evolution described above. Nevertheless we have also deliberately chosen the name "Java" because many people really seem to believe that Java is the final language computing science will ever produce. Such a belief is not a new phenomenon in the history of computer science. But what *is* new is that this time the belief is not restricted to the ones in charge of economics and fashion. As time moves on, more and more universities change their programmes into Java schools and renowned conferences are ever more overwhelmed by Java related submissions. Not surprisingly, research currently being conducted in the field of Ambient Intelligence is very often formulated in Java terms.

It is one of the very explicit assumptions of our work that, in the long run, the impact of Java on scientific progress in AmI will be limited. As the AmI application demands will increase, the limitations of Java and its associated technology will act as a drag and hamper new innovations from happening. Of course, this is a claim that is hard to prove. Nevertheless, we have very good reasons for the assumption:

- Even aside from AmI requirements, Java is pretty rudimentary as an object-oriented language, a claim we will extensively defend. For example, it does not feature constructs as basic as closures, but tries to re-introduce them through the cumbersome anonymous classes proposal with its extremely limiting technical restrictions.

- Java is a statically typed language with a fairly simplistic type system. It is getting clearer and clearer by the experience with currently popular middleware solutions that static typing and distribution are not each others best friends. Distributed code written in Java is generally full of type casts and many subtle errors such as the problem of non-interchangeability of objects and their proxies are type-related. We claim that this will get worse in software that has to run in open networks, i.e., networks the number of nodes of which and the acquaintance relation is not predefined.

- Java has huge security problems unless one is willing to stick to the sandbox model adhered to by applets. This model is pretty useless for realistic applications because an applet can basically do nothing but communicate with the server that delivered it.

- Java has no mobility provisions. Although there exist middleware solutions that offer limited forms of mobility, Java is essentially stack-based and has no provisions to manipulate that stack explicitly. As such, it is extremely hard to implement truly mobile applications in Java.

- Java has no provisions to deal with unanticipated situations such as partial failure of a network. The only language constructs it offers to deal with such situations are selection ("if") and exception handling ("try-catch"). These are fairly primitive and often result in polling techniques. Furthermore, they pollute the code with instructions that have nothing to do with the actual application logic but with the uncertainties of the hardware constellation.

- Java does not allow objects to change dynamically whenever they end up in situations that might require such a change. This is because Java objects belong to a class and the object-class relationship is fixed at object-creation time for the entire lifetime of the object.

- Java has a very simplistic concurrency model and the distributed and autonomous nature of the hardware described above requires concurrency to be the norm in AmI applications instead of the exception.

- In contrast to what many people think, Java has extremely poor distribution facilities. We will extensively get back to this in chapter 7.

Of course, there are ample possibilities to work around these shortcomings in specific situations. E.g. there exist experiments that preprocess Java code in order to transform every method in such a way that it explicitly copies its own stack frame to an object allocated in the heap [TRV$^+$00a]. As such it gets possible to implement mobility of running Java code. However, such solutions are often cumbersome to work with and usually do not offer a complete solution that is conceptually simple and consistent. Moreover, having such a "tool based" solution to *all* of the problems discussed above seems simply unmanageable in practice. What we need is an entirely new generation of programming languages that offer an integrated solution.

1.2 Language Design Research Criteria

Since this dissertation is about language design, we should say something about this topic in general. It is very difficult to come up with clear cut criteria to evaluate programming languages scientifically. Ever since Alonzo Church postulated his famous thesis, all languages are considered equally powerful and hence it is very hard to "prove" that one programming language is "better" than another one. However, this is also known as the Turing Tar-pit: in theory all languages are equally powerful but often, the more one struggles to get some real work done in them, the deeper its technical inadequacies suck one in. Of course, *expressivity* is the keyword here and this is extremely hard, if not impossible, to measure in a scientifically justifiable way.

1.2.1 Measuring Expressivity

Although there is no litmus test for expressivity, there *are* ways to study programming languages. Roughly spoken, we can divide them into two camps:

- The *pragmatic camp* studies expressivity of programming languages purely from a programmer's perspective. The main issue here is how a language can improve the way programmers think about their problems and structure their systems. This is often the perspective extolled by engineers. Often, they consider a language expressive if it offers a fan of features a programmer can choose from to tackle his problems.
- The *language theoretical camp* studies programming languages from a conceptual angle. Issues such as orthogonality, semantic simplicity and regularity are the main concerns. Although most of the comparative studies in the language-theoretical camp are mathematical in nature (such as denotational semantics [Sch86]) less formal, yet conceptual, comparisons have been made too. An outstanding example is [Mac87].

Many languages result from an excessive adherence to one of both camps. C++ is a good example of language design in which the pragmatic point of view has prevailed. Formal studies of full-fledged C++ are nonexistent to the best of our knowledge. Examples of the other extreme are purely functional languages which are theoretically very well understood but are considered to be not practicable by most practitioners. In our research, we were strongly inspired by languages such as Scheme and Smalltalk because we think both Scheme and Smalltalk are excellent combinations of both points of view. On the one hand, they are conceptually well-understood and have relatively readable formal descriptions [KCE98] [CP89]. On the other hand, they were designed with sufficient pragmatism in mind such that they can be used to write realistic software.

1.2.2 Language Design vs. Language Implementation

In our conception of language features, we have supported the vision that programming language design and programming language implementation are two different fields of computer science and should — in principle — not influence each other too much. If language designers in the sixties and seventies had allowed the hardware and implementation techniques of those days to influence their proposals, then surely Smalltalk, Lisp, Prolog and many others would never have been invented. All these languages contain features that had inacceptable performance penalties at that time. Nevertheless, their designers have persevered in their goal without paying too much attention to speed and memory consumption. Later on, hardware improvements have made these languages much more acceptable. Furthermore, the controversiality of the features itself has resulted in implementation techniques some of which currently live on in mainstream language processors such as the JVM. Although Prolog's Warren Abstract Machine did not find its way to the mainstream (yet), techniques such as v-tables, selector table indexing and branch prediction for Smalltalk and lambda lifting and garbage collection for Lisp did.

Of course, this does not mean that language designers can design just about anything they want. Some features are inherently inefficient and one has to be vigilant not to step into this trap when designing languages separately from implementation level considerations. But having said this, it is our explicit goal to keep language design and language implementation apart as long as possible. As such, this dissertation proposes some features that have only been provided with inefficient prototype implementations. This does not mean that we do not consider those issues important. They are just the topic of another dissertation.

1.2.3 The MIT Approach vs. New Jersey Approach

As C.A.R. Hoare describes in his famous text *Hints on Programming Language Design* [Hoa73], there exists a huge difference between designing language *features* and designing *languages*. Hoare calls the former innovation and the latter integration and explains that designing a language is much more difficult than designing a language feature. So many years later, it has become clear that Hoare was painfully right. While the world is full of well-designed language features, only very few well-designed languages exist. Most languages are a dreadful result of feature piling. Good examples of well-integrated languages are Algol, Scheme and Smalltalk. Examples of feature piling are Cobol, PL1 and C++. Java started out as a reasonable example of integration but is with each new release degenerating more and more into a pile of features.

The difference between results of integration and results of feature piling is a reflection of the main activity of the language designer. Whereas designers of integrated languages are constantly *unifying* (i.e., *intersecting*) features, designers of feature piles are constantly *uniting* features. This activity is usually driven by "new programming examples" that cannot be expressed in an existing language. Upon encountering such examples, the main reflex of the intersectional school is to redesign existing features to make them cover the new examples. The main design activity is reconsideration, integration, polishing and generalisation. The reflex of those belonging to the uniting school is to add a new feature that covers the intended examples. Very often this results in unexpected interactions between the newly added features and the existing ones.

These diametrically opposed visions on programming language design have been well-characterised by Richard Gabriel in his famous “Worse is Better” paper series [Gab94]. In this series, Gabriel contrasts the so-called MIT/Stanford-approach in language design with the New Jersey approach. He characterises them as follows (cited literally):
The **MIT Approach** strives for The Right Thing:

- Simplicity – the design must be simple, both in implementation and interface. It is more important for the language to be simple than the implementation.
- Correctness – the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency – the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness – the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

The **New Jersey** philosophy is mainly driven by pragmatics:

- Simplicity – the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the language. Simplicity is the most important consideration in a design.
- Correctness – the design must be correct in all observable aspects. It is slightly better to be simple than correct.
- Consistency – the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
- Completeness – the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

After years of reconsideration, Gabriel is still doubtful about what is the best philosophy to follow. Having published follow-up papers entitled “Worse Is Better Is Worse”, “Is Worse Really Better?”, “Back to the Future: Is Worse (Still) Better?” and “Back to the Future: Worse (Still) is Better!”, he finally gave up and writes “Decide for yourselves.”[Gab].

In this dissertation we have mainly followed the MIT-approach. This is consistent with the fact that we decided to disconnect language design from language implementation as explained in section 1.2.2. It also aligns very well with our opinion about finding the right balance between the theoretical and the pragmatic camp we have presented in section 1.2.1, and with our application of Occam's Razor reviewed below.

1.2.4 Occam's Razor

Apart from considerations specific to computing science, the general scientific principle called *Occam's Razor* has been adhered to as much as possible. In general, this principle states that when there are two explanations for the same phenomenon, then the explanation which uses the smallest number of assumptions and concepts must be the right one. Literally the principle states that *one should not increase, beyond what is necessary, the number of entities required to explain anything.*

We have always kept to this view on research in the sense that we have always preferred a small number of conceptually simple (yet general) language features over huge numbers of features that finally result in big programming languages. Whenever "example programs" could not be expressed in our model, we have tried to reshape, generalize and polish the model instead of adding new features to cover the examples.

1.3 Problem Statements

The work presented in this dissertation does not claim to be a complete solution to the AmI problems addressed in section 1.1.3. But it does make some contributions to the field. The main contribution of the dissertation is that it presents a consistent, relatively small, yet very powerful language in the MIT-camp that addresses four fundamental problems that will *have to be solved* by a new generation of programming languages targeted at Ambient Intelligence:

Problem #1: The Ambient Object Paradigm Problem.

The first problem is a seemingly unresolvable stalemate. We will argue that it will be **very hard if not impossible to program such systems using class-based object-oriented programming languages**. We will present a thorough analysis that will show that class-based object-oriented programming languages induce a number of fundamental paradigmatic problems when being deployed in the context of open distributed, and especially, in mobile systems. We will argue this in great detail in chapter 3. This analysis in itself is one of the contributions of this dissertation. Part of the solution to this problem will be the promotion of prototype-based languages. However, as we will show in section 3.4, existing **prototype-based languages have a number of encapsulation problems that are unacceptable in the context of mobile and distributed programming for open networks** which is disastrous for security. Expressed in a nutshell, the current suite of prototype-based programming languages make it impossible to implement the i-ticket of the scenario. Therefore, a large part of our research was about redesigning prototypes-based languages in order to render them applicable in the AmI context, without having to resort to classes.

Problem #2: The Concurrent Parent Sharing Problem.

A second fundamental problem is a fundamental mismatch between **sharing of state and code as extolled by prototype-based object-oriented programming** paradigms on the one hand, and the obsessive attempts to **avoid sharing in concurrency models** for object-oriented programming languages on the other hand. The latter is due to the fact that state sharing in general does not combine very well with concurrency because of a phenomenon called "race conditions"; situations in which the semantics of two concurrent processes that manipulate shared resources depends on whichever process wins the race. Such problems have made designers of concurrent object-oriented programming languages shun state sharing as much as possible by resorting to systems in which objects solely communicate by message passing. However, in the context of pure prototype-based programming this nearly automatically leads to actor systems which have extremely nice scientific properties but which do not seem to scale up precisely because of their lack of sharing mechanisms. We will show in chapter 8 that there *is* a way out of this situation. By cleverly aligning parent sharing as advocated by prototype-based programming with state sharing between concurrently running objects, a "third paradigm" is possible that seems to combine the best of two worlds.

Problem #3: The Distributed Sharing Problem.

The third fundamental problem has to do with sharing as well and is strongly related to the second one. It boils down to the fact that whereas conceptually, distributed systems are all about **structural sharing**, current distributed programming languages **merely offer referential sharing**. This is best explained on the basis of a virtual white board, i.e., a big sheet of distributed paper onto which different cooperating parties can draw simultaneously. From a modeling point of view, the sheet of paper really *is* on all those machines at the same time. This is in sharp contrast to the technical provisions offered by programming languages. Although distributed languages come in several versions, most of them only offer objects the ability to refer to objects on different machines and not to be or **be part of an object on a different machine**. We will make an initial proposal in this direction in chapter 8. In alignment with the solution put forward to tackle the second problem, we will exploit the parent sharing technique of prototype-based programming languages to offer programmers objects that are truly distributed on different nodes of a network.

Problem #4: Move Considered Harmful.

A fourth fundamental problem is that currently **existing language abstractions for mobile systems inevitably lead to "unmanageable object soups"** whose relative locations are totally unpredictable, even by human readers of the code that have generated this object graph. As we will motivate in chapter 9, the basic problem is that current day mobile languages introduce mobility by means of a "move" instruction that explicitly takes the mobile object and its destination node as parameters. This instruction, combined with standard object-oriented techniques such as message passing and double dispatch will easily lead to programs human readers of which can no longer predict which object is residing where. We will argue that this “move” instruction therefore relegates **current day mobile programming languages** to the same level in software engineering as **languages with “goto”** at the end of the sixties. We will formulate the need for languages supporting “structured mobility” and make an initial technical proposal.

1.4 Contributions

The goal of our research is to develop a high level language prototype that has features dedicated to deal with the dynamics induced by the sort of networks described above. Needless to say, not all aspects of such a language can be developed within the context of one PhD dissertation, a statement that is endorsed by the size of our future work section. Nevertheless this dissertation presents a language prototype which we claim to be a step in the right direction. The language is a full-fledged prototype-based object-oriented programming language with powerful features like delegation-based inheritance, cloning, first-class methods, active objects, synchronous and asynchronous message passing, distributed objects, distributed inheritance and last but not least an initial proposal for structured mobility. The language, called ChitChat, is an **integration of three fairly independent research tracks** in the field of object-oriented programming language design:

1. Although ChitChat has characteristics of class-based languages, it inherits the powerful features of **prototype-based languages** such as Self without adopting their insecurity due to their flexibility. Some of ChitChat’s characteristics as a prototype-based language are innovative on their own, notably its provision of first-class methods.

2. It inherits much of the concurrency model of ABCL, one of the most advanced **concurrent object-oriented languages** in existence. Active and passive objects are combined and a powerful asynchronous message passing is adopted.

3. It inherits distribution and strong mobility provisions from **distributed and mobile languages** as advanced as Obliq, Argus and Emerald. However, it has a very innovative vision on distribution to wit the idea that parts of objects can reside on different machines and share attributes by having two objects sharing the same part. This is mapped onto networked delegation with parent sharing.

ChitChat is the result of a language design exercise that is both interesting as an *integration* of earlier language features and as a suite of innovative language *features*. Its **most important contributions are**:

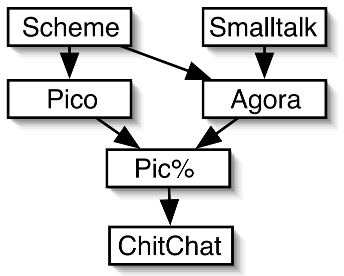


Figure 1.1: The Language Tree of the Dissertation

1. It is a marriage of class-based and prototype-based programming languages. It combines the advantages but avoids their disadvantages.

2. It proposes first-class methods and thereby intersects lambda-based languages such as Scheme and prototype-based languages like Self.

3. It proposes a delegation-based model for concurrency and distribution. Shared parents are applied as the way to share mutable state between distributed and/or concurrently running objects.

4. It proposes a set of language features that enable mobile objects in a structured way, i.e., without resorting to an explicit *move* instruction that results in uncontrollable distributed object-soups.

The research leading to these contributions was conducted at the Programming Technology Lab (PROG) of the Vrije Universiteit Brussel. This Lab has quite a rich history in dynamic (object-oriented) programming languages. Ever since the early eighties, members of the lab have been actively involved in the design, use, implementation and teaching of such languages. PROG was one of the driving forces of introducing MIT's Scheme-based Abelson&Sussman freshmen course at the Vrije Universiteit Brussel and was arguably one of the very first promotors of object-orientation — in the incarnation of Smalltalk — in Belgium. The work presented here is in many aspects a consolidation of that history. As shown in figure 1.1, the ChitChat language presented in this dissertation is a direct intellectual descendant of Agora and Pico, two languages previously designed by former and current members of PROG. Since research at PROG is never a strictly individual matter we have also contributed to (the comprehension of) these languages before consolidating that knowledge into a new research artifact. Hence, instead of bluntly introducing ChitChat we have decided to present the contributions made to these projects as well. They form an inseparable part of our work.

1.5 Limitations

As already mentioned in section 1.3, ChitChat is not a complete programming language that solves all AmI problems. Its most important limitations are:

- ChitChat does not deal with exception handling. Although most modern programming languages feature an exception handling mechanism, some basic assumptions of such mechanisms no longer hold in an AmI setup. The main problem is that (because of the independence of devices) the code that detects an exceptional situation and the code that is able to deal with it might no longer be connected or might be in a different execution context because of asynchronous communication. Existing exception handling schemes will probably need to be redesigned.
- ChitChat was merely given a prototype implementation in which network connections where simulated and in which only very little effort was spent on performance issues, both from an execution speed as well as from a memory consumption point of view. Furthermore, no studies where undertaken to measure the impact of an actual unreliable network on the semantics of ChitChat's language constructions.
- Although the ChitChat research spent a lot of effort on how to deal with new objects that unexpectedly appear in the network, *no* effort was spent on objects that disappear from the network and the consequences this has on the rest of the system. This instance of the so-called partial failure problem was beyond the scope of our research. However, as mentioned in the future work section, we are not avoiding the problem. It is one of our current research topics.

1.6 Roadmap

Although ChitChat surely has some rough edges left and needs more polishing here and there, we consider it as a fairly good language design attempt with respect to the language design criteria set out in section 1.2.

An inconvenience that all such languages seem to have in common is that they are hard to explain because they are based on a very small number of fundamental concepts that can be combined endlessly. Moreover, continuous integration has related these concepts into a sophisticated network with subtle and deep interactions. This phenomenon also renders languages like Scheme and Smalltalk hard to master fully. We have nevertheless tried to decompose our proposal into independently digestible principles and have structured the dissertation accordingly.

- We already indicated that ChitChat is a prototype-based object-oriented programming language. Chapter 2 therefore reviews some of the properties and characteristics of prototype-based languages in general and discusses a number of important languages of this family.

- Chapter 3 is the main motivation for our work. We show that class-based languages have a number of inherent paradigmatic shortcomings (that are due to the very class concept) for being deployed in the context of open networks. We also show that classical prototype-based languages have inherent problems in the context of encapsulation and security. From this analysis we will formulate the need for a third way — a classless language family that does not suffer from the security problems prototype-based languages suffer from. Three important criteria, to wit *extreme encapsulation*, *reflection protection* and *Granovetter-connectivity*, will be postulated as yardsticks against which we can evaluate the languages discussed in the dissertation.

- Based on a comparison between the denotational semantics of class-based languages and that of prototype-based languages, chapter 4 proposes such a third paradigm of object-orientation that inherits the advantages but avoids the drawbacks of classes and classical prototypes. We present the Agora model put forward by Steyaert's PhD dissertation [Ste94] as the technical incarnation of this model. Agora will be shown to be a powerful object-oriented programming language that does not contain classes yet endorses the extreme encapsulation and reflection protection principles. However, although the Agora model will be shown to have the right language theoretical properties, Agora has many problems when it comes to practical applicability.

- To overcome Agora's problems, in chapter 5 we will take a step back and derive Pic%. Pic% is an object-oriented programming language that is based on Pico, our homeground Scheme derivative the keyword of which is *small*. Pic% will be shown to incorporate Agora's characteristics and thus be a scion of the "third family" we propose. But Pic% is much simpler, avoids Agora's drawbacks and is in some respects even more expressive than Agora. Two particularly innovative language features of Pico and Pic% are a new form of parameter passing and first class environments.

- Both Pic% and Agora have been around for quite some time in the form of more or less experimental implementations. This has made it possible to expose other members of PROG and a few generations of graduate and undergraduate students to the languages. The result of this small user committee is the distillation of some programming techniques and idioms some of which are presented in chapter 6.

- Chapter 7 will review the most important literature on concurrency and distribution in object-oriented programming languages. We will review several radically different approaches to concurrency and distribution varying from the application of libraries, over reflection to dedicated programming languages. Apart from reviewing the literature this chapter will also evaluate existing approaches in our context of open networks.

- Based on the knowledge of chapter 7, the problem statements of chapter 3 and the intermediate Pic% language presented in chapter 5, chapter 8 will present, ChitChat, our approach to concurrency and distribution. ChitChat will be an extension (and re-iteration) of Pic% with concurrency and distribution features. The main characteristic of ChitChat is that it extolls networked delegation, i.e., the ability for objects to extend objects that reside on other machines. Since two such objects can share the same object, this gives rise to networked parent sharing. It is exactly this kind of parent sharing that will be used to have a controlled form of state sharing between concurrently running objects. Parent sharing will also be used as a way to physically share the state that is conceptually shared by distributed objects.

- Chapter 9 extends the ChitChat model of chapter 8 with features for strong mobility. The chapter extensively reviews different kinds and causes of mobility and gives an overview of existing language proposals for code mobility. Subsequently, a suite of desired properties for mobile programming languages is distilled by conducting a number of gedankenexperiments and by considering the restrictions of the open networks we target. This will allow us to formulate one of the main theses defended by this dissertation: a "*move*" instruction in mobile languages is a harmful feature. Based on this thesis, the design of ChitChat's strong mobility is presented and a collection of mobile programming patterns is presented.

- Chapter 10 presents our conclusions, related work and formulates some future topics of investigation spawned by our research.

Chapter 2

Prototype-based Languages

As explained in the introduction, one of the keys to our solution to the problems outlined in section 1.3 is the promotion of prototype-based programming languages in the context of distribution and mobility. This will be extensively argued in chapter 3. We will now give a general overview of prototype-based languages, review their characteristics and have a closer look at three important case studies that have put prototype-based programming languages on the map. In order to have a solid frame of reference, section 2.5 presents a general language theoretical framework that is often used to contrast prototype-based programming languages with their class-based analogues.

2.1 Introduction

In contrast to what most people think, object-oriented programming is not necessarily about writing programs using classes and inheritance. Classes and inheritance constitute but one branch in the object-oriented programming language taxonomy. The languages in this branch are known as *class-based languages*. An entirely different branch is the branch of so called *prototype-based languages*. In these languages, programs are written in an object-centered way. This means that instead of designing classes and building class hierarchies using inheritance, the main activity of a programmer consists of building concrete objects and specifying the relations between objects.

Although prototype-based languages are far less well-known than class-based ones, from a purely scientific point of view, we can safely say that the space of object-oriented programming languages is fairly neatly divided into class-based ones and prototype-based ones. Indeed, if we count the number of prototype-based languages and have a look at the richness of the set of features described in the context of prototype-based languages, they can safely be considered to constitute at least half about what is currently known about object-oriented programming languages. Contemplating the field from a certain distance, this binary division of the object-oriented programming language design space is not really surprising. They simply emerge from two radically different ways to model the world around us. Roughly spoken, prototype-based languages emerge from frame-based languages that AI used for knowledge representation. These frame-based languages were particularly well-suited to formalize concept graphs which consist of concepts linked together by relational connections. Class-based languages too emerged from the need to formalize concepts. Indeed, the first class-based language, Simula67, was designed to model objects and their movements in the physical world. But although frame-based languages put the emphasis on "real" objects, Simula required the programmers to make an abstract specification of the set of all the objects to be modeled (i.e., the class). As we will see in the following section this is a technical reflection of two radically different ways of thinking that philosophers have put forward to describe the world we live in. This is the philosophical history of prototype-based languages. The bridge between these two branches of object-oriented programming languages was made by the influential paper of Henry Lieberman [Lie86] who introduced the work done in cognitive sciences (i.e., prototypes) into the object-oriented programming languages research community (that was dominated by the class-based way of thinking). From that point onwards, the language theoretical perspective took over in the history of prototype-based object-oriented languages. From this perspective the development of prototype-based languages can be explained as the result of "language simplification efforts" undertaken by researchers in object-oriented programming languages.

2.2 History Of Prototype-based Languages

A language theoretical perspective of the how-and-why of prototypes is deferred to the following chapter because it requires a deeper understanding of the underlying semantics of both class-based and prototype-based languages. In this section, we briefly summarize the philosophical origin of prototype-based languages. Prototype-based language originally emerge from the field of knowledge representation in AI (mainly developed by Minsky), a field which, in its turn, owes a lot to late twentieth century philosophers like Wittgenstein. The bulk of the material presented here is a summary of the excellent article written by Antero Tailvalsaari in [Tai98].

The idea of dividing the world around us into "objects" is almost as old as science itself. Ancient Greek philosophers like Plato and Aristotle classified the things around them into "ideal ideas" and "instances of these ideas". The ideas were some form of idealized "template" desription of the instances. New ideas were formulated by describing the "genus" plus "differentia", i.e., the basis of the idea (another idea, that is) and the differences of the new idea with respect to the basic idea. Through the ears of a computer scientist this all sounds extremely familiar. Ideas are classes, instances are objects and the mechanism to describe new ideas based on old ones is known as inheritance.

This framework of classifying the world into classes and objects withstood the test of time for more than two thousand years. It is only in the late nineteenth and in the course of the twentieth century that philosophers seriously started questioning them. In [Tai98] many arguments are listed illustrating that this way of describing the world is probably not the right one. The most important criticisms, for our purposes, against the Greek school are:

- There does not exist a unique objective classification of the world because there is that there is no general algorithm to determine the properties upon which such a classification ought to be done. Deciding which properties are characteristic for a particular class of objects is a human process that uses a lot of interpretation. For instance, many people will not agree to call Conway's Game of Life a game, simply because it does not involve players. But others will, simply because it is fun to play with.

- For many classes of objects, humans consider some objects to be "better members" of the class than others. Indeed, probably everyone except mathematicians will consider zero, one and two to be "better numbers" than 0x13FEED2. So there is more to it than "just" ideas and instances. Apparently, some instances are "better" instances than others.

- Many classes that we use on a daily basis do not even have clear boundaries. For instance, how does one define "a work of art"?

The theory of prototypes was invented as a solution to these deficiencies of the Aristotelian view of the world. The most prominent philosopher to defend this theory of prototypes was Wittgenstein. The general idea of prototype-based thinking is to tie the world of objects together according to "resemblance" links: some objects are named "prototypes" and other objects "resemble" these prototypes. They "point to" the prototype but also list their differences with respect to the prototypes they point to. As such the world gets modeled as a network of representative prototypes and similarity links.

Just as the Aristotelean way of thinking can be considered the philosophical basis for class-based languages, the prototype theory of Wittgenstein can be considered as the fundamental basis of prototype-based programming languages. As we will see, in prototype-based languages, all programming activity consists of direct manipulation of concrete objects. Classification consists of linking prototypes together with one or more sharing links to exisiting prototypes. This view of organising systems with prototypes that share knowledge is largely the work of AI researchers such as Minsky, Winograd and Goldstein who designed knowledge representation mechanisms based on Wittgenstein's prototype theory in the mid to late seventies. The basic idea of their models was to declare knowledge frames (sort of like "a struct"), and to construct networks of knowledge by linking these frames together with all kinds of meaningful relations. A second important step to get prototypes into the realm of programming languages was made by Lieberman in the early eighties with Act1. He introduced the prototype-based way of thinking into OO research with his influential paper at OOPSLA'96 [Lie86]. As indicated before, from that point onwards, the language theoretical history of prototype-based languages took over.

The verb “take over” in the previous sentence was deliberately chosen. Whereas Lieberman proposed prototype-based languages as a new way to think about the organisation of systems, almost everything that was subsequently published about prototypes was about the design and technical properties of prototype-based programming languages from a language design perspective, and no longer about designing and structuring systems. This language theoretical driving force nevertheless produced a lot of research and is responsible for the body of knowledge that forms the current state of prototype-based languages. This driving force can be briefly summarized as the quest for smaller, simpler and conceptually cleaner languages.

In section 2.4 we will discuss three concrete prototype-based languages in detail and in section 2.5 we will give a taxonomy of prototype-based languages according to the language features they implement. Let us first give a general overview of the idea of a prototype-based language.

2.3 Prototype-based Languages in Brief

A prototype-based language is actually just an object-oriented programming language without classes. The general idea of a prototype-based language is that all programming is done in terms of concrete directly manipulatable objects, called prototypes. In class-based languages, objects are created by *instantiating* a class and classes can share behaviour by means of *inheritance*. In prototype-based languages, the most commonly occuring ways to create objects are *ex-nihilo creation* (e.g. by putting a number of fields and methods between matching parenthesis), *cloning* (i.e., shallow copying) and *extending* other objects (a new object is an extension of an existing object).

Creating an object ex-nihilo means “just write down an object”, out of the blue in the same way functional languages allow one to “just write down a function” using a lambda. In many languages this is done by simply listing a number of slots between a pair of matching parenthesis. Cloning a prototype yields a shallow copy of that prototype. Here all the slots of the shallow copy get automatically initialized to the value of their corresponding slot in the original prototype. In some languages, new objects are created by listing a number of attributes that will distinguish it from another object. In other words, an object gets created by extending and overruling an existing object.

The structure of the objects in a prototype-based language is often much more flexible than the structure of the objects in class-based languages. Indeed, many prototype-based languages feature mechanisms to add and delete methods and data fields to an object during the lifetime of that object.

The sharing role that inheritance plays in a class-based language is taken over by a (possibly dynamically modifiable) “inheritance relation” between objects. This relation is usually referred to as *delegation* or *object-based inheritance*. The idea is that one object can dynamically identify another object to be its parent (through a so called “parent-of-link”) in such a way that every message that cannot be handled by the object itself is (automatically) delegated to the

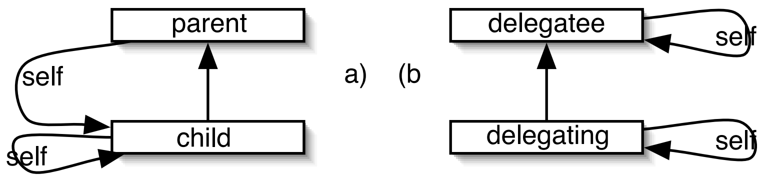


Figure 2.1: Delegation vs. Messsage Forwarding

parent object. This "inheritance link" between objects can vary substantially from one language to another. This is the topic of section 2.5. But in almost every language[1] it basically allows a programmer to state that "messages not understood by this object, should be delegated to the parent of the object".

A crucial point made by Lieberman [Lie86] is that delegation in prototype-based languages is not quite the same as a "simple" message forwarding mechanism often erronously called "delegation". The epitome of this error is the "delegation pattern" in the famous book on design patterns [GHJV95]. The important difference is that real delegation goes hand in hand with *late binding of self* as illustrated in figure 2.1a. Messages sent to the `self` (or `this`) pseudo variable in the parent have to "come back" to the object that originally received the message. This dynamic meaning of the `self` variable is exactly the same as the dynamic meaning of `this` in superclasses in a class-based language. The only difference is that delegation has this result on a per-object basis instead of classes. This is a crucial feature of delegation as originally defined by Lieberman and most prototype-based languages implement it that way.

Delegation gives rise to a feature of prototype-based languages that is commonly known as *parent sharing*. When two different objects designate an object to be their parent, then this parent is a shared parent and the meaning of `self` in the parent depends on the delegating object. Messages send to `self` will "come back" to the object that originally delegated the message to the parent. Notice that state changes of one child in the parent will affect all the other children as well: the parent is truly shared between the children. This is a feature of prototype-based languages that can only very clumsily be simulated in a class-based language with lots of "pointer plumbing".

Based on what we have explained so far, a brief (and very rough) initial comparison between prototype-based and class-based object-oriented programming languages is summarized in the following table we took from [NTM98].

[1]Kevo is a notable exception, see section 2.4.4.

	Class-based	Prototype-based
inheritance relationship	instance-of, inherits-from	delegates-to
creation metaphor	build according to a plan	clone an object
initialization	execute a plan	clone an example
one-of-a-kind	need extra class object	no class needed
infinite regres	class of class of ... ∞	not required

As already stated, objects in a prototype-based language obtain their characteristics from other objects using delegation. Objects in class-based languages obtain these characteristics both through the inheritance relationship between classes and subclasses, and through the instance-of relationship that exists between the class and the object. The object creation metaphor in a prototype-based language is the biological metaphor of cloning. In class-based languages, objects are created by "unwinding" a creation and initialization plan, to wit the constructor of the class. One-of-a-kind objects such as `True` and `False` (and singletons [GHJV95]) are extremely natural in a prototype-based language. Class-based languages require one to design a special (and often useless) class for them. This leads to strange situations such as in Smalltalk that has `True` and `False` classes as well as `true` and `false` objects. The last row of the table is a topic that was often treated in a stepmotherly fashion outside the incrowd of language designers. This is because it is only applicable to languages that enable meta-programming, a technique that was largely academic until recently. The problem is as follows. Uniformity of the language and the existence of meta-level information in a class-based language implies classes to be objects and thus implies classes to have a class. The result is an infinite conceptual chain of classes being classes of each other. This problem is absent in prototype-based languages. Everything is an object and that's really it, also at second sight.

2.4 Prototype-based Languages: Case Studies

Now that we have given a general background on prototype-based programming, we can move on to some representative prototype-based programming languages. We briefly review three important prototype-based ones: Self, Kevo and NewtonScript. Before we dig into the languages, let us briefly explain why we have chosen them, and, why we have omitted others.

2.4.1 A Case for the Cases

Here is an motivation for the languages.

- **Self.** Self is a pure prototype-based language with a uniform Smalltalk-like syntax. We have chosen to give an in-depth overview of Self, because Self can be considered a pearl in language design. Furthermore, Self is considered to be *the* prototype-based language "par excellence" by most people in the research community.

- **NewtonScript.** NewtonScript is a language the designers of which themselves "admit" that it is not really a shiny beacon in the sea of languages. Nevertheless we consider NewtonScript important because it was the first language (besides JavaScript, see below) to show that prototype-based languages are industrially viable. NewtonScript was the language in which the operating system and the applications of Apple's Newton palmtop series were implemented.

- **Kevo.** Kevo is a language developed by Taivalsaari in the context of his PhD thesis [Tai93]. The reason for discussion Kevo is that it is a prototype-based language that does not feature some derivative of delegation in order for objects to share properties. Instead, Kevo uses a mechanism called concatenation instead.

There are a few languages, which are for some particular reason "important enough" to appear in any text on prototypes, but which are not discussed here:

- **Omega.** Omega was developed by Blashek [Bla94]. At first sight, Omega is special as it is a statically typed prototype-based language. On second sight, Omega distinguishes between objects and prototypes. Objects are always created from prototypes and inheritance is only statically allowed on prototypes. As far as we are concerned, Omega's prototypes are simply called classes in other languages.

- **JavaScript and Flash.** JavaScript is probably the most widely used prototype-based language. Nevertheless, we will not discuss it as it cannot really be considered an example of good language design. Moreover, the new JavaScript 2.0 standard [Wal] adds a class-like feature to the language.

2.4.2 Case 1: Self

Self [UCCH91] [US87] was designed by a group of researchers at Sun Microsystems in the late eighties and early nineties. Self can be considered as a prototype-based version of Smalltalk that has been cleaned up considerably. Self is extremely minimal. It provides no classes, no variables and no control structures. Instead, everything is an object and all computation proceeds by sending messages to objects. Even methods are objects, and their activation records are conceptualized as clones of methods.

Self is more than a programming language. Just like Smalltalk, Self is a "world", a system in which one is is exploring and manipulating objects. But instead of working on "different levels" as is the case in Smalltalk (instance, class, meta class, ...), Self has mere objects. In a highly uniform world of malleable objects, the Self user interface provides an extremely direct object experience: e.g. the mouse pointer in Self is called "the hand" and the result of object manipulating operations "sticks to the hand". The resulting objects can be given a name in the global name space, which is again, an object. Having mere objects implies that programmers also use objects to build abstractions.

These emerge by copying objects and factoring out methods to shared parents. The system contains drag&drop tools to facilitate these programming patterns. By building a realistic programming environment, the Self group has also shown that the language can be provided with an industrial-strength, highly efficient implementation [SU98]. As a matter of fact, just about all of the JIT technology currently available in Java implementations was discovered by the Self team.

In Self, an object is a collection of slots. Slots map a name onto a method. Variables are emulated as a pair of read and write methods that access the same memory location. Some variable slots can be marked with an asterisk (`*`). In this case, the object that fills the slot is considered to be a parent object of the object in which the slot resides. Being a parent objects means that messages that are not found in the child object, will be automatically delegated to that parent such that the parent object will have the initial object as “self”.

Self: The Language

Everything in Self is an object and all computations are triggered by sending a message to some object. Following the Smalltalk example, Self distinguishes between *unary* messages (e.g. `myDict size`), *binary operator* messages (e.g. `3+4`) and *keyword* messages (e.g. `myDict at:key put:value`). Self features three kinds of objects which are all created by a different syntactic construction: plain objects, method objects and block objects. *Plain objects* are created by putting their slot names (together with a possible initial value for that slot) between vertical bars and separated by dots (e.g. `| x. y |`). Self allows any such ex-nihilo object creation to be followed by an expression. In that case, we speak of a *method object*. The idea is that the variables between vertical bars will act as the local variables of the method such that the evaluator will execute the expression in the context of the object every time the method is invoked through an invocation. Hence, in `(| x. y | x:3. y:4. x squared + y squared )`, `x` and `y` only exist in the context of the code to the right of the bars. Hence, it can be considered as a method with local variables `x` and `y`. In order to initialize the slots of an object, we can use the `<-` operator followed by an expression whose value will be the slot value. In the same vain, the `=` operator is for immutable slots. The following method object expressions illustrate this:

```
(| x =  3. y =  4 | x squared + y squared )

(| x <- 3. y <- 3 | y: y + 1.  x squared + y squared )
```

Method objects can be parametrised by preceding the local variables of a method object with a colon. Those variables preceded by colons are considered to be arguments for that method object. For example, the code `(| :x. :y | x squared + y squared )` is a method object expression that is parametrized by the variables `x` and `y`. Hence, the code represents a method with two arguments `x` and `y`. These method objects can be given a name by using them as the filler value for a slot in another object. For example, the expression `add: = (| :el | addLast: el)` could be used as part of an object definition. It binds the expression `| :el | addLast: el` to the name `add:`. Like Scheme, a Smalltalk-like shorthand is provided to allow parametrized methods with a name to be specified more naturally. This allows the object (read: method) `add: = (| :el | addLast: el)` to be written more naturally as the following expression shows: `add: el = (addLast: el)`.

Combining the syntax construction rules described until now, allows us to construct complex objects that look like this:

```
(|
 parent* = traits cloneable.
 x <- 0.
 y <- 0.
 r = (| x2y2 |
       x2y2: x squared + y squared.
       x2y2 squareRoot ).
  + p = (| newPt |
        newPt: copy.
        newPt x: x + p x.
        newPt y: y + p y.
        newPt)
|)
```

This code excerpt shows an ex-nihilo created plain object that inherits from `traits cloneable` with 5 slots: `parent`, `x`, `y`, `r` and `+`, the latter two of which are methods. The object represents a point prototype.

Just like Smalltalk, Self features blocks which are nothing but Scheme-like lambdas. Besides plain objects and method objects, *block objects* are the third kind of objects in Self[2]. The difference between a method object and a block object is that a block object belongs to the lexical scope in which it was created. They are used to pass around code, together with the context in which that code was created (i.e., together with the lexical environment). A block's code is evaluated when it is sent a `value` message. Block objects in Self are created in the same way method objects are, with the only difference that the parentheses are replaced by square brackets. The following code excerpt shows these concepts. First, a bank account is created and subsequently, a block `b` is declared in the context of the bank account. (Much) later on in the code, the block is executed twice. When doing this, the account is still visible to the block as one would expect from lexical scoping rules.

```
acc: bankAccount copy.
acc balance: 100.
b: [acc deposit: 50].
acc balance.    "returns 100"
\\a lot of code here
```

[2]Technically, blocks differ from ordinary objects because the “self” of a block is the object in which it appears, and not the block itself”.

```
        b value.
        b value.
        acc balance.    "returns 200"
```

Just like the methods discussed above, blocks can be parametrized with arguments. This in shown in the following excerpt

```
        adder: [| :x. :y | x + y].
        adder value: 3 With: 4   "returns 7"
```

In order to illustrate blocks, we notice that `true` contains a method `IfTrue:False:` that implements the well-known Church boolean system:

```
        ifTrue: t False: f = (t value)
```

As in Smalltalk, it is used in combination with blocks to delay evaluation:

```
        x != 0 ifTrue: [x reciprocal] False: 0
```

The final language feature to be discussed is Self's inheritance system. As indicated before, any slot can be assigned an asterisk. This turns the slot into a parent slot and makes the child inherit all the slots of the parent slot. Notice that multiple parents are possible in Self. Since Self is not a statically typed language, it is in general not possible to determine in which parent(s) a message will be found. Self therefore uses (several experimental method) lookup mechanisms [SU98]. As explained before, Self implements the "late binding of self" rule which means that self references in the parent will automatically refer to the child that delegated to the parent. Since many objects can take the same object to be their parent, Self implements the feature commonly referred to as parent sharing. The language feature commonly known as "super sends" is called *resends* in Self. For resends, a special dot syntax is used. The expression preceding the dot is either `resend` or a parent object. In the former case, the default method lookup in the parent is used. In the latter case, we talk about a *directed* resend. Examples follow:

```
        at: key = (resend.at: hash: key)
        (|
          parent1* = (| foo = 3 |).
          parent2* = (| foo = 4 |).
          foo = (parent1.foo + parent2.foo)
        |)
```

This concludes our overview of Self. Before moving on to the second language we discuss two techniques which can be applied to any prototype-based language, but which were developed in Self.

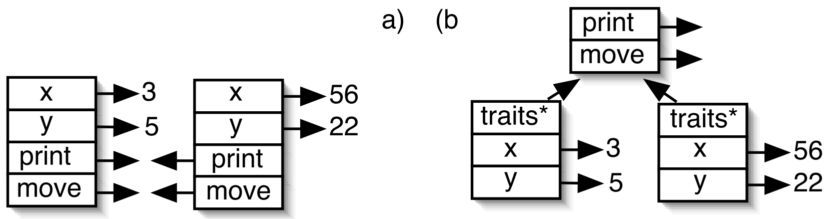


Figure 2.2: A Point Traits Illustrated

Idioms and Techniques in Self: Traits and Maps

The experience acquired by programming in Self has yielded two important idioms that can be used for other prototype-based languages as well, namely *traits* and *maps* [UCCH91]. Both are useful to reduce the amount of memory used by prototype-based programs.

Traits The idea of a traits object corresponds to method-tables in class-based languages. In class-based languages, it is often said that objects contain the state (i.e., instance variables) of a program together with "a pointer to their class which contains the methods". The latter is exactly what constitutes a traits object in a prototype-based program. Consider a point object `p1` that is copied in order to yield a second point `p2`. Both objects will list the same number of slots containing a pointer to the values of those slots. For "value slots", this duplication is useful as both points will probably contain different values. But for the method slots, the duplication is hopelessly inefficient. Even though the methods themselves are never really copied, all the pointer-slots are. This is depicted in figure 2.2.a. It is better to abstract these slots in a separate object, called a traits objects (this point-traits will contain all the "traits" of all points). This is depicted in figure 2.2.b. The traits object contains all the slots that are shared by all objects sharing the traits. The "actual" objects then inherit from the traits. Because of the late binding of the `self` variable, a method in the traits will always correctly refer to the right slots in the "actual object". The advantage is of course a severe gain in space.

In a language like Self, traits are not treated in a special way by the language. Traits are to be seen as one of several programming techniques to be used by prototype-based programmers in general. Explicitely introducing traits in a prototype-based language would yield a class-based language.

Maps Maps are not really linked to the notion of prototypes, but rather to dynamically typed object-oriented programming languages in general. Furthermore, instead of being a programming technique or language feature, they are to be considered an optimization technique that should be implemented by the language processor, i.e., the interpreter or the compiler. Consider two point objects with slots `x`, `x:`, `y: y` and `pointTraits*` to acces the coordinates and the point traits which contains all the methods applicable to points. Although we already have applied the traits technique presented above (leading to the `pointTraits`), it is easy to see that a lot of space is wasted just to store the *field names* themselves. And indeed this is a waste of space: although the values of the fields are different for both objects, the field names `x`, `x:`, `y:`, `y`, `pointTraits` and `pointTraits:` themselves are the same for all the clones. Nevertheless, the names are really necessary at run time because the language processor needs them for method lookup. This can be made more efficient simply by providing every object with a pointer to its "map". The map is nothing but a mapping from slot names to slot indexes. The "real" objects then merely consist of tables of things. Message passing thus becomes a combination of "index determination" and "memory indexing". When a message is sent to an object, first the map pointer of the object is followed. In the map, the index beloning to the slot name is determined and, subsequently, the index is used to read and invoke the slot in the actual object. For readers acquainted with C++, maps are actually v-tables for dynamically typed languages.

Conclusion

Self is a very pure language which is implemented in terms of a few primitives used to construct and modify objects. Self was very important to put prototype-based languages on the map. Perhaps one of the reasons why Self never really broke through is that Self required more than 50MByte of RAM to run, in times when 8 or 16 MByte were standard practice. Nevertheless, many Self techniques are still alive today: the morphic user interface still lives on in Squeak Smalltalk [GR01] and the Self compiler technology is at the heart of the Just In Time compiler in Java JDK [CU89].

2.4.3 Case 2: NewtonScript

In this section we shortly discuss NewtonScript, the second prototype-based language we will consider in detail [Smi98]. NewtonScript is a prototype-based language that is pretty unique, not because of its design but, because of the fact that it is one of the few ones that was so widely used in an industrial setting.

Prototypes on the Palm

NewtonScript was specifically designed to run the Apple Newton operating system. Programs written in the language had to run on different kinds of hardware and this portability requirement had to be combined with severe restrictions concerning memory size and processor speed (we are talking early nineties here). This is reflected in some aspects of the language where purity was compromised. The designers claim that "the language cannot be held up as a pristine example of language purity" [Sic, [Smi98]]. The language is said to be "simplified Scheme augmented with prototype-based features". NewtonScript is not object-oriented all the way down because it combines prototype-based features with procedural characteristics. Furthermore, it has "objects" that correspond directly to underlying parts of memory such as bitmaps. NewtonScript was compiled to bytecode (to be loaded onto the machine) with access to C++ primitives. Apart from the general design, the way objects are mapped onto the RAM-ROM architecture of the Newton gave it some unique characteristics:

Memory model The small memory size (640 K first version) of the Newton series had a big influence on the language. The Newton was shipped with a number of standard applications such as a note pad, an address book and an appointments manager which all read and write their data into so called object-soups, logic groups in the object-store. Third party products can extend these applications and access the data in the object soups.

View model Another major force that influenced NewtonScript was the user interface model of the machines. This model gave rise to a pretty exotic inheritance mechanism that was specifically designed for it. The idea behind the inheritance mechanism comes from the fact that the user interface is decomposed into views (e.g. a window) with subviews (e.g. a button in that window). Every view is represented by an object and one aspect of the inheritance mechanism consists of linking views with a `_parent` link to their surrounding view. Therefore (this aspect of) the inheritance mechanism of NewtonScript is sometimes called "visual scoping": messages not answered by objects (e.g. a button) get delegated through the `_parent` link to the surrounding view (e.g. a window). But aside from this `_parent` link, objects also have a `_proto` parent and it is the combination of these two that yield a fairly complex inheritance mechanism. The philosphy is that a view's `_parent` resides in RAM while its `_proto` is located in ROM. The `_parent` is the "data parent" of the view and the `_proto` is the "traits parent" containing the behavior. Hence, objects in NewtonScript consist of two inheritance chains. One chain is the chain of data objects linked with `_parent` links in RAM. Every part of this chain has a `_proto` link and these proto-parts also form a chain (by their respective `_parent` pointers) that resides in ROM. One can think of the parts in RAM as the traits while the parts in ROM are the real objects, and both the traits and the objects are linked in an inheritance chain.

NewtonScript: The Language

Let us now look at what NewtonScript programs look like and which features it possesses. As explained above, the proto/parent structure of the inheritance chain is the most interesting feature from a language theoretical point of view.

Non OO features. NewtonScript has a Pascal-like syntax and Pascal-like control structures. The exception handling mechanism looks like those of C++ and Java. Numbers, strings, arrays, `nil`, `true`, `symbols`, and binary objects are not objects. NewtonScript features two kinds of functions: global functions and user functions. Their difference is syntactically visible at the call-site. Global functions are called as expected: `Length(ar)`. User functions are called with the `call with` construction as exemplified by the following code excerpt:

```
fact:= func (n) if n=0 then 1 else n * call fact with (n-1)
```

Basic OO features: frames. The basic object-oriented notion of NewtonScript is a *frame*. A frame is a collection of slots, inspired by Self [Sic [Smi98]]. This is illustrated in the next code excerpt which shows how a frame `checkBox` is defined with 5 slots (`caption`, `bounds`, `state`, `Click` and `Redraw`). Note the double colon syntax for message passing.

```
checkBox := {
   caption: "Checkbox",
   bounds: { left:0, top:0, right:100, bottom:100 },
   state: nil,
   Click: func()
      begin
         state:= not state;
         self:Redraw();
      end,
   Redraw: func()
      begin
         DrawText(left+20,0,caption);
         SetFill(if state then Black else White);
         DrawSquare(left, top, left+16, top+16);
     end }

checkBox:Click();          - draws black cb
checkBox:Click();          - draws white cb
```

Inheritance and Variable Lookup. Inheritance in NewtonScript is sometimes called "comb-inheritance" because the hierarchy looks like a comb. Objects can contain two special slots named `_parent` and `_proto` that define inheritance in NewtonScript. For variable referencing and method lookup, these are interpreted as follows. When a message is sent to a frame, or when a variable reference is resolved, the rule is to look for a matching slot name in the receiver's frame. If it is not found, the lookup tries the frames in the receiver's `_proto` chain (recursively), then goes up to the `_parent` frame and tries that `_proto` chain, and so on. Hence the inheritance hierarchy looks like a comb whose teeth are `_proto`-chains tied together by `_parent`-chains. For variable assignment, the algorithm is subtly different. The assignment is only allowed to alter an object in the `_parent` chain, not one in the `_proto` chain. This is because the latter usually resides in ROM. If necessary, a new slot is created in the object on the `_parent` chain nearest to the object where the slot was found. This mechanism was called *value sharing* in Malenfant's classification [DMC92]. We will explain it further by the following two sample NewtonScript programs.

Example 1 The first example shows assignment along the "`_proto`-chain". The example shows how a traits object `protoCheckBox` for a check box is used to implement two concrete check boxes that have the traits as `_proto` parent.

```
protoCheckBox := {
   state: nil,
      Click: func ()
         begin
            state := not state;
            self:Redraw();
         end,
      Redraw: func ()
         begin
            DrawText(left+20,0,caption);
            SetFill(if state then black else white);
            DrawSquare(left,top,left+16,top+16);
         end };

   check1 := {
      _proto: protoCheckBox,
      caption: "Checkbox 1",
      bounds: {left:0, top:0, right:100, bottom:16} };

   check2 := {
      _proto: protoCheckBox,
      caption: "Checkboc 2",
      bounds:{left: 0, top:20, right:100, bottom:36},
      Click: func()
         begin
         inherited:Click();
         Print(state);
      end };
```

The focus of our attention will be the `state` variable. In the beginning, both `check1` and `check2` "have" a `state` variable because they "inherit" it from the `protoCheckBox` traits. Its initial value is `nil` which is the same as false. So this means that upon receiving the message `Redraw`, the color will be white. But now look at what happens when the message `Click` is sent to one of the objects. Although the message will be found in the `protoCheckBox` traits object, the assignment will happen in the original receiver. This is because the `protoCheckBox` object is the `_proto` chain of the receiver and assignments never happen in the `_proto`-chain. Hence, the `Click` method will actually add a `state` slot to the receiver. This slot wil then shadow the original state slot and from that point on, all references to the `state` variable will refer to the new slot. Notice that this aligns perfectly with the idea that objects in the `_proto`-chain usually reside in ROM: if they execute assignment statements, these will happen in the part of the object that is in RAM (i.e., the part in the `_parent`-chain).

Example 2 The following example exemplifies `_parent` inheritance and shows how NewtonScript's visual scoping works. This example shows a cluster of radio buttons (called `cluster`) that has `protoRadioCluster` as its traits object. Hence it inherits from that traits via the `_proto` link. Both radio buttons `radio1` and `radio2` have `protoRadioButton` as their traits and therefore inherit from it along the _proto link.

```
protoRadioButton := {
   state: nil,
   SetState: func (newState)
      begin
         state := newState;
         self:Redraw();
      end,
   Click:func ()
      self:ChooseButton(self),
   Redraw: func () .... end};

protoRadioCluster := {
   clusterValue: nil,
   chosenButton: nil,
   ChooseButton: func (whichBtn)
      begin
         if chosenButton <> nil then
            chosenButton:SetState(nil);
         whichBtn:SetState(true);
         clusterValue := whichBtn.value;
         chosenButton := whichBtn;
         self:ValueChanged();
      end};

cluster := {
   _proto: protoRadioCluster,
   ValueChanged: func() Print(clusterValue)};

radio1 := {
   _proto: protoRadioButton,
   _parent: cluster,
   caption: Cheddar,
   bounds: {left:0, top:0, right:100, bottom:16},
   value: cheddar };

radio2 := {
   _proto: protoRadioButton,
   _parent: cluster,
   caption: Swiss,
   bounds:{left: 0, top:20, right:100, bottom:36},
   value: swiss }
```

Apart from objects and traits objects (along the _proto-chain), the example also illustrates the visual inheritance mechanism of NewtonScript. Because the radio buttons are visual subcomponents of the cluster, they name the cluster as their `_parent` parent. This means that messages sent to the radio buttons will first be looked up in the radio button traits along the `_proto`. When this fails, the `_parent` pointer is followed and the message is looked for in that object and again in the `_proto` pointer of that object, and so on.

Conclusion

The authors of [Smi98] conclude that "programming with prototypes is pleasantly concrete. ..." and "we feel that prototype-based languages have great potential to improve programmer productivity and hope to see plenty of commercial applications in the future". As the most important design flaws in NewtonScript, they list:

- NewtonScript programs seldom contain full objects: a prototype is written with the assumption that another object will be `self` and will fill in the gaps using the special assignment semantics. This is an indication that programs are written by specifying traits (read: classes). We will see in chapter 4 that this reveals a deeper problem, namely that the construction of many objects *inherently* requires the unfolding of a construction plan.
- They missed private slots. They had troubles with method capture when developing the second version. Methods in parents (that were conceptually private to those parents in a new version) were by accident overriden by existing applications [SLMD96].
- Inheriting behaviour through the `_parent` chain turned out to be less useful than hoped. In many cases, a "view method" is specifically *not* intended to be inherited from parent views.

Nevertheless, NewtonScript showed that prototypes can be succesful in the real world. Apart from its intriguing inheritance scheme, this was also one of the reasons why we spent so much space on NewtonScript in the dissertation.

2.4.4 Case 3: Kevo

Kevo is the third language we discuss. The reason for including it is that Kevo does not feature delegation or inheritance at all. Its sharing is accomplished in exactly the way Wittgenstein's prototype theory proposed: clones are related to their prototype by a "similarity" link thereby sharing information because changes to one object can propagate to others. To this end, Kevo maintains a set of cloning families which are sets of clones spawned from the same object without modifications made to them. From a language theoretical point of view, Kevo shows that other organizational structures than delegation for prototypes have a lot of benefits too. The three concepts that play a central role in Kevo are *cloning families*, *propagation* and *concatenation*. Concatenation is the mechanism that takes care of incremental modification in Kevo.

Kevo is a system in which the programmer finds himself in a unix-like shell where commands can be executed. But instead of interpreting these commands in the context of "the current object": in Kevo, objects are like directories of variables and methods. The variables contain other objects.

Kevo: the language

Kevo is a keyword-based language with a syntax similar to Forth. This means that there are actually as good as no syntax rules and that control structures are used in the same way other keywords are. Message passing is denoted by a dot operator. In Kevo terminology, message passing is a special case of what is generally called *binding*. Kevo features four binding variants:

`x`	early binding in self
`self.x`	late binding in self
`resend`	binding of the same message in parent
`object.x`	late binding in other object

The table shows the four different ways an identifier `x` can be bound in code. Notice that the `self` keyword cannot simply be omitted. Omitting it causes the identifier `x` to be statically bound in the current object. `self.x` on the other hand will cause `x` to be looked up in a late bound way. Hence, in contrast to languages with modifiers like `final` or `static`, Kevo allows the binding to be specified at the access site instead of the declaration site. The consequences thereof are thoroughly explained in [Tai93].

Kevo objects are collections of named attributes. The object attributes are of the following four categories.

- `REF`-attributes are similar to class variables in a Smalltalk or static variables in Java. The `share` operation — to be discussed later — will not copy these variables such that all members of a clone family (see later) will share the same variable.

- `VAR`-attributes are similar to instance variables in a class-based language. Each time a clone of the object is made, the `VAR`-attributes are copied.

- `CONST`-attributes are like `REF`-attributes. Apart from being shared by the members of a cloning family they are immutable as well.

- `METHOD`-attributes contain methods. Conceptually these are deep-copied every time a copy of the object is made. This is the way Kevo accomplishes "late binding of self" [sic, see [Tai93]].

There are two operations (`HIDE` and `SHOW`) programmers can apply to these attributes. This way programmers can encapsulate object attributes to preclude other objects from accessing them. Strangely enough, Kevo's encapsulation mechanism is actually class-based encapsulation because objects that belong to the same cloning family can access each other's private parts just like Java objects of the same class can access each other. Hence when a Kevo object spawns a copy of itself, it can access its own as well as the copy's private parts.

Objects and Object Operations

Kevo objects always belong to a cloning family which is a set of clones that are created from the same object without modifications. Cloning families are maintained automatically and the programmer has no access to them. Each time attributes are added to or deleted from an object, a new cloning family is started and the system tries to keep the number of cloning families as small as possible. To this extent, "reversible object manipulation operations" will give rise to annihilation of generated cloning families as much as possible.

Kevo objects do not inherit from each other but are organised in a composition-based hierarchy: objects contain other objects. Hence, a Kevo system is to be thought of as hierarchically organized name space much in the style a unix directory system is organized. In contrast to a delegation-based system, the properties of the objects are independent from their position in the name space. This hierarchical organization is totally independent from the organization of the cloning families. As such, Kevo allows a programmer to organize her applications in the same way she would organize her libraries. Furthermore, multiple classifications are possible because objects can reside in several hierarchies. In general, these name spaces are manipulated by so called module operations as first specified by Bracha [BL92]. The module operations fall in two categories: those that operate on a single object and those that operate on an entire cloning family. The latter is always denoted by adding an asterisk to the name of the former. In general the syntax for applying a an operation to an object is `object OP attributes`. The different operators are summarized in the following table:

`ADD` and `ADD*`	add attributes to an object or a cloning family
`REMOVE` and `REMOVE*`	remove attributes from an object or family
`RENAME` and `RENAME*`	rename an attribute
`REDEFINE` and `REDEFINE*`	assign an attribute
`HIDE` and `HIDE*`	encapsulate an attribute
`SHOW` and `SHOW*`	decapsulate an attribute

Although Kevo has syntax to apply these operations, the general way to invoke them is through the programming environment. This environment works with respect to its propagation settings with determine how changes to objects are *propagated* to other objects. These settings can be "this object only" or "clone family". In the first case the operations applied to an object will only affect that object (and thus usually generate a new cloning family) or the entire cloning family the object belongs to. The reason why "this object only" generates a new cloning family is that Kevo works with so called *concatenation*: every object in Kevo is a linear composition of a clone and added properties. This means that operations on an object are always applied on a copy of that object. As explained, this will automatically spawn or merge elements in the network of cloning families. This is the topic of the following section.

Cloning in Kevo

Kevo does not have a clone operation that is applied *to* objects. Instead cloning happens by sending a message to an object. Although every new object contains a default implementation for the Kevo cloning methods, it is possible to redefine them in order to have a dedicated cloning strategy. This is a very useful feature. Indeed, as stated in [Tai93] shallow cloning is usually too shallow and deep cloning is usually too deep. Therefore designers of objects may want to implement their dedicated cloning strategy by redefining the standard cloning methods. Kevo objects generally feature three cloning methods:

- `share` is the primitive to make shallow clones. The idea is to implement more complex cloning operators based on this primitive. `share` will make a shallow clone of the receiving object thereby copying all the `VAR` and `METHOD` attributes, where a new "self" is filled in for the method. The `CONST` and `REF` attributes are shared between original and clone.

- `new` is a cloning method that corresponds to constructors in class-based languages. The idea is that this method calls `share` and performs some initialization on the copy.

- `clone` is a cloning method that is akin to `new`. The only difference is that `clone` will generate an exact clone of the receiver, while `new` will generate a "freshly initialized clone".

In Kevo terminology the inheritance mechanism of class-based languages is replaced by the cloning mechanism: when copying an object, one "receives" (Kevo terminology) all the methods from the original object. In order to keep the cloning families generated this way automatically consistent, the system executes a suite of pre and post modification rules. The pre-modification-rules are executed immediately before and the post-modification rules immediately after one of the operations of the previous section is efffectively executed.

There are three *pre-modification rules* that take care of spawning new cloning families whenever necessary.

- if the operation is object-specific and if the target object is the only one in its cloning family, then nothing happens.

- if the operation is object-specific and if there is more than one object in its cloning family, then a new family is created for the modified object.

- if the operation is family-oriented (i.e., an operation with an asterisk), then the operation is propagated to all the objects that are in the cloning family of the target object.

After applying these rules, the operation itself is applied and after that, the *post processing rules* are triggered. These might annihilate cloning families such that the number of cloning families is kept to a strict minimum:

- if the operation is object specific and if the attributes of the object match the attributes of a cloning families that resides "next" to the cloning family the object belongs to, then both cloning families are merged.

- if the operation is family-targeted and the attributes of the object after modification match the attributes of some of the families that reside "next" to the cloning family, then all the members in the family will be merged to that family.

Conclusion

Kevo was included in this dissertation because it shows that prototypes and delegation are not necessarily the same thing. Kevo objects are organised by a sophisticated "copy and modify" strategy, making them a true implementation of Wittgenstein's ideas.

2.4.5 Conclusion: "is-a" vs. Reentrancy

Class-based languages make a clear distinction between objects and classes. In a class-based system organisation, there are two important relationships between software entities: the *is-a* relationship is established between classes and help programmers to structure their object soup in a hierarchical way. Another relationship is the *reentrance* relationship between objects and classes. Classes contain code which can be re-entered for every object that belongs to the class. When looking at the software organisation techniques promoted by each of the three concrete case study languages, we notice that they have to deal with these same relationships as well, most of them quite clumsily:

- In Self there is only one kind of relationship between objects and this relationship is used both for conceptual purposes ("is-a") as well as for reentrancy purposes (i.e., the traits technique). This is problematic as it easily gives rise to multiple inheritance hierarchies: objects delegate to each other along an is-a link and delegate to their traits objects to establish a reentrancy link. But the traits objects in their turn also delegate to each other along an is-a link. In the best case this results in a multiple inheritance hierarchy, which is problematic in itself [SU98]. In the worst case it can result in strangely warped hierarchies the implications of which have never been thoroughly investigated[VDD94].

- In NewtonScript, the designers have also bumped into this duality in links. They have solved this by an inheritance mechanism that is quite unique but at the same time pretty hard to understand. Objects have both a `_proto` and a `_parent` pointer. The former is used to take care of reentrance. The latter is used to establish the is-a relationship. This results in a complicated comb-inheritance scheme with complicated combinations. For example, it is unclear whether proto-objects have `_proto` pointers themselves.

- Kevo seems to provide a satisfying solution to the problem. Objects are conceptually idiosyncratic entities that do not share code. Reentrance is accomplished by conceptually copying all the methods thereby filling in a new "self". The is-a relationship is established precisely according to Wittgenstein's theory: objects are-like the prototypes from which they were created and from which they differ in some ways. The downside of Kevo is that it requires complex implementation technology that keeps cloning families consistent behind the scenes. Unfortunately very little has been published about it.

We will return to this is-a versus reentrancy discrepancy in section 5.9 when we discuss the innovative solution to this problem put forward by our Pic% model.

2.5 Epilogue: The Treaty Of Orlando

Now that we have a really good view on prototype-based languages we will put them further in context using *The Treaty of Orlando.* The Treaty of Orlando [LSU87] was an influencial taxonomy paper that was written by David Ungar (one of the chief designers of Self), Henry Lieberman (the inventor of prototype-based languages in the OO sense) and Lynn Andrea Stein (author of a paper that "proves" that delegation is the same as inheritance [Ste87]). In the Treaty of Orlando, the authors explain having reached an agreement on some fundamental characteristics of object-oriented programming languages, during a meeting at the OOPSLA conference in Orlando. The result is an abstract taxonomy that can serve as a framework for classifying object-oriented programming languages.

The Treaty of Orlando states that both prototype-based languages and class-based languages can be considered as languages that feature objects and message passing, enriched with two fundamental mechanisms called *empathy* and *templates*. *Templates* are entities from which new objects can be created by a "cookie-cutting" mechanism. In prototype-based languages, templates are objects, while in class-based languages, they are classes. The other mechanism is called *empathy.* Empathy is the ability of a language to allow sharing of behavior between templates without explicit redefinition of that behaviour. In prototype-based languages, the empathy mechanism is being taken care of by delegation (or copying in Kevo), while in class-based languages, inheritance between classes plays the role of the empathy mechanism. The abstraction presented by the Treaty of Orlando can be depicted as in figure 2.3.

The Treaty of Orlando classifies object-oriented languages according to how their empathy mechanism varies along the following three dimensions:

- **when: static vs. dynamic** When is the sharing relation fixed? In pure class-based languages, the relation is usually fixed statically, although languages like Smalltalk allow the dynamic generation of subclasses. In "extreme" prototype-based languages, the relation is entirely dynamically fixed. Self, e.g., even allows objects to change their parent at run time.

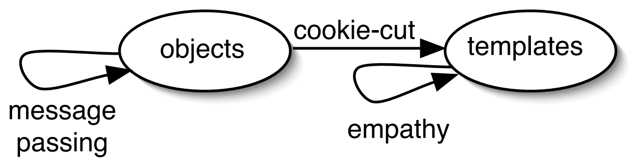


Figure 2.3: The Treaty of Orlando Schematically

- **how: implicit vs. explicit** Does sharing between objects happen automatically, or does a programmer explicitly have to direct the patterns of sharing? Most object-oriented languages have an implicit sharing mechanism: code does not have to specify "by hand" that it is shared amongst objects or classes. Messages that cannot be found in an object or class, are searched for in the parent.
- **for: per object vs. per group** Is behaviour specified for an entire group of objects or just for one single object? Can idiosyncratic behavior be specified for a single object.

Besides this classification of the empathy mechanism, the Treaty of Orlando classifies object-oriented languages according to the following two dimensions in which the template mechanism can vary:

- **what: objects vs. classes** A template is a "cookie cutter" producing objects. In class-based languages this role is played by classes. In prototype-based ones, objects play the role of cookie cutters as new objects are created from existing objects.
- **how: strict vs. nonstrict** Templates are strict if objects cannot acquire new attributes after they have been cookie cut from a template. This is typically the case in class-based languages. In prototype-based languages the templates are often nonstrict.

Based on these criteria for classifying the empathy and the template mechanisms, the Treaty of Orlando classifies object-oriented programming languages in the following table. Since the Treaty was published in 1987, we have added some more recent languages. Our own additions are added below the double line. We have added Agora and Pic% for completeness although they are the topic of the chapters to come.

	Empathy			Templates	
Language	When?	How?	For?	What?	How?
Actors	runtime	explicit	per object	objects	nonstrict
Delegation	runtime	both	per object	objects	nonstrict
Self	runtime	implicit	per object	objects	nonstrict
Simula	compile	implicit	per group	classes	strict
Smalltalk	creation time	implicit	per group	classes	strict
Hybrid	runtime	both	both	any	nonstrict

C++	compile	implicit	per group	classes	strict
Java	compile	implicit	per group	classes	strict
Kevo	compile	implicit	both	objects	strict
NewtonScript	runtime	implicit	both	objects	nonstrict
Agora, Pic%	runtime	implicit	both	objects	strict

For a complete in-depth explanation of the table, we refer to the Treaty itself [LSU87]. Perhaps worth noticing is the Hybrid language that was proposed by L.A. Stein [Ste87]. In this language, both classes and objects exist. Besides the usual mechanisms, an extra *promotion* mechanism is provided to turn objects into templates. In the following chapters, we will argue against this kind of language design that emerges from unifying distinct features of other languages. As we will argue in section 4.6 our view on language design is more "intersectional" in the sense that good languages are distilled by trying to remove and redesigning features, instead of by adding them. This view on language design was also promoted in section 1.2 of the introduction.

2.6 Conclusion

In this chapter we have given a thorough introduction of prototype-based languages. We have introduced their general properties and we have put them in historical and scientific perspective. Furthermore we discussed three important example languages in detail. Finally we have situated prototype-based languages in the realm of object-oriented languages using the Treaty of Orlando.

This chapter presents a frame of reference in which we can position our research of chapters 4 and 5.

Chapter 3

Classes, Prototypes and Open Networks

This chapter presents the main motivation for our work. We show that class-based languages are bound to perform extremely poorly in the context of open networks set out in chapter 1. However, prototype-based languages are shown not to be a panacea either. We will show that their property which most of their antagonists describe as "too flexible" can actually be traced back to identifiable shortcomings both in their denotational semantics as well as in the programming practices they promote. This chapter unravels these problems and concludes by pinpointing the fundamental problem of prototype-based programming languages. Subsequently, the manifestation of this fundamental problem will be shown to yield unacceptable security problems of prototype-based languages in open networks and mobile software.

3.1 Introduction

Now that we have an initial insight about the main differences between prototype and class-based languages, we can evaluate both paradigms in the context of the open networks we described in chapter 1. In section 3.2 we will demonstrate that classes are not really a viable option and that the very class-based paradigm poses fundamental problems for constructing the kind of software envisioned in chapter 1. This is the main motivation for using prototypes in our research. However, prototype-based languages have their problems too.

Antagonists of prototype-based languages often consider them to be "too flexible to write robust software". Solutions for this problem seem to point towards class-like features. To understand this tension and get a characterisation of the fundamental semantical differences between both paradigms, we will look at them from two radically different points of view:

- We will first have a critical look at prototype-based languages from a

software engineering point of view. This analysis will list a number of tangible problems of prototype-based languages all of which seem to ask for classes in order to solve them.

- Our second critique of prototype-based languages will be based on a **language theoretical point of view**. By having a look at the denotational semantics of prototype-based languages and comparing it with the standard denotational semantics of class-based languages the arguments of the first analysis will find their mathematical counterpart.

These two analyses are the topic of sections 3.3 and 3.4 respectively. Both analyses will allow us to clearly formulate the basic problem of prototype-based languages in section 3.5 which will essentially be that too many language features are defined on objects. In section 3.6 we show that this property results in severe security problems in the context of open networks and mobility, which renders the prototype-based languages we discussed in chapter 2 totally unacceptable in this context. A large part of our research consisted of distilling a powerful object-oriented programming language that does not suffer from these drawbacks without having to resort to classes again. This will be the topic of chapters 4 and 5. Let us now first have a look at our main motivation for using prototypes in the first place.

3.2 Evaluating Classes for Open Networks

In this section we argue that it is nearly impossible to write software in a class-based language for open networks that is not bound to result in inconsistencies and runtime errors after some time. In a nutshell, the argument boils down to the fact that:

> According to the Treaty of Orlando reviewed in section 2.5, classes are a static, **implicit** (i.e., transparent) sharing mechanism of state and behaviour between objects. In [GF99] the authors argue that "distribution transparency is impossible to achieve in practice, and precisely because of that impossibility, it is dangerous to provide the illusion of transparency."

As a consequence, we argue throughout the dissertation that prototype-based programming is fundamentally better suited to program software that is to run in open networks because:

> Idiosynchratic objects help abstracting semantically coherent software units by merging state and behaviour, the two essential ingredients in computer chemistry. Furthermore, thanks to encapsulation and polymorphism they are prepared for highly dynamic environments where emphasis is put on unanticipated interaction.

Before we start zooming in on these fundamental claims starting from section 3.2.2, we first mention some technical problems which might be circumvented by future class-based languages, but which currently are not. We therefore call them mere "inconveniences" because they are not inherently "unfixable".

3.2.1 Inconveniences Of Classes for Open Networks

Besides some general software-engineering drawbacks of class-based programming to be listed in section 3.3.1, the context of open networks raises a few extra difficulties when applying them. One of them is that the nature of the network will often require **new objects of new types to be introduced dynamically into the network**. This requires that new classes are somehow injected into the network. Technology in the form of dynamic class loaders can be a solution for this problem. However, as explained below, dynamic class loaders are a severe source of security related problems. In general, introducing unforeseen idiosyncratic objects, as featured by prototype-based languages, in a network is much easier and expressive a solution.

Another problem with classes and class-loaders is that, in current mainstream languages, **classes induce an inheritance relationship and class loaders explicitly use this** relationship to check type compliance of newly loaded classes. For example, if a networked system is written referring to a certain class `c` then it is not possible to dynamically inject a new class `c'` into the system that is to be substitutable for `c` and that is not a subclass of `c`, even if `c'` has the same method signatures as `c`. Of course a solution is to type the original system with an interface that is implemented by both `c` and `c'` but this solution is not feasible "after the facts". Again, prototype-based languages offer a much higher flexibility and render things much easier to manage.

Classes have two possible links, the implicit *instance-of* link and the *subclass-of* link. An inconvenience of that *instance-of* link is that, upon transmitting an instance over a wire, the corresponding class must be sent along. The *subclass-of* link forces the **recursive transmission of all the superclasses** as well, for the object would not be well defined otherwise (the set of methods and attributes would be incomplete). In a dynamically typed language (e.g. Smalltalk), the transitive closure problem ends here. But in a statically typed language like Java, argument-type classes, result-type classes and exception-type classes must be transmitted also — together with their corresponding transitive closures. In classless systems this is conceptually much simpler because objects are transmitted self-contained, with their state and behaviour together, without requiring a priori recursive transmission of peripheral classes.

Finally, in [SV97], Vitek makes an analysis of security problems in open networks and mobile contexts. He finds no less than five ways to circumvent the encapsulation of objects, every time by cleverly combining class-based inheritance and static (class) variables. But even before starting his devastating analysis he already remarks that class loaders are already a (technical) problem on their own as they only load classes once. This means that when two networked applications running on the same (virtual) machine use the same class, then they will start influencing each others behaviour by manipulating the class's static variables. Vitek shows that the **security breaches due to the implicit sharing relation** established by the static variables that reside in classes can be devastating.

3.2.2 Classes: Fundamental Problems in Open Networks

Apart from these inconveniences of class-based languages (notably Java) for open networks presented above, there are more fundamental problems. We repeat the quote from [GF99] that "distribution transparency is impossible to achieve in practice, and precisely because of that impossibility, it is dangerous to provide the illusion of transparency." A painful illustration of the truth of this statement is precisely the main problem with class-based languages in the context of distributed and mobile programming. The problem is that the implicit, unavoidable *class-instance* relationship becomes explicit under distribution and mobility because the cost of keeping the class-instance relationship consistent (and thus implicit) behind the scenes in a dynamic distributed system is prohibitive. Hence, the programmer has to deal with the relationship all the time, yet he has no way to control and manipulate it, or to avoid its in specific situations. The following sections elaborate on this problem.

Sharing vs. Distribution

The main problem of using classes in distributed systems and mobile systems is that classes call for the well-known conflicts between sharing and distribution:

- When shared data is kept centralised, the node containing the data becomes a bottleneck, and worse, the entire system is fragile because there is a dependency on that single node.
- If, on the other hand, the data is replicated among multiple nodes, the system must keep all the copies synchronized and we have a replica management problem.

In class-based programming, classes must be either centralised or replicated in every node. Both alternatives are no option in the open networks we target. Note that we do not argue that information sharing can or should be avoided in distributed systems: in fact it is essential. We *do* argue that using classes as the carriers of code and data in open networks is extremely problematic because instances cannot exist without their classes. Two concrete manifestations of the sharing vs. distribution conflict in classes are shown next.

State Sharing: Class Variables

A patent manifestation of the sharing vs. distribution problem is class variables (a.k.a. static variables) as already mentioned in [BGL98]. All current class-based languages overlook it. The semantics of class variables is not enforced in the presence of distribution: copying a class containing class variables from one node to the next does not start an underlying replica management system that would keep all the variable copies synchronised[1]. Hence distribution easily breaks the semantics of class variables. This is even more problematic in our context of open networks: if two devices go out of reach and they update a class variable with different values and afterwards rejoin the network, the inconsistency cannot be resolved unequivocally. The alternative of centralising classes is also not possible in processor clouds: in these dynamic networks there cannot be a central authority, e.g. suppose a device goes out of reach; who would be its central authority? It would have to stop operating until rejoining the network, wich is unacceptable.

One might reply to this state sharing argument by eliminating class variables from class-based programing, which is feasible. This would fix the problems presented so far. But the problem is in fact the sharing of *any* resource. As shown next, similar problems arise with code sharing — one of the principal roles of classes. One can eliminate class variables from class-based programming as suggested before, but one cannot "fix" the paradigm by overruling methods.

Code Sharing: Class Methods

Apart from class variables, method implementations are also shared among class instances[2]. As such, the problem is identical for methods. Suppose a node receives the same class along two different paths, but the two versions have a different implementation for a method. Which one is to be considered correct?

In class-based technologies, classes (e.g. class libraries) are often replicated among nodes without an appropriate replica or version management mechanism. The effects on distribution are somewhat relaxed by the standardisation of the basic classes (e.g. all the *java.** packages in Java). Since a copy of the basic classes can be assumed to exist in every node, replication consistency is guaranteed. It is clear though that this is a partial solution only, as any user-written class breaks the replication harmony. But even for standard libraries, there might be many versions of a class circulating in the system (e.g., JRE 1.1 vs. JRE 1.2). This issue can be solved by backwards-compatibility, i.e., by marking obsolete methods "deprecated" as in Java. This way, given any two versions of the same class, one class will be a subset of the other (i.e., the newer class will contain all the methods of the older class, some possibly deprecated). The interference problem is solved then by choosing the newest version, which is supposed to be compatible with the older one. This solution is weak and ad hoc, as classes would grow forever, full with deprecated methods, becoming

[1]Some middleware technologies such as PerDis [FSB+99] offer programmers in class-based languages library support to circumvent this shortcoming. However as we will explain in chapter 7, these middleware solutions are not free of problems either.

[2]Changing a method implementation is analogous to changing a class variable value. Either dynamically (e.g. in Smalltalk) or statically (e.g. by editing source code and recompiling), methods can be modified. The dynamic/static distinction does not add to the discussion in a distributed setting: the only important point is that nodes might have different definitions of the *same* class at the same point in time.

incontrollable entities. And even if only a few methods are deprecated, sharing this legacy code all the time is inefficient.

Workarounds

A solution is to make classes constant, so that replicas can be distributed without any synchronisation issues arising. This not only implies turning class variables into class constants (which is equivalent to overruling class variables from class-based programming), but also freezing method implementations. The consequence is that *every* change in the implementation of a class would forcibly imply the introduction of a *new* class in the system. However, defining a new class each time renders existing instances (which might live on another device) incompatible with the new version.

A class-based programming protagonist could argue that one can make class-instance relation explicit, e.g. letting an object specify, upon migration, if its class will stay in the origin node and be invoked remotely, or it will move also to the destination node; or letting a class specify whether its superclass will migrate together with it or not (in the latter case a remote-parent link would be established). But making these relations explicit and managing them "manually" is precisely what prototype-based programming promotes!

3.3 Prototypes in Software-Engineering

The previous section presented our main motivation for promoting prototype-based languages in the research context set out in chapter 1. However, prototype-based languages have their problems too and, as we will see, solving them seems to point back to classes. We now try to pinpoint this tension. First, this section presents a software-engineering driven comparison, and then section 3.4 presents a denotational vision on the matter. Both analyses will allow us to formulate clearly the fundamental problem of prototype-based languages in general and for open networks in particular in sections 3.5 and 3.6 respectively.

3.3.1 Advantages of Prototypes in SE

Arguments used to extoll prototype-based languages usually are pragmatic ones concerning the flexibility of prototype-based languages, and cultural ones concerning the underlying systems and visions on programming in general.

Pragmatic Differences

A general comparison of prototype-based and class-based languages is presented by Taivalsaari in [Tai98]. We merely summarize his analysis:

- In uniform class-based languages where everything is an object, one easily ends up with very complex concepts such as "the parallel hierarchy" in Smalltalk. Likewise, in Java classes are objects of the class `Class` and

`Class` is an object that is its own class. Prototype-based languages do not suffer from this problem. Everything is an object, also on second sight.

- Bugs or incompleteness in constructors can lead to meaninglessly initialized objects. In prototype-based languages, fields in clones always have a meaningful initial value, namely the value of the field of the cloned object.

- In class-based languages it is not possible to individualize the behaviour of objects unless one makes extra subclasses. In Self, we can e.g. put a "halt"-statement in one object which will only affect that object.

- Prototype-based languages support singleton objects [GHJV95] (such as `True`, `False`) for which no class needs to be constructed. Smalltalk has the classes `True` and `False` with `true` and `false` as only instances.

- In prototype-based languages two or more objects can inherit (by delegation) from the same object such that changes in the parent applied by one child are also "felt" by the other children. This powerful mechanism, called *parent sharing*, allows one to define views on objects. Simulating this in a class-based language leads to *extremely* cumbersome indirections as illustrated by the entire role-modelling problem [Fow97].

- Classes play no less than eleven roles [BL92], amongst others, object generation, description of representation and behaviour of objects, taxonomization of objects, code reuse, modularity, encapsulation and visibility definition and types definition. Of course by removing classes a lot of these roles will be taken over by objects. But in that case we know objects fullfil *all* the roles without arbitrarily deferring a few of them to classes.

- Prototype-based languages allow classes to be reintroduced to literally **classify objects**, possibly at the level of the programming environment. For instance, the language StripeTalk [GBO$^+$98] allows programmers to attribute objects with "stripes" and to query (groups of) objects according to their stripes (e.g. "Give me all the blue objects").

The Look-and-Feel of Prototype-based Languages

Apart from these rather technical arguments a lot of the argumentation in favor of prototype-based languages has to do with culture. These arguments are not very scientific in nature but proponents of prototype-based languages keep on citing them as being important [NTM98]:

- Prototype-based languages are **closer to the way people think** about knowledge [Lie86]. People think in terms of examples and not in terms of abstract descriptions. When talking about elephants, people have one or several elephants "in mind". We simply do not think in terms of "grey mammals with four legs, a trunk and big ears".

- Teaching a prototype-based language is **simple**. The student only has to learn how to handle concrete objects. Even at the level where one really starts to master a language and starts digging deeper into it, everything stays concrete. Concepts like meta-classes never pop up.

- One can argue that abstract concepts are only useful in mathematics or in fields that have been highly formalized by mathematicians (such as physics). Most modeling activities, however, do not involve abstract concepts. Proof of this is the tremendous difficulty with which programmers have to "distil" abstract classes from already made concrete implementations. Most "things" a requirements engineer encounters in the real world are concrete. Extracting the abstractions is actually the difficult phase called *design*. It has been argued that this phase becomes unnecessary in a prototype-based language. Instead, **Rapid-prototyping-development** (RAD), which turns a domain analysis directly into code, seems to be at the heart of pure prototype-based programming.

Inheritance vs. Delegation

In the late eighties and early nineties, when research in prototype-based languages was taking off, people started to wonder about the theoretical differences between delegation and inheritance. It was clear that delegation could simulate inheritance. Indeed, every class is simulated by an object that contains no state but only methods. The inheritance link is played by a delegation link between such objects. Instances are represented by objects that only contain state and have their class object as a parent. This simple scheme shows that it is fairly easy to simulate class-based programming in a prototype-based language. But is it also possible to do the simulation the other way around? In her paper "Delegation is Inheritance", [Ste87] Lynn Andrea Stein "proved formally" that it is, but we strongly disagree with her results. The paper essentially shows that in a Smalltalk-like language (where classes are objects and where subclasses can be dynamically created!) it is possible to do all the programming with classes. State is represented by class variables and delegating objects are represented by dynamically created subclasses. Parent sharing is also accomplished because two objects (read: classes) share the same parent object (read: common superclass). Of course, Smalltalk is a bit of an exception in the class-based world. Most class-based languages do not allow classes to be subclassed dynamically nor do they attribute "object facilities" like message passing, a `this` and a `super` pseudo variable to classes. I.e., in most class-based languages it is not possible to do object-oriented programming using classes alone. We therefore strongly disagree with Stein and consider prototype-based languages more powerful due to the dynamic mechanisms they feature.

Traits: The Malenfant Analysis

In section 2.4.2 we have presented the traits technique developed in Self. Although traits objects are but a programming idiom and are not an enforced language feature, we have to face the fact that the traits technique *actually* boils down to writing class-based programs in a prototype-based language: the traits objects play the role of classes and the instances refer to the "class" by means of a "parent pointer" instead of an "instance-of pointer". Aside from some technical details this is the same as class-based programming. Therefore, Dony [DMC92] argues that it is better to speak about object-centred programming instead of prototype-based programming. Object-centred programming refers to writing programs that do not use these "class-centered techniques" too much but instead put the object as a "design entity" in a central position.

Conclusion

These comparisons resulting from our literature study seem to turn out extremely positive for proponents of prototype-based languages. However, the following section shows that prototype-based problems have their problems too.

3.3.2 Problems with Prototypes in Software Engineering

Most people that hear about prototype-based languages have the general vague feeling that they are "too flexible" and do not offer programmers the ability to specify or impose a "rigid design" in the way classes do. We have tried to characterise this feeling by reducing it to a number of technical problems [DDD03b].

- The construction of certain objects requires a **construction plan** to be executed (e.g. building up a GUI). In class-based languages, it is possible to formalize this plan as a constructor residing in the class of the object. In prototype-based languages objects are created by cloning and it requires a lot of discipline to make sure the right initialisation procedures are applied to the clone. This problem is worsened by the fact that prototype-based languages are often equipped with very powerful programming environments that enable a direct manipulation of objects on the screen. As a result it is easy to end up with "half baken objects", objects that are copied and left behind somewhere without being properly initialized. Part of this problem is that a cloning operator is applied *to* the object without active participation of the object being cloned.

- The **prototype corruption problem** was described in [Bla94]. Prototypes (e.g. the empty string) can be inadvertently modified. This can lead to subtle bugs in other objects that are created from that prototype and that rely on certain properties of the prototype (e.g. that it is the empty string) because a state change to the prototype might affect future clones. Again this is because the clones are created by an "external" clone operator that is applied *to* the prototype and the prototype has no active role in this cloning process.

- As concluded in section 2.4 on Self, NewtonScript and Kevo, prototype-based languages suffer from a **re-entrancy problem** in the sense that two orthogonal relationships seem to be needed to express object-oriented software, to wit an is-a relationship and a code-reuse relationship. As explained explained in section 2.4.2, the problem can be circumvented using the traits technique which easily gives rise to multiple inheritance problems. Another solution is a fairly complicated inheritance strategy such as the comb-inheritance proposed by NewtonScript (see section 2.4.3) that works differently along `_proto` and `_parent` links. The basic problem is that the better modeling capabilities of prototype-based languages (e.g. use parent sharing to model roles) are paid for by weaker code reuse giving the programmers the feeling of yielding less structure.

- Some concepts are **inherently abstract**. E.g. in order to describe a stack one *has* to go to the abstract level. When writing the code for `push` and `pop`, one writes code for all possible stacks (empty stacks, full stacks,...) and hence one is *by definition* "writing a class". The problem is that "stack" is an inherently abstract concept because the code is written for stacks in general. Writing the code for "the empty stack" as a prototypical stack e.g. would be counter-intuitive: since it is impossible to pop an element from the empty stack, the empty stack prototype would not need a `pop` method. But a stack without a `pop` method can hardly be considered a prototype for stacks.

- Some prototype-based languages allow for **varying template hierarchies**. Indeed, if we look back at the table presented in section 2.5 on the Treaty of Orlando we see that quite a few prototype-based languages offer a non-strict definition of templates (i.e., objects in prototype-based languages). Many languages allow objects to change structure dynamically. In Self, e.g. it is possible to change the parent of an object simply by assigning one of its parent slots. Many programmers consider this as far too flexible. Whatever taxonomy they come up with, a single (accidental) assignment instruction might change the entire structure of the program.

- A common problem of most prototype-based languages is their complete lack of what we call **reflection protection**. Whereas reflective class-based languages define their reflection operators in classes, most prototype-based languages feature such operators to modify the structure and behaviour of objects dynamically. Also, these operators usually allow one to bypass the encapsulation boundaries of objects: by switching to the meta-level, malicious programmers can circumvent encapsulation and protection barriers. Until now, this seemed to be a binary situation: either one avoids reflection, or one accepts the fact that the language is totally unsafe. We show in chapter 4 that this does not necessarily have to be case.

All these software engineering problems can be solved by re-introducing classes but we want to avoid that because of our analysis of section 3.2. In chapters 4 and 5 we distill a new family of object-oriented programming languages that do not suffer from these problems and that do not feature classes as an implicit sharing mechanism. The new language family will be distilled by analysing the denotational semantics of class-based and prototyp-based languages which is the next topic.

3.4 A Language theoretical Point of View

In the previous section we have presented the software-engineering arguments. In this section we present our second, language theoretical, analysis of prototype-based languages. This analysis is based on our study of the denotational semantics of prototype-based languages with respect to the one of class-based languages. Before we can present our argument we will briefly summarize the denotational semantics of class-based languages.

3.4.1 The Cook Semantics of CBL

In this section we briefly review the main results of W. Cook [CP89] who was the first to give a satisfactory denotational semantics for classes, inheritance and instantiation. Although older denotational descriptions of class-based languages exist (e.g. [Wol88] and [Kam88]) they are mere mathematical versions of the classical method lookup algorithm. The Cook semantics is simple, concise and really explains the classes-inheritance-instantiation phenomenon.

Objects are records

The functional version of the Cook semantics[3] treats objects as records of attributes. Attributes are methods[4] which are procedures that can refer to a "hidden" *self* and a *super* pseudo-variable. The domain equations used to model objects for such a language are mere records mapping names to methods:

$$Object = Identifier \rightarrow Method$$

whereby methods transform argument objects into a result object:

$$Method = Object^{\star} \rightarrow Object$$

Message passing consists of method lookup (i.e., record indexing) followed by method invocation. This is characteristic for every implementation of class-based programming languages:

$$pass_{CBL} : Object \times Identifier \times Object^{\star} \rightarrow Object$$

$$pass_{CBL}(o, m, os) = o(m)(os)$$

[3]In his PhD thesis [Coo89] Cook gives a full semantics of several languages, with state. The state makes the formalism much more complicated without adding to the understanding of classes, instantiation and inheritance.

[4]We will only focus on classes, objects, inheritance and message passing. We will ignore important features such as updatable state and exception handling. Adding them to the semantics is not a trivial exercise, but does not really add much to the understanding of the concepts under investigation.

Generators and Fixed Points

Given this definition of objects, let us now have a look at how classes and inheritance are modeled by the Cook semantics.

A class can be considered as a reentrant object: the code of a class of an object is like the code of the object itself, except that it should be usable for other objects of the same class as well. Hence, the code of a class can be thought of as the code of a true object, except for the state that resides in "self". Hence, we say that a class is an object that is parametrized over "self". Mathematically:

$$Class = Object \rightarrow Object$$

Object instantiation then takes a class c and "fills in the self of that class with a real object s". The result of that operation is the required instance i. Hence $i = c(s)$. But which real object s should that be? It is of course the instance i itself. Therefore $i = c(i)$. This equation gives us the specification for one of the essential results of W. Cook, namely that an instance of a class is a fixed point of that class. In other words, instantiation of a class is mathematically explained as taking the least fixed point of that class:

$$new : Class \rightarrow Object$$

$$new(c) = fix(c)$$

Since new objects are generated from the class by taking the fixed point of that class, the semantic analogue of classes is called *generators*. For the mathematical details we refer to [CP89].

Inheritance in class-based languages

One of the reasons of the success of Cook's semantics is that it also explains inheritance in a satisfactory way. To inherit from a class is "to add some delta to the class" so that a new class emerges. This "delta" can be seen as another class that has another variable, namely *super* with the well known semantics. Hence, the set of subclasses (often called mixins when they get a special status in a language, see [BC90], [Hen91]) can mathematically be denoted as:

$$Mixin = Object \rightarrow Object \rightarrow Object = Object \rightarrow Class$$

I.e., a mixin is parameterized by itself and by its parent, i.e., a class that is further parameterized by a parent. Inheritance takes a mixin (the code of the subclass) and a class, and produces a new class. In this new class, all self references of the original class and the self references of the mixin code now have "to point" to the same "new" self reference of the resulting class. The corresponding inheritance operator thus looks as follows:

$$inherit_{CBL} : Class \rightarrow Mixin \rightarrow Class$$

$$inherit_{CBL} = \lambda c.\lambda m.\lambda s.c(s) +_r m(c(s), s)$$

where $+_r$ is the right-preferential record combinator that takes care of overriding. The inheritance operator takes a class c and a mixin m and produces a class λs.... We clearly see that the passing around of self variables takes care of correctly binding "self" in sublasses, the famous "late binding of self" property of object oriented languages.

For more details of the Cook semantics, we refer to [Coo89] which proves that its is indeed applicable to a wide range of class-based object-oriented languages: the semantics was shown to cover Smalltalk, Simula, Beta and a few others.

Super is NOT a first class object

In the above semantics, we can see that, once inheritance is involved, the existence of a `super` variable emerges such that the resulting subclass can refer to the code in the superclass. From the domain equations we infer that this super variable denotes an object: the "parent object" of the object under investigation. This is however misleading. The reason is as follows: since the super variable is an *Object*, it is a record whose "self" is already fixed, and since "the self of the super has to refer to the new object", this self is indeed fixed to refer to the new object. This implies that, if we would require the super object to be a referable entity in our language, we would have an object whose self does not refer to itself! That is the reason why:

- In Smalltalk, a `super` pseudo variable exists which looks as if it denotes a regular object that can be used just like any other object. However, the programming tools preclude programmers from using it as such. Therefore in Smalltalk, the expression `super eq:self` is valid, while `self eq:super` is not. Indeed, the first expression can be classified as a *super send* and not as a *super reference*. In the second code excerpt on the other hand, a message is sent to `self` with `super` as an argument: this means that `super` is referred to as if it were a normal object. But as we stated this is not semantically consistent because "the self" of that object is not itself, but the inheritor. That is why the Smalltalk programming tools will reject the second expression.

- In Java, the `super` pseudo variable was removed alltogether. Instead, a dedicated syntax was added to do super sends (i.e., `super.m()`). Again, the reason is that it does not make sense to allow "a first class super object" to be accessible from within running programs: it is an object whose "self" is not itself!

In brief, in a class-based language **`super` is not an object**. In class-based languages, objects are indivisible entities even when inheritance is involved. Class-based languages do not allow us to speak about "parent objects".

3.4.2 A Semantics of PBL

Now that we have explained the essentials of the denotational semantics of class-based languages, we can present our analogous treatment of prototype-based languages. One of the things the Cook semantics teaches us is that basic inheritance (i.e., to make "old" attributes visible to the newly created class/object) can easily be achieved through right preferential record combination, but that in order to get correct late binding of self (i.e., to make "old" self references point to the "new" self), we need to parametrize the records with a self variable and let this variable be bound by the inheritance operator. This insight leads us to the conclusion that in any satisfactory denotational semantics for a prototype-based language, objects need to be represented as generators and not as records. Indeed, in addition to message passing, inheritance (or some other form of object extension) is also defined *on objects*. And since inheritance requires generators, **objects necessarily have to be modeled as generators**.

But there is more to explain. Since objects are modeled as generators, and since self is also an object, self also has to be modeled as a generator. This leads us to a new generator domain:

$$Generator = Generator \rightarrow Record$$

$$Object = Generator$$

Here we see that objects are now modeled as generators that take themselves as an argument. This is done by a self application, an operation we dubbed *wrap* in [SD95]:

$$wrap(g) = g(g)$$

Notice that this wrapping of objects cannot be done at object creation time. This would "fix the self" of an object such that it becomes impossible to inherit from the object later on. Hence, objects are modelled as unwrapped generators. This defers the time of wrapping to the latest possible moment, which is when the record is really needed: message passing time. Hence, the message passing operator of prototype-based languages looks fundamentally different from the class-based one. It consist of first wrapping the receiver, subsequently doing the standard method lookup in the resulting record, and applying the found attribute to the actual parameters:

$$pass_{PBL} : Object \times Identifier \times Object^{\star} \rightarrow Object$$

$$pass_{PBL}(o, i, os) = wrap(o)(i)(os)$$

This theoretical result is also very tangible in practical implementations of prototype-based languages: the message passing operator is parameterized with an extra "hidden" argument denoting the receiver of the message.

Again, this generator model is very satisfactory because of its ability to model the inheritance mechanism (called delegation) of prototype-based languages. Indeed, in a stateless model an extension of an object is simply created by "applying" a mixin to an existing object. The result is a new object which has the original object as a parent object:

$$inherit_{PBL} : Object \rightarrow Mixin \rightarrow Object$$

$$inherit_{PBL} = \lambda o.\lambda m.\lambda s.o(s) +_r m(o(s), s)$$

Now that we have presented the essentials of a denotational description of prototype-based languages, our language theoretical critique can be presented.

3.4.3 Encapsulation vs. Inheritance

Our language theoretical critique against prototype-based languages is essentially that the prototype-based model suffers from an *inherent* encapsulation problem. From a certain distance this is already visible in the above message passing operators. By "feeding" a "strange self" to an object, the object can be "fooled" and encapsulation can be broken.

From a more technical point of view, it is easy to come up with examples of inheritance breaching encapsulation. This was also noticed by Snyder for class-based languages [Sny86]. Snyder distinguishes between *message sending clients* and *inheriting clients*. Inheritance is always a breach of encapsulation because inheriting clients are granted more access to the implementation than message sending clients. In a class-based language there is a clear separation between inheriting clients and message sending clients: inheriting clients are (statically declared) subclasses and cannot access living objects. But the problem becomes more severe in prototype-based languages as inheriting clients can dynamically breach the encapsulation of existing objects. To illustrate this, consider the following example (which was also the running example in [SD95]) written in an imaginary prototype-based language where objects can be dynamically extended. The example show how an inheriting client (in this case, the extension of the circle that is local to the window thief) can inadvertently access the private variable of the object from which it inherits, simply by overriding a method.

```
CircleWithExpensiveGoldenWindow IS OBJECT
  PRIVATE VARIABLE ExpensiveGoldenWindow
  METHOD DrawInWindow(aWindow)
     ...
  METHOD Draw()
     SELF.DrawInWindow(ExpensiveGoldenWindow)

WindowThief IS OBJECT
  METHOD StealGoldenWindow (aCircle)
     RETURN (aCircle EXTENDED WITH OVERRIDE
                       METHOD DrawInWindow(aWindow)
                          RETURN Window).Draw().

WindowThief.StealGoldenWindow(CircleWithExpensiveGoldenWindow)
```

Our example shows the most extreme way an inheritor can breach the encapsulation of a parent object: by intercepting self sends where private variables are passed around. But the situation is much worse in real-world languages. A real world language will typically have access modifiers such as the `protected` modifier in Java. This modifier states that only inheriting clients are allowed to access a variable which is normally not accessible by message sending clients. But in a prototype-based language with objects and nothing but objects, *any object* can play the role of an inheriting client and hence any object can access those protected variables! In the research context of objects that roam networks described in section 1.1, we consider this a **severe problem**. We will get back at this in section 3.5. The problem is **inherent to prototype based languages**. Indeed, whereas class-based languages make a clear cut distinction between "inheriting clients" (i.e., classes) and "message sending clients" (i.e., objects), this distinction is absent in prototype-based language with only objects. To avoid this kind of problems, the language family we will propose will adhere the *extreme encapsulation* principle: objects should be subject to message passing and message passing alone. As we will show in chapter 4, this will still allow for more complex language features like object extension and object cloning.

3.4.4 Extreme Encapsulation: Objects + Messages

When it comes to encapsulation, the fundamental problem of prototype-based languages is that inheritance and message passing are both defined on objects. Denotationally this is clearly expressed by the fact that objects and generators are aligned. In class-based languages this problem does not exist. Message passing is defined on records of methods and fully respects encapsulation. Inheritance is an operation that is not defined on objects, but on a special kind of language entities: generators, i.e., classes.

In section 3.3 we have already coined the term "reflection protection" referring to the fact that reflection operators should not allow clients to access more of an object at the meta level than they already can at the base level. This is an important point because many prototype-based languages define reflection operators and require these reflection operators to be applied, again, to objects. Needless to say, this application of relection operators *on* objects is another source of encapsulation breaching.

A final similar argument that can be made is focused on cloning. While class-based languages define object-creation on classes, prototype-based languages define a cloning mechanism on objects. It is very easy to construct an analysis about cloning similar to the one we have presented here for inheritance. By having a cloning operator that operates *on* objects, the language makes it possible to make copies of objects that "should not be copied". We will show in section 3.6 that this can lead to severe security problems in open networks.

We therefore conclude that we should look for languages in which **objects can be mathematically modeled by records and in which the class-based message passing operator of section 3.4.1 can be used**. All models in which clients of object can do more than send messages to those objects are potentially dangerous. We will refer to this property as **extreme encapsulation**. It will not be very surprising that this property is of vital importance to mobile and distributed systems as described in section 3.6.

3.5 The Fundamental Problem of Prototypes

Now that we have thoroughly investigated the problems of prototype-based languages both from a software engineering point of view (section 3.3) and from a language theoretical point of view (section 3.4) , we can pinpoint the exact problem of existing prototype-based languages: **The fundamental problem of existing prototype-based languages is that, from a semantical point of view, too many operators are defined on objects.** This statement is motivated by the analysis we gave in the previous sections:

- Every prototype-based language has the notion of message passing. So all these languages define a message passing operator on their object domain.

- As we have discussed in our language theoretical analysis, a major problem with conventional prototype-based languages is that they align their object domain and their generator domain. This is a breach of encapsulation. The main cause is that the inheritance operator is defined on objects.

- Section 3.3 explained the problem of incomplete objects and the prototype corruption problem. This problem is pretty much the same as the inheritance case: the object that is subject to cloning plays no active role in the process, and as such, message passing clients can "mess up the object" inadvertently. Both problems are due to the fact that prototype-based languages feature an *external* cloning operator that is applied *to* the object being cloned in such a way that that object does not play an active role in the cloning process.

- Section 3.3 identified the problem of varying templates. It basically boils down to the fact that parent assignment is too flexible a language feature for most programmers. Combined with the language theoretical arguments presented in section 3.4 parent assignment it also is a source of encapsulation problems. Again, this can be characterised by the fact that parent assignment is an operator defined on objects.

- A more severe manifestation of the problem occurs if we (as most prototype-based languages actually do) add introspection or reflection operators to the language. We have already called this problem the *reflection protection problem* in section 3.3.2. Indeed, by introducing reflection in the language, the operators defined on objects become really accessible in the language itself. This has the consequence that nearly all protection mechanisms of the language can be bypassed. As discussed in section 3.6, this is *really*

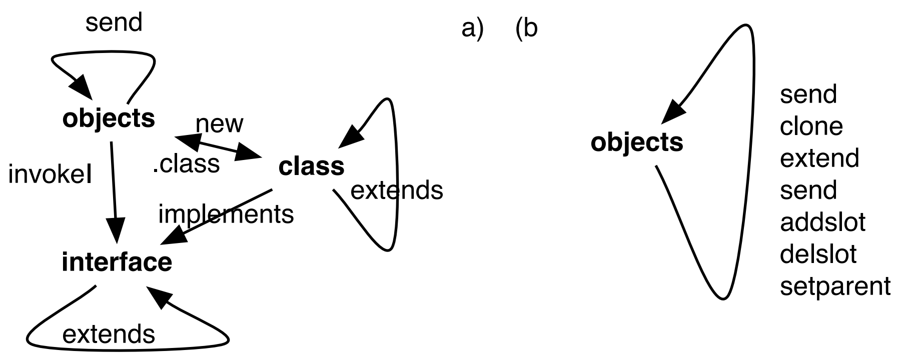


Figure 3.1: Language Values and Language Operators

problematic in a language used for distribution and mobility because of the importance of protections against hostile hosts.

The situation is schematically depicted in figure 3.1. Whereas class-based languages have a multitude of language values that are connected by language operators (fig 3.1a), prototype-based languages have only one kind of value, to wit objects. They therefore define all their language operators directly on objects (fig 3.1b). The consequence thereof is that prototype-based languages give programmers the feeling of being too flexible and that they suffer from inherent encapsulation problems.

3.6 Evaluating Prototypes for Open Networks

Prototype-based languages suffer from inherent encapsulation problems because of the multitude of language operators they define on objects. During our analysis we already heralded the problems this poses in an open distributed and mobile setting. In this section we analyse this further. But let us first shed some light on the different levels of security in network-oriented programming.

3.6.1 Issues in Security

It is clear that security is an important issue in distribution on open networks. This is especially the case when combined with code mobility, the topic of chapter 9. Thorn [Tho97] carefully distinguishes *safety* from *security*. Safety is what language designers can relegate to system software. Security, is not.

- *safety* means that bugs in applications will *never* affect the execution of other parts of the computational environment. This involves the guarantee that applications do not manipulate pointers outside their address space and so on. Safety is largely an aspect of the operating system and of the runtime of programming languages. An example of the latter is the guarantee to check for array operations to stay within their bounds.

- *security* primarily concerns applications or hosts whose application logic was explicitly devised to be malicious. Thorn [Tho97] distinguishes between four levels of security:

  - *Communication level security* is meant to avoid communication with a malicious partner. It involves secure communication protocols based on cryptography. This is beyond the scope of our research.
  - *Operating system level security* is close to safety. Safety is often ensured by hardware. Operating system level intervention is needed if hardware protection is impossible, absent or undesirable. An example is access control to files. This is not the focus of our research.
  - *Abstract machine level security* is the security one gets when an additional layer exists between applications and the operating system. The Java sandbox model is a good example of this. But as section 3.6.2 shows, there are better solutions for this.
  - Finally, and most important for our work, there is the **programming language level of security**. This involves issues such as scope rules and access rules for the way data is manipulated. Unfortunately, programming language level security is often treated in a stepmotherly fashion by language designers[5]. As we will shown in section 3.6.3 prototype-based languages are weak in this. As Thorn [Tho97] puts it *"...especially, security are issues that have not received enough attention so far in the area of programming languages"*. The principles of extreme encapsulation and reflection protection described in sections 3.4.4 and 3.5 are to be viewed in this light.

  The boundaries of safety and security are not crisp. Safety is a necessary ingredient of security. It is not a sufficient condition.

Continuing the analysis of security at **the programming language level**, Vitek [SV97] distinguishes between four types of security problems in mobile systems. They are especially relevant for open networks.

- *breach of secrecy:* Breach of secrecy occurs when a computation unauthorizedly accesses the direct state of another computation.

- *breach of integrity:* Breach of integrity means that a computation can somehow change the state of another computation illegally.

- *masquerading:*. A computation that pretends to be a computation which it is not by usurping its identity.

[5]At "The first European Lisp Workshop" it was generally agreed in a plenary session that the absence of security is a painful shortcoming of Lisp in modern application engineering.

- *denial of service:*. A computation can excessively consume the finite resources of another computation.

By illustrating these security breaches in Java, Vitek shows that object-oriented encapsulation is not enough to protect objects. However, the attacks he presents all involve a combination of inheritance and static (class) variables. So his analysis for class-based languages is akin to the one about inheritance and encapsulation in prototype-based languages we presented in section 3.4.3.

3.6.2 Capabilities, Security and the Granovetter Operator

An issue often found in the literature when combining objects, distribution and security is the notion of *capabilities* as found in the Eden Programming Language (EPL) [Bla85]. The idea is that kernels can create capabilities that represent access rights for objects. Objects can never create them but can only pass on capabilities to other objects. As Miller puts it, in a capability-based system, *connectivity begets connectivity* [MMF01]. This is important for the abstract machine level security discussed above. Instead of letting distributed software (applets, say) read your system in all kinds of uncontrolled ways (using string-based file and directory descriptors in combination with native libraries, e.g. `getFile(f,d)`) it is better to pass along an initial reference to the code. All other references to resources should be obtained through, and only through, that object. Although a capability can be anything (it might be an integer indicating which access rights one has), in an object-oriented system they are best represented by objects. Based on this, Miller describes that *all* connectivity should result from successively applying the **Granovetter operator**[6] as depicted in figure 3.2. The figure shows that, if Alice has a reference to Carol and Bob does not, then the only way for Bob to get a reference to Carol is that Alice **introduces** Carol to Bob. This can happen by sending a message `foo` containing a reference to Carol. However, it can also take the form of introducing Carol to Bob upon creation of Bob, in the case that Alice creates Bob. Yet other forms of the Granovetter operator are programming language scoping mechanisms that introduce Carol to Bob by using the scope rules that relate Bob and Alice. Miller argues that architectures that allow objects to obtain a reference to resources in *other* ways than applying this operator are very likely to have security holes. Good examples of flagrant bypasses of the operator typically are string-based object accesses (e.g. `System.getDatabase("Secret Database")`) of which the security is very hard to manage without leaks popping up. As we will see, the ChitChat model endorses this vision completely. The only way an object is able to access a resource is when it was give a reference to that resource by another object as a consequence of message passing.

[6]The operator was named after the sociologist Mark Granovetter who introduced the kind of diagrams depicted in figure 3.2 to study the evolution of human acquaintance relationships.

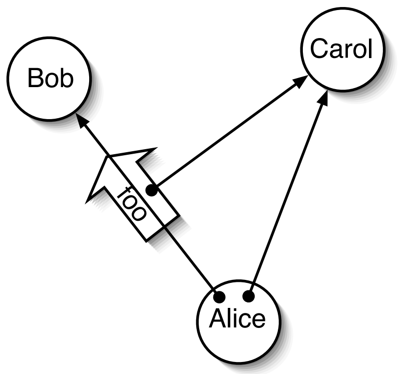


Figure 3.2: Miller's Granovetter Operator

3.6.3 The Security Performance of Prototypes

The analysis of section 3.2 leads us to conclude that for open networks, especially compared with mobility, prototype-based languages fundamentally outperform class-based ones. Surely language implementations can apply aggressive optimisation techniques to improve code sharing (such as Kevo's family algorithm of section 2.4.4). However, the very definition of prototype-based languages forces programmers to think about idiosyncratic objects, to reason carefully about the relationships between those objects (possibly via shared traits — see section 2.4.2) and above all, to make these relationships *explicit* in the design of their systems. As seen in section 3.2.2, shunning implicit relationships this way is a vital property for on open networks.

As thoroughly explained in sections 3.4 and 3.5, conventional prototype-based languages have fundamental problems too. The fact that they define all their language operators (message sending, cloning, inheritance, reflection,...) on the concept "object" results in them being too flexible and possessing inherent encapsulation problems. We conclude this section by showing that this leads to severe security problems when deployed in open networks and mobile environments. All cases shown below are examples of objects that obtain references to other objects by circumventing the Granovetter operator.

Delegation vs. Security. Until now, we have extensively discussed the encapsulation problem from a software engineering point of view. But in open networks, this situation is totally unacceptable. By dynamically extending an object, techniques such as overriding, self sends and privileged access to private state, unacceptable security breaches such as the ones discussed above are impossible to prevent. Especially Vitek's *breaches of secrecy* can be constructed easily. As explained in section 3.4.4 a radical adherence to message passing is really the only available option.

Cloning Operators vs. Security. A similar argument holds for cloning operators defined on objects. We have already explained how problematic this is in the context of regular software engineering (see section 3.3) but in the context of distribution on open networks and in the context of mobile systems this problem is far more severe, taking into account the programming language level security outlined above. The following code excerpt shows the consequences of a cloning *operator* that is to be applied *on* objects without sending them a message. By copying the money without sending a message, a malicious object can buy for ever on the network.

```
badAmazoneProxy IS OBJECT
  VARIABLE Amazone = Ref(www.amazone.com)
  METHOD buy(book, iMoney, shippingAddress)
    WHILE not(theCowComesHome) DO
      Amazone.buy(book, CLONE(iMoney), shippingAddress)
```

In our scenario of section 1.1.1, Harry carries an i-ticket on his Gizmo the very identity of which has to be transfered to the cashier in order not to use it multiple times to buy the butter. In a classic prototype-based language, this cannot be assured. These are the kind of problems the Vitek analysis presented above calls *masquerading*.

Reflection vs. Security. It is commonly accepted that modern object-oriented programming languages "should" feature meta programming and reflection techniques. However, as we already explained thoroughly in section 3.5, reflection is a potential source of problems when contrasted with encapsulation. Indeed, if objects reveal more on the meta level than they do on the base level, then the meta level can easily be used to breach the object's encapsulation boundaries, unless the meta level is restricted to some objects that belong to a certain "class". In all other cases, access to the object is virtually free simply by going to the meta level. In 3.3 we have called the requirement that objects should not reveal more at the meta level then they do at the base level *reflection protection*. In an open distributed and mobile setting, this requirement becomes tremendously important. The following code excerpt shows how a malicious client can try to break the encapsulation boundaries of an object given that the meta level allows one to "dissect" objects using slot access operators like `HASSLOT`, `SETSLOT` and `SETSLOT`. The problems illustrate Vitek's *breach of secrecy* and *breach of integrity*.

```
METHOD break(safeObject)
  LOCAL slotName = void;
  REPEAT
    slotname:=generateRandomNameContaining(''visa'')
  UNTIL HASSLOT(safeObject,slotName)
  visaNr := READSLOT(slotName,safeObject)
  SETSLOT(slotName,0)
```

3.7 Conclusion

In this chapter we have shown that class-based languages have paradigmatic problems in order for them to be viable in open networks combined with mobility. This was our main motivation for arguing in favour of prototype-based solutions. However, we have also shown that prototype-based languages as defined in the previous chapter are no panacea. We have done this by scrutinized existing prototype-based languages, both from a software engineering and from language theoretical point of view. We have identified a number of problems which basically boil down to the fact that prototype-based language define too many language operators on their object model. This results in encapsulation problems and in a number of software engineering problems which programmers usually perceive as prototype-based languages being too flexible.

In the context of distribution and mobility, especially the encapsulation problems are extremely problematic. We have therefore formulated the extreme encapsulation and reflection protection principles which states that objects should be fully encapsulated in such a way that every operation performed on objects should be accomplished through message sending. The following chapter demonstrates that this not necessarily has to lead to uninteresting languages that only feature record-like objects without any structure.

Chapter 4

Intersecting Classes and Prototypes: The Agora Family

In chapter 3 we have stated that class-based languages have fundamental paradigmatic problems in the context of open networks and that classic prototype-based languages have unacceptable encapsulation problems. We have identified this problem from a denotational point of view and have formulated clear criteria for a denotational semantics to avoid these problems. Built hereupon, a new family of object-orientation is distilled in this chapter. First a denotational model is conceived based on the observations of chapter 3. Subsequently, it is used to define a full-fledged object-oriented language family called Agora. The new family combines the advantages of class-based and prototype-based languages but avoids their disadvantages.

4.1 Introduction

Now that we have a solid understanding of the semantical tension between class-based languages and prototype-based languages, based on the denotational arguments presented in section 3.4.3 and the related argument about language operators being defined on objects in section 3.5, we can try to resolve the stalemate outlined in chapter 3. Building further on the denotational analysis of section 3.4 this chapter drafts a denotational model for classless object-oriented programming that endorses our extreme encapsulation principle explained in section 3.4.4. Subsequently, this denotational model will direct us in distilling a full-fledged object-oriented programming language family that has the desired properties of both class-based and prototype-based languages without re-introducing classes.

Although combinations of prototype-based and class-based language families have been proposed in the past, the resulting languages are usually straightforward unions of prototype-based and class-based languages features. They typically feature classes, instantiation, dynamic extension of objects previously instantiated from classes, reclassification schemes, and sometimes operators to "promote" objects to classes. Unfortunately, as explained in section 1.2, such languages not only combine the advantages of the desired features but also their disadvantages. Furthermore unexpected interactions of the features arise, often with undesirable side effects. Therefore, these languages are often hard to reason about and tend to be big with lots of baroque features. In contrast, our work adheres to a more intersectional point of view. The language family we propose unifies the positive features of class-based and prototype-based languages into one consistent model that avoids their drawbacks.

In the following section, we continue our denotational analysis of chapter 3 by presenting a theoretical language, called MiniMix, that clearly shows — by construction — that powerful classless languages exist that do not suffer from the encapsulation and security problems outlined in chapter 3. Subsequently, the Agora model will be introduced in section 4.3 as a full-fledged programming language family that endorses the ideas of MiniMix. Section 4.4 restates the essence of the MiniMix denotational semantics in Agora terminology. **It will be the central yardstick for many language features presented throughout the remainder of the dissertation**.

4.2 Extreme Encapsulation Denotationally

The most obvious difference between class-based and prototype-based languges is the difference in inheritable entities. As explained, class-based languages strictly separate their objects and inheritable entities into objects and classes which at the semantic level boils down to records and generators. Prototype-based languages strive towards a unification of these entities by removing classes and, instead, define inheritance directly on objects. Therefore, the object model employed by conventional prototype-based languages consists of generators which suffer from *inherent* encapsulation problems.

But rather than conclude that pure class-based languages (or uninteresting languages featuring only objects and messages) are the only solution, we will reconsider the design characteristics of object inheritance, keeping in mind the need for a clear distinction between inheriting and other clients. However, a straightforward restriction of the visibility of the internal details of objects to a limited "set of valid inheriting clients" must be approached with caution. As long as objects can *arbitrarily* subscribe to such a set by some kind of declaration, the problem remains. Solutions where the object under extension *itself* is not an active participant in the subscription process are *not* a substantial improvement. We will now propose an inheritance mechanism that takes this into account. The mechanism is characterised by the fact that the object under extension is the one in control of the extension process. Inheriting clients thus have no option but to send the object a message. The message *might* give rise to an extension process, *if* the object receiving the message agrees to be extended.

At the denotational semantics level, this boils down to the fact that generators are always encapsulated. This is achieved by a hygienic use of wrapped generators to denote objects. Wrapped generators are encapsulated objects and therefore adhere to the extreme encapsulation principle presented in section 3.4.4. But the semantics of inheritance requires generators. This can still be accomplished by giving objects a limited access to *their own* generator. So, objects will be wrapped generators and some messages sent to these objects will give rise to a mechanism being triggered that accesses the (unwrapped version of the) generator and performs extensions to it. The mechanism is exemplified by a toy language, called MiniMix, the single goal of which it is to give a solid foundation of Agora. The basic idea of MiniMix is that some messages might be implemented by a special kind of methods, called *mixin methods*. These mixin methods access the generator of the object in which they reside and define an extension of this generator. The resulting generator is immediately wrapped before it is returned as the result of the method.

In what follows, we present the syntax of MiniMix. A program is an object expression OE. Object expressions are identifiers, message expressions, `self`, `super` and ex-nihilo created objects. These objects are abstractions AB listing attribute descriptions AD. Every attribute description is either a regular method[1], or *a mixin method*. Mixin methods also list attribute descriptions.

$$\begin{array}{lll}
P & \rightarrow & OE \\
OE & \rightarrow & ME \\
 & | & I \\
 & | & \texttt{self} \\
 & | & \texttt{super} \\
 & | & \texttt{object}\ \mathrm{AB} \\
ME & \rightarrow & \texttt{send}\ OE_1\ \texttt{(}\ OE_2\ \texttt{)} \\
AB & \rightarrow & \texttt{[}\ AD^{\star}\ \texttt{]} \\
AD & \rightarrow & \texttt{method}\ I_1\ \texttt{(}\ I_2\ \texttt{)}\ OE \\
 & | & \texttt{mixin}\ I_1\ \texttt{(}\ I_2\ \texttt{)}\ AB
\end{array}$$

The idea of mixin methods [SCD+93] is as follows. When someone sends a message to an object and when that message happens to be implemented by a mixin method, then a new object is created whose parent is the receiver of the message, and whose extra attributes are the ones that are listed inside the mixin method. The new object is a "view" on the original one. To illustrate this, consider the following code that is used to construct points and circles.

```
[ mixin Point1D ( x ) [
    method getX (ignore) x,
    mixin Point2D ( y ) [
      method getY (ignore) y,
      mixin Circle (r) [
         method getR (ignore) r ] ] ] ]
```

[1]We only consider unary methods.

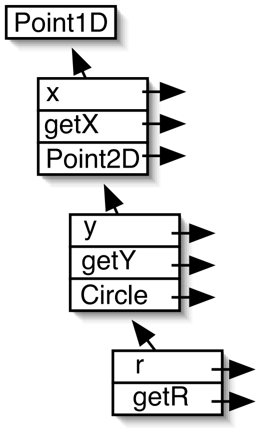


Figure 4.1: Mixin methods Illustrated

Suppose this object is known as `Root` then the expression `send (send (send Root Point1D (0)) PointD2 (0)) Circle (1)` will construct the trigonometric circle. This object can be depicted as shown in figure 4.1. It clearly shows how the object results from subsequent mixin method applications to the root.

Let us now have a look at the denotational semantics to explain why mixin methods do respect the extreme encapsulation principle described in the section 3.4.4. For the sake of clarity, the semantic domains of MiniMix are summarized below:

$$\begin{array}{rcl} Object & = & Ident \rightarrow Attribute \\ Attribute & = & Object \rightarrow Object \\ Generator & = & Generator \rightarrow Object \\ Env & = & Ident \rightarrow Object \\ Mixin & = & Object \rightarrow Generator \end{array}$$

In the semantic equations of MiniMix, object expressions are always evaluated using the current unwrapped self (g in the equations below), the current environment (e in the equations) and the current super object (p in the equations). The interpretations of identifiers and pseudo variables then are

$$\begin{array}{rcl} eval_{OE} : ObjExpr & \rightarrow & Env \rightarrow Object \rightarrow Generator \rightarrow Object \\ eval_{OE}(\texttt{self}) & = & \lambda e.\lambda p.\lambda g.wrap(g) \\ eval_{OE}(\texttt{super}) & = & \lambda e.\lambda p.\lambda g.p \\ eval_{OE}(I) & = & \lambda e.\lambda p.\lambda g.e(I) \end{array}$$

The interpretation of `self` confirms that objects are modelled by encapsulated objects. A self reference is achieved by wrapping the "current generator". Stated differently, generators are only passed around behind the scenes and will never be injected into the programming language value space.

The semantics of message passing consists of selecting the appropriate attribute in the receiver and applying it to the semantic object denoting the argument. Notice that this is the same kind of message passing as the class-based one discussed in section 3.4.1. No self parameters are being fed into the object. Objects are fully encapsulated.

$$\begin{aligned}
&eval_{ME} : Msg \to Env \to Object \to Generator \to Object \\
&eval_{ME}(\texttt{send}O_1 I(O_2)) \\
&\quad = \lambda e.\lambda p.\lambda g.eval_{OE}(O_1)(e)(p)(g)(I)(eval_{OE}(O_2)(e)(p)(g))
\end{aligned}$$

As can be seen from the grammar, an object can be created ex-nihilo by writing an abstraction `object` AB. Therefore, the semantics of ex-nihilo created objects requires understanding the semantics of abstractions. The body of a mixin-method is also an abstraction. Since the body of a mixin-method is a modifier, its denotation is a "$Mixin$" and hence abstractions are denoted by values from the $Mixin$ domain:

$$\begin{array}{rcl}
eval_{AB} : Abstr & \to & Env \to Mixin \\
eval_{AB}(\texttt{[}AD^{\star}\texttt{]}) & = & \lambda e.eval_{AD}(AD^{\star})(e)
\end{array}$$

An ex-nihilo created (encapsulated!) object is nothing more than a wrapped mixin with an empty super object (`[]`):

$$eval_{OE}(\texttt{object}AB) \quad = \quad \lambda e.\lambda p.\lambda g.wrap(eval_{AB}(AB)(e)(\texttt{[]}))$$

An abstraction is a sequence of attribute declarations. As above, all attribute declarations are evaluated making use of the current environment, the current super and the current unwrapped self. This gives us a specification for the semantic function of attribute declarations. A list of attributes is recursively defined by the concatenation of two such other lists[2]. The same therefore goes for its semantics.

$$\begin{array}{rcl}
eval_{AD} & : & AttDec \to Env \to Object \to Generator \to Object \\
eval_{AD}(A_1\texttt{;}A_2) & = & \lambda e.\lambda p.\lambda g.(eval_{AD}(A_1)(e)(p)(g) +_r eval_{AD}(A_2)(e)(p)(g)
\end{array}$$

[2]We are aware of the fact that this is actually an ambiguous grammar. However we decided to keep it that way for notational convenience as it makes the semantics much simpler.

At the lowest level in such a sequence, methods and mixin methods are declared. This declaration binds a name to a single parameter function containing the effect of invoking the attribute (records are represented as `[ i→` $\lambda o.$`...]`). For methods this effect consists of evaluating the object expression making up its body. But for mixin-methods however, that effect consists of taking the current (unwrapped) generator (i.e., the receiver, unwrapped) and extending it with the mixin that corresponds to the body. The resulting generator is cleanly wrapped to yield an encapsulated object that can be returned as the result of the mixin method.

$$\begin{aligned}
&eval_{AD}(\texttt{method}I_1(I_2)O)\\
&\quad = \lambda e.\lambda p.\lambda g.[I_1 \rightarrow \lambda o.eval_{OE}(O)(e[I_2/o])(p)(g)]\\
&eval_{AD}(\texttt{mixin}I_1(I_2)AB)\\
&\quad = \lambda e.\lambda p.\lambda g.[I_1 \rightarrow \lambda o.wrap(inherit(g, eval_{AB}(AB))(e[I_2/o])]
\end{aligned}$$

Since all objects in MiniMix are denoted as encapsulated records, MiniMix adheres to the extreme encapsulation principle. However, by cleverly allowing objects to passing around their own unencapsulated generator, they are capable of delivering extensions of themselves upon reception of a message. This is exactly what mixin-methods do. Hence, MiniMix shows that it *is* possible to design a prototype-based language that features nothing but extremely encapsulated objects but that still features dynamic object extension.

4.3 The Agora Model

The argument developed in the previous section merely focusses on inheritance. This section uses the MiniMix model as a basis to devise a full-fledged classless object-oriented programming model, called Agora, that features object extension, cloning and reflection, without sacrificing extreme encapsulation.

Agora was actually the topic of Steyeart's PhD thesis [Ste94], but we have contributed sufficiently to the language and to its full comprehension in order to include it in this dissertation [De 98a]. Historically, Agora was defined as a minimal object-oriented language in which nothing but objects and messages where incorporated. It was planned to add richer features on top of this model in order to study their semantic interactions with the basic message passing model. As a surprise however, it was perfectly possible to add these features without changing the initial message semantics passing at all. Agora has shown that it *is* possible to construct a language that is solely based on objects and messages, and to which features like inheritance, cloning and reflection can be added without adding extra language values such as classes, and, without defining extra operators on objects the way other prototype-based languages do.

In what follows we will often draw the parallel between Agora and Scheme. This is because many of Agora's features are the object-oriented equivalent of similar Scheme features. One of the similarities of the languages has to do with syntax. In the same way Scheme's syntax actually defines a syntax framework instead of just one single syntax, the syntax of Agora is a framework that allows for the definition of many different languages. Indeed, in the same way versions of Scheme can be completely different because of the different set of special forms they implement, different scions of the Agora family can be completely different because of the different "reifier messages" they implement. Hence, Agora is not a language but a language family instead. Just like Scheme's suite of special forms can be extended by macro-programming, Agora's suite of reifier messages can be extended by reflection. This, however, will be deferred to section 4.5.

4.3.1 Agora: A Language Family

Apart from the syntax, there are plenty of other similarities between Agora and Scheme. These are in fact consequences of the syntactical choices made for Agora. As we will see, one of the consequences of Agora's syntax is that Agora allows for an easy treatment of programs as data structures, which opens up the possibilities for reflection. This will be the topic of section 4.5. In the vanilla variant explained here, the emphasis is completely on objects and messages.

Agora Message Expressions. Scheme uses a uniform notation but distinguishes between "ordinary functions" and "special forms". Whereas Scheme uses applicative order evaluation for its ordinary functions, special forms are evaluated differently. E.g., when encountering `(define x 2)`, Scheme will apply the 'define' procedure to the unevaluated `x` and `2` parameters. The fact that `x` is not evaluated and that `2` will get evaluated depends completely on `define`. Each special form uses its own evaluation order. Likewise, Agora knows two sorts of messages: ordinary messages and reifier messages. Ordinary messages correspond to ordinary function applications. An example is `3 + 4` where `+` is sent to the evaluated `3` with the evaluated `4` as argument. Reifier messages correspond to the special forms of Scheme. An example is `x VARIABLE:3` which is used to define a variable in Agora by sending the `VARIABLE:` message to the identifier `x` with the expression `3` as parameter. Hence, as in Scheme, the essential difference between ordinary messages and reifier messages is their evaluation order. Agora reifier message names consist of completely capitalized or boldfaced identifiers, depending on the implementation[3].

Agora also distinguishes between receiverful and receiverless message expressions. Receiverless message expressions are exactly like "receiverful" message expressions, except that a receiver expression is syntactically missing. At this point it is not far wrong to think of them as function calls.

Finally, all messages come in unary, operator and keyword form, just as in Smalltalk and Self. Hence, Agora features 12 kinds of messages that vary along the dimensions $\{receiverless, receiverful\}$, $\{ordinary, reifier\}$ and $\{unary, operator, keyword\}$. The complete system of messages is summarized in figure 4.2. We do not give an example of operator reifier messages because there is no

[3]Another possibility is to prefix all reifier messages names by a special symbol such as μ.

capitatised analog of things such as `+`[4] to have reifier operators.

Receiverful		Operator	`3 + 4`
	Ordinary	Unary	`3 abs`
		Keyword	`dict at:"key"`
		Operator	
	Reifier	Unary	`message SUPER`
		Keyword	`myVariable VARIABLE: 4`
Receiverless		Operator	`- 5`
	Ordinary	Unary	`myVariable`
		Keyword	`at:key put:athing`
		Operator	
	Reifier	Unary	`SELF`
		Keyword	`CURRENTLY:not IN:use`

Figure 4.2: Agora Message Expressions

Example: The following expression elaborates on the Agora syntax. The informal meaning of the expression is to install a method `compute:value` in the object in which the expression occurs.

```
compute:value METHOD: ((SELF try:value) + (5 abs))
```

This is a reifier keyword message `METHOD:`. Its receiver is the receiverless ordinary keyword message expression `compute:value`, here acting as a formal pattern. The argument `value` is a receiverless ordinary unary message expression. The argument of the `METHOD:` message is an ordinary operator message expression `+` with `5 abs` as argument. The receiver of the operator message is an ordinary keyword message expression `SELF try:value` with `SELF` as receiver. `SELF` is a receiverless reifier unary message expression.

Agora Objects. The simplest Agora objects are literals such as integers, floats, characters, strings, booleans and `null`. In addition to these built-in objects, new objects are constructed ex-nihilo by listing their attributes between square brackets, separated by semicolons.

```
[ ...; ...; ...;  ... ]
```

The entries of the object must be valid Agora message expressions. This typically will be reifier messages such as `x VARIABLE:4`.

4.3.2 A Concrete Agora Scion: Some Reifiers

Just as variants of Scheme are defined by their special forms, variants of Agora are defined by their reifier messages. This section explains some elementary

[4]In versions where reifiers are recognised as boldfaced or prefixed with μ, this poses no problems.

reifier messages defined in most members of the Agora family[5]. Most of them define new attributes.

Variables. The simplest kind of attributes one can install in an object's slots are variables. Variables are created by sending the `VARIABLE:` reifier message to an identifier with the initial value of the variable as argument. The `VARIABLE:` message installs two slots. Whenever a variable `x` is declared by sending the message `x VARIABLE: 5`, a reading slot named `x` and a writing slot named `x:` are installed. These slots are accessor methods to the variable. As in Self, users of the object can read the variable by sending `x` to the object. They can modify the variable by sending `x:` with the new value as an argument.

Methods. As illustrated before, sending the `METHOD:` reifier message installs a method in an object. The receiver of that message must the formal pattern of the new method. The argument of `METHOD:` can be any expression serving as the body of the method. Expressions may be grouped between curly braces as in Java. Recursive methods are programmed by sending messages to `SELF`. Evaluation of this receiverless reifier unary message expression always returns the 'current' receiver.

Views. In section 4.2 we have explained "mixin methods" that deliver extensions of their receiver upon invocation. In Agora, we have two versions of such mixin methods: mixins and views. A view is what we have called a mixin method in MiniMix[6]. It is a method whose body contains a group (delimited by { and }) of expressions which will be evaluated in a new object that has the receiver as parent link. The following example shows a `point` onto which circle views can be added by sending `circle:` messages to the `point` object.

```
point VARIABLE:
  [ x VARIABLE:0;
    y VARIABLE:0;
    circle:r VIEW:
      { radius VARIABLE:r;
        inCircle:p METHOD:
          { ((p x) sqr + (p y) sqr) sqrt <= (SELF radius) }
      }
  ]
```

Just as in MiniMix, invocation of a view creates an extension of the receiver (i.e., an object whose parent is the receiver) but does not destructively change the receiving object. Each time a `circle:` message is sent to the `point`, a new object is created with the receiver (`point`) as parent-of link. The slots of this extension are determined by evaluating the body of the view in the context

[5]The exact technicalities of the reifiers discussed here come from Agora98, a version of Agora implemented on top of Java [De 98b].

[6]Over the years, the terminology has changed somewhat. Whereas originally Patrick Steyaert talks about mixin-methods [SCD$^+$93], [CDDS94] makes a distinction between functional mixin-methods (that add a layer to an object but do not change the object itself, and imperative mixin-methods which effectively change the object. From Agora98 onwards, functional mixin-methods are called views and imperative mixin-methods are called mixins.

of the extension. The extension is the result of sending the 'view message'. Notice that views automatically yield parent sharing because all the views are extension of the same parent object.

Mixins. While views do not destructively change their receiver, mixins do. In the following example, sending `circle:` to the `point` object, really adds a `radius` variable and an `inCircle:` method to the original point. All objects that can access the point, can now also access `radius` and `inCircle:`.

```
point VARIABLE:
  [ x VARIABLE:0;
    y VARIABLE:0;
    circle:r MIXIN:
      { radius VARIABLE:0;
        inCircle:p METHOD:
          { ((p x) sqr + (p y) sqr) sqrt <= (SELF radius) }
       }
  ]
```

It is important to understand the difference between views and mixins. While views only put an extra inheritance layer around an object, with the original object as parent, mixins really change the object. Everyone refering to the object, including views, will notice that the object has been destructively extended. Mixins thus allow one to change an entire object hierarchy in one stroke. This shows that parent assignment *is* possible, without adding an extra parent assignment operator to the language.

Cloning Methods. Until know we have argued that mixin methods in the incarnation of mixins and views show how inheritance is possible without introducing extra language operators on objects. Here we show that the same is true for cloning. Agora does not feature a security-breaking (see section 3.6.3) cloning *operator* that is applied *on* objects. Instead, a cloning method must be used to clone objects. A cloning method is installed by sending the message `CLONING:` to a pattern, just like an ordinary method is declared. Upon invocation of a cloning method, *its body is executed in the context of a clone of the receiver* instead of the context of the receiver itself. The following example illustrates this. It shows a `new` cloning method in the `listnode` object. Upon sending `new` to the `listnode`, the `next` and `elmt` variables in the copy are initialised to `null`. By default, the result of a cloning method is that copy and `SELF` in a cloning method refers to the copy.

```
listnode VARIABLE:
   [ next VARIABLE: null;
     elmt VARIABLE: null;
     new CLONING:
      { SELF next:null;
        SELF elmt:null
      }
   ]
```

Some Other Reifiers. Figure 4.3 gives an overview of some other frequently used reifiers. We refer to the language manual [De 98b] for more details.

`SUPER`	Forwards a message to the parent object.
`TRY:CATCH:`	Catches an exception when needed.
`RAISE`	Raises an exception.
`IFTRUE:IFFALSE:`	Tests a conditional and evaluates one of the branches.
`WHILETRUE:`	Leading-condition loop.
`UNTILTRUE:`	Trailing-condition loop.
`FOR:TO:DO:`	Bounded loop.

Figure 4.3: Frequently Used Reifiers

Agora and Extreme Encapsulation. It is important to note from the previous paragraphs that Agora is a full-fledged prototype-based programming language with inheritance and cloning. Nevertheless, objects are its only language values and message passing is its only built-in operation. Hence Agora is a non-trivial prototype-based language that adheres to the principle of extreme encapsulation. A commonly uttered criticism against Agora is that all the extensions and possible cloning algorithms have to be within the object at object creation time. This will be dealt with in section 4.6.2.

4.3.3 Local and Public Attributes

In section 4.3.1 we made the distinction between receiverless and receiverful messages. We stated that it is not very wrong to think of receiverless messages as function calls. Agora objects actually consist of a local part and a public part and accessing the local part is what receiverless messages are good for. Hence, a receiverless message expression of the form `msg:arg` will be searched for in the local part of the object. This means that an object can send two kinds of messages to itself. Receiverless messages are sent to its local part, while receiverful messages to `SELF` are sent to its public part. Note that `SELF` is itself a receiverless unary reifier message expression that returns the current receiver.

By default, attributes are added to the public part. In order to define an attribute locally, the unary reifier message `LOCAL` can be used. The default modifying unary message is `PUBLIC`. In the following example, the `elmt` variable is accessible to everyone while the `next` variable is only visible to methods declared inside the list node.

```
listnode VARIABLE:
   [ elmt VARIABLE: null;  // same PUBLIC VARIABLE
     next LOCAL VARIABLE: null
   ]
```

Agora's syntax really requires receiverless messages: indeed, if we did not have receiverless messages, the syntactic receiver of a message expression would have to be a message expression again, ad infinitum. In order to find a semantics for them, an alignment was chosen of access rules for private attributes and the semantics of receiverless messages. But unfortunately, this yields a very complicated semantics: since receiverless messages have to be thought of as lexically scoped function calls, this required us to align local parts of objects with their lexical scope, yielding a combination of two hierarchies (the inheritance and the scoping hierarchy) that does not appear to make sense. In section 5.9 we will show how Pic% fixes this problem.

4.3.4 The Semantics of Agora Expressions

In order to completely appreciate Agora, it is necessary to have a look at its semantics which is an adult version of the MiniMix semantics described in section 4.2. Again the parallel with Scheme will be drawn. Looking at the details of a Scheme evaluator [AS85], it essentially consists of the following ingredients:

- A memory of cons cells containing Scheme data structures and programs. Both are internally represented as Scheme cons cells.
- A procedure *eval* that can be applied to any Scheme list. *eval* dispatches over its argument list and calls the appropriate evaluation rule for it.
- An environment system binding names to their values. *eval* is parameterised by an environment with respect to which it evaluates the expression at hand. The environment is recursively passed down the evaluator.
- A procedure *apply* to apply a function to a suite of arguments. When this happens, the body of the function is evaluated with *eval* which might again consist of function applications handled by *apply*. This recursive game between *eval* and *apply* is the heart of the evaluator.

An Agora evaluator ought to be implemented in an object-oriented language[7]. This is part of the definition of Agora, just like functional list processing (i.e., applying procedures on lists) is an inherent part of a Scheme evaluator [De 98a]. But except for this difference, each of the above ingredients can be found back in Agora, albeit translated to their object-oriented equivalent:

- A memory of objects. These objects represent data structures (i.e., Agora objects) and parse trees (i.e., Agora programs). Hence, just as in Scheme, programs and data structures are represented in the same way.
- All Agora parse tree objects understand the message *eval*. While the Scheme *eval* dispatches over the expressions it has to evaluate, this dispatching is automatic in the Agora implementation by virtue of polymorphism: *eval* is sent to a parse tree.

[7]Notice that we have also created an evaluator in Scheme, but still, closures were used to implement the implementation structures. Closures can be regarded as objects.

- The *eval* message is parameterised by a context object that represents the environment in which the expression is evaluated. This context object recursively travels through the evaluator. It contains a reference to the 'current' lexical scope, the 'current' self, the 'current' parent etc.

- Each Agora object is internally represented by an implementation level object. This implementation level object understands a message *send* which takes a message and a list of arguments and produces another Agora object. We consider *send* as the object-oriented analogue of *apply*.

General Evaluator Architecture. In the same way that the execution of a Scheme program can be considered as a recursive interplay between *eval* and *apply*, the execution of Agora programs can be seen as an alternating interaction of *eval* and *send*. The following properties further elaborate on the fundamentality of *send* and *eval*:

- Seen through the eyes of the Scheme evaluator, functions are represented as an abstract data type for which *apply* is the *only* operation. Once *apply* is called, it can access the internal details of a function consisting of its formal parameters, its lexical environment and its body code. However, these constituents are invisible outside the function concept. In the same way, seen through the eyes of the Agora evaluator, *send* **is the only message for Agora objects**. This is the reflection in the evaluator of our **extreme encapsulation** principle.

- In Scheme, *apply* takes a function and a list of arguments. *apply* evaluates the body of the function using the environment of definition, augmented with new bindings of arguments to formal parameters. The same is true for the *send* message in Agora. *send* takes the name of the message to be sent together with a list of actual arguments. It uses *no* hidden arguments like self references, environments or other, but evaluates the method corresponding to the message in the context of the receiving object and the actual arguments.

The remainder of this section studies this architecture in detail. First, we summarize the abstract syntax. Then, Agora's object model and *send* are studied. Finally, we focus on *eval*.

Agora Abstract Syntax. Figure 4.4 gives an overview of the Agora *abstract* syntax. Of course, the difference between operator, unary and keyword messages is a mere lexical issue and thus not visible here.

Object Structures and Method Lookup. As we already explained, an Agora object consists of a local part and a public part. Internally, these are tied together by an object identity. Hence, every ex-nihilo created object has a reference to an object identity. The object identity has a reference to the public part and the local part of the object. Figure 4.5 exemplifies this. The reason for making a distinction between objects and their identities is amongst others to enable the evaluation of mixins, a topic dealt with in section 4.4. The

Basic Literals	b
Ex-nihilo Objects	$[e_1, \ldots, e_n]$
Grouped Expressions	$\{e_1, \ldots, e_n\}$
Ordinary Message Expressions	$e.m(e_1, \ldots, e_n)$
Reifier Message Expressions	$e.M(e_1, \ldots, e_n)$
Ordinary Receiverless Message	$m(e_1, \ldots, e_n)$
Reifier Receiverless Message	$M(e_1, \ldots, e_n)$

Figure 4.4: Agora Abstract Syntax

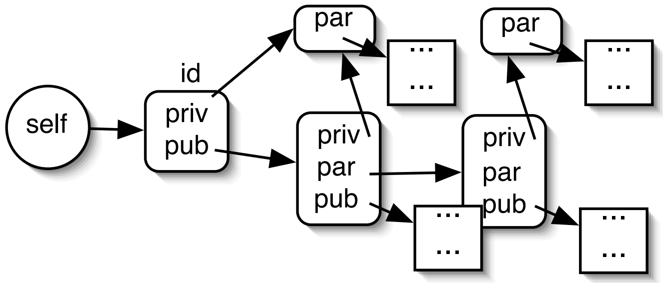


Figure 4.5: Agora Object's Structure

object itself is what message sending clients get to see. It implements the *send* message. Internally *send* is implemented by delegating the message through the chains of frames that constitute the object. These arise from successive applications of views and mixins, exactly as in MiniMix. Once the attribute corresponding to a message is found in this way, it is processed by sending *do*. While searching for the attribute, a context is gradually built up that contains the internal frames of the object. This context is necessary to evaluate the body of the corresponding method because *eval* is parameterised by a context. This is summarised in figure 4.6. When a message m is sent to an object o with arguments $a_1, \ldots, a_n$, o will refer to its own private identity generator *id*. From this identity it further delegates the message to the public generator. In each generator g, delegating the message consist of looking up the message in the method table. If it is found, the corresponding attribute is invoked by sending *do* to it. If it is not found, the message is delegated to the next generator in the chain. In this delegation process, a context c is passed around that contains references to the object itself, the identity generator, the public generator and the local generator. This context is used in *do* to evaluate the body code of the found attribute. Note that all the semantic rules in this chapter are given in an imaginary object-oriented programming notation. As we have said before, the fact that Agora is implemented in an object-oriented medium is inherent to the definition of Agora. We will explain this further in section 4.5.

It is important to understand *the difference between message passing and delegation in Agora*. The evaluator uses message passing to send messages to objects using the *send* operation which is only parameterised by the message name and the arguments. Delegation in Agora consists of implicitly traversing the internal generators of the object structure (following parent-of links formed by succesive invocation of view and/or mixin attributes) until the method is found. The distinguishing fact between message passing and delegation is the context object which contains the information about the object itself. In an object, *nothing* is allowed to enter an object except for the message pattern and the arguments. Delegation on the other hand, manipulates several extra 'hidden' arguments in the context. In terms of the analysis presented in section 4.2, message passing is defined on wrapped generators, i.e., objects. Delegation is defined on the frames constituting the object, i.e., generators. **The same way as in MiniMix, unwrapped generators never enter the language**.

$$o.send(m, a_1, \ldots, a_n) = o.id.pub.delegate(m, c, a_1, \ldots, a_n)$$
$$\text{where } c = \begin{pmatrix} \text{slf} & = & \text{o} \\ \text{loc} & = & \text{o.id.loc} \\ \text{pub} & = & \text{o.id.pub} \\ \text{id} & = & \text{o.id} \\ \text{par} & = & \text{o.id.pub.par} \end{pmatrix}$$

$$g.delegate(m, c, a_1, \ldots, a_n) =$$
$$\begin{cases} att.do(c[par \rightarrow g.par], a_1, \ldots, a_n) & \text{if } att = g.lookup(m) \\ g.par.delegate(m, c, a_1, \ldots, a_n) & \text{otherwise} \end{cases}$$

Figure 4.6: Agora Message Passing Operator

Evaluation Rules. Let us now turn to the evaluation rules for the syntax outlined in figure 4.4. These rules are given in figure 4.7, in which we use angular brackets $\langle$ and $\rangle$ to delimit parse tree objects.

- The evaluation rule for basic literals consists of creating a new basic object which understands *send* as its only implementation level message.
- Ex nihilo created objects are constructed by creating new public and local generators by extending existing frames (a.o. the root of the system).
- Evaluating a group of expressions in a context simply consists of evaluating all the expressions from left to right in that context.
- Evaluating an ordinary message $e.m(e_1, \ldots, e_n)$ is accomplished by evaluating the receiver and the actual arguments in the current context, and then sending the message using *send*.

$\langle literal\rangle.eval(c)$	Create new basic object understanding *send*
$\langle[e_1;\ldots;e_n]\rangle.eval(c)$	$\langle e_1\rangle.eval(c');\ldots;\langle e_n\rangle.eval(c')$ where $c' = \begin{pmatrix} loc & = & c.loc.addFrame() \\ pub & = & rootPublic.addFrame() \\ id & = & \text{new Identity}(pub, loc) \\ par & = & rootPublic \\ slf & = & \text{new Object}(id) \end{pmatrix}$
$\langle\{e_1;\ldots;e_n\}\rangle.eval(c)$	$\langle e_1\rangle.eval(c);\ldots;\langle e_n\rangle.eval(c)$
$\langle e.m(e_1,\ldots,e_n)\rangle.eval(c)$	$\langle e\rangle.eval(c).send(m, \langle e_1\rangle.eval(c),\ldots,\langle e_n\rangle.eval(c))$
$\langle m(e_1,\ldots,e_n\rangle.eval(c)$	$c.loc.delegate(m, c, \langle e_1\rangle.eval(c),\ldots,\langle e_n\rangle.eval(c))$
$\langle e.M(e_1,\ldots,e_n)\rangle.eval(c)$	$adHocEval(c, \langle e\rangle, M, \langle e_1\rangle,\ldots,\langle e_n\rangle)$
$\langle M(e_1,\ldots,e_n)\rangle.eval(c)$	$adHocEval(c, M, \langle e_1\rangle,\ldots,\langle e_n\rangle)$

Figure 4.7: Evaluation Rules (part 1)

- As explained in section 4.3.3, receiverless messages are looked up in the local generator of the object. This local generator can be read from the evaluation context because it was put there when *send* was invoked.

- The final syntactic category to be evaluated are reifier messages and receiverless reifier messages. As explained earlier, reifier messages are the object-oriented analogue of special forms. They are thus handled in an ad-hoc manner. For instance, the ad hoc evaluator for the `VARIABLE:` reifier message evaluates the initial value in the current context, ands installs a read and a write slot in the public generator that resides in that context.

4.4 The Power of Attributes

This section shows **the very essence of the Agora model**: i.e., how it is possible to have a full-fledged prototype-based programming language with features like dynamic object extension and cloning, while adhering to the principle of extreme encapsulation which dictates that objects should be subject to message passing and message passing alone.

As we have seen in figure 4.6, a message to an Agora object generates a method lookup process that finally results in $do(c, a_1, \ldots, a_n)$ being sent to the attribute corresponding to the name of the message. This is where it all happens. Just like MiniMix, Agora features different kinds of attributes. Let us briefly go through figure 4.8 which gives the semantics for each kind of attribute discussed in section 4.3.2. Apart from the variable accessor methods, the technique used is always the same. When *do* is sent to an attribute object with a given context c, a new context c' is constructed in order to evaluate the body expression of the attribute. In each case, the local generator is extended to contain the formals-actuals bindings. In the *do* for cloning methods, a copy of all the generators is created and a new object is made with the copies. The body of the method is evaluated in the context of the copied generators. Particularly interesting is the difference between mixins and views. In both cases, generators are extended. In the case of views, a new object is created in which the body expression is evaluated. A view frame is attached to the *id* such that later mixins on the parent will affect that view. In the case of a mixin, the given object identity is provided with the extended public and local generator, but the object and the object identity stay the same. Thus, the difference between views and mixins is the motivation distinguishing between objects and their identities.

$varGet.do(c) =$	return contents of the variable
$varSet.do(c, a_1) =$	assign variable to a_1
$method.do(c, a_1, \ldots) =$	$method.bodyCode.eval(c')$ with $c' = \begin{pmatrix} loc & = & c.loc.addFrame(a_1, \ldots) \\ pub & = & c.pub \\ id & = & c.id \\ par & = & c.par \\ slf & = & c.slf \end{pmatrix}$
$cloning.do(c, a_1, \ldots) =$	cloning.bodyCode.eval(c') with $c' = \begin{pmatrix} loc & = & c.loc.copy().addFrame(a_1, \ldots) \\ pub & = & c.pub.copy() \\ id & = & \text{new Identity}(pub, loc) \\ par & = & c.par.copy() \\ slf & = & \text{new Object}(id) \end{pmatrix}$
$view.do(c, a_1, \ldots) =$	view.bodyCode.eval(c') with $c' = \begin{pmatrix} loc & = & c.loc.addFrame(a_1, \ldots) \\ pub & = & c.id.addFrame() \\ id & = & \text{new Identity}(pub, loc) \\ par & = & c.id \\ slf & = & \text{new Object}(id) \end{pmatrix}$
$mixin.do(c, a_1, \ldots) =$	mixin.bodyCode.eval(c') with $c' = \begin{pmatrix} loc & = & c.loc.addFrame(a_1, \ldots) \\ pub & = & c.pub.addFrame() \\ id & = & c.id.assign(pub, loc) \\ par & = & c.pub \\ slf & = & c.slf \end{pmatrix}$

Figure 4.8: Agora Attribute Invocation

As we have already shown in our presentation of the MiniMix model, this is the heart of the Agora paradigm: instead of defining all kinds of operators on objects such as cloning, slot addition and deletion, parent assignment and so on, attributes of an object are evaluated in the context c containing the internal details of the object in which the attribute was found. Each kind of attribute knows what to *do* with these details. Stated differently, Agora objects are extremely encapsulated (due to *send*) but upon invocation of a message, attributes can access the internal generators of the object they reside in. By cleverly using these internal generators, extensions and clones of the receiver can be returned by the attributes. This model can be extended: the more complex the internal structure of objects is, the more types of attributes that can be invented. This will be exploited in chapters 8 and 9 to add distribution and mobility language features to the basic model.

Conclusion: As shown in section 3.5, other prototype-based languages implement their language features by enriching their object model with several operators such as cloning and slot addition and deletion. In our theoretical analysis of chapter 3, this was called a 'change in object model' as it required objects to be represented as generator. Agora shows that a full-fledged prototype-based language can be built that has much of the flexibility of other prototype-based languages but that inherits the extremely encapsulated object model of class based languages. **The key to the solution is the fact that objects can react differently to messages depending on the type of attribute with which the message happens to be implemented** [DMS96].

4.5 Adding Reflection: Language Symbiosis

As promised in section 4.3, this section extends Agora with reflection operators. Wand and Friedman [WF88] have shown that a language can be made reflective by reifying implementation level structures into the language, and absorbing language values back into the evaluator. They have investigated this in the context of Brown, a fully reflective variant of Scheme. It is one of Agora's merits to transpose their reflection model to object-orientation in a clean way.

4.5.1 Reification of Implementation Objects

In order to talk about reflection, it is necessary to distinguish two levels in the Agora semantics. The 'down' level is the level of the (object-oriented) implementation language such as Java, Smalltalk or C++. The 'up' level is the Agora level being evaluated by the 'down' level. Using more standard terminology, the 'down' level is the meta level and the 'up' level is the base level.

The evaluation rules of figure 4.7 show that Agora actually knows two kinds of Agora objects, namely basic literals, and ex nihilo created objects that consist of chained generators. As can be seen in the evaluation rule for basic literals, Agora also represents them as objects that understand the *send* message. These objects are actually wrapped versions of their corresponding 'down' object. For example, the Agora literal `3` is represented by wrapping the corresponding (Java, Smalltalk or C++) object 3. This wrapping process is accomplished by sending the *up* message to the implementation level object. Hence, $\langle 3\rangle.up()$ is an Agora object that understands *send*. The implementation of *send* for upped objects will map every message onto the corresponding message at the down level. This is accomplished by bringing both the receiver and the arguments to the down level. After *actually* sending the message, the resulting down level object is brought back to the up level by sending *up*. This addition to the message passing operator of figure 4.6 is shown in figure 4.9. The evaluator therefore knows two kinds of objects with the same *send* interface. However, due to extreme encapsulation (i.e., only *send* is possible), the evaluator is unable to distuingish between these two kinds of Agora objects: only the objects know whether they are ex-nihilo created or upped implementation level objects.

$$\boxed{o^u.send(m, a_1, \ldots, a_n) = o^u.down().m(a_1.down(), \ldots, a_n.down()).up()}$$

Figure 4.9: Agora Message Passing Operator for Upped Primitives o^u

4.5.2 The Evaluator Reconsidered

The idea of upping implementation level objects can not only be applied to primitive literal objects, but to *all* implementation level objects. In each Agora implementation, all implementation level objects understand the message *up*[8]. This can be used to give a much cleaner semantics to reifier messages. Instead of handling a reifier message in an ad-hoc manner, the model with *up* treats reifier messages as real Agora messages to upped syntax objects. Together with the treatment of literals as discussed in the previous section, this gives us figure 4.10 which is an improved version of the evaluation rules outlined in figure 4.7. As we can see in figure 4.10, reifier messages are no longer implemented in an ad-hoc fashion. Instead they are really sent to the reified (i.e., upped) versions of their syntactically appearing receiver. This is why messages like `VARIABLE:` were called reifier messages in the first place: they are reifications of the corresponding messages defined on the implementation level objects that represent parse tree nodes. The advantage of processing reifier messages this way is that by the late binding of reifier messages (i.e., they are really looked up in the upped object), we can install our own reifiers which turns Agora into a reflective language. This is also the justification why Agora must be implemented in an OO medium.

$\langle literal \rangle.eval(c)$	$\langle literal \rangle.up()$
$\langle e.M(\ldots, e_i, \ldots) \rangle.eval(c)$	$\langle e \rangle.up().send(M, c.up(), \ldots, \langle e_i \rangle.up(), \ldots).down()$
$\langle M(\ldots, e_i, \ldots) \rangle.eval(c)$	$c.loc.delegate(M, c.up(), \ldots, \langle e_i \rangle.up(), \ldots).down()$

Figure 4.10: Evaluation Rules (part 2)

In figure 4.11, we illustrate the computational process induced by sending the reifier message `x VARIABLE:3`. The applied rules are the ones for evaluating

[8]In each Agora implementation, we had to come up with a different technical trick to make sure every implementation level object understands *up*.

a reifier message (figure 4.10) and the ones for mapping Agora messages on upped objects onto their corresponding implementation level message (figure 4.9). When encountering the message `x VARIABLE:3`, the `VARIABLE:` message is sent to the upped version of `x` with the upped version of `3`. Of course, the result must be brought back to the evaluator level by sending it *down*. Because of the message passing operator outlined in figure 4.9, this means that the `VARIABLE:` Agora message will be mapped onto the implementation level message *variable* defined on identifiers (or more precisely: receiverless unary patterns).

$$\begin{array}{ll}
 & \langle x.VARIABLE(3)\rangle.eval(c) \\
= & \langle x\rangle.up().send(VARIABLE, c.up(), \langle 3\rangle.up()).down() \\
= & \langle x\rangle.up().down().variable(c.up().down(), \langle 3\rangle.up().down()).up().down() \\
= & \langle x\rangle.variable(c, \langle 3\rangle).up().down() \\
= & (\text{install slots `x' and 'x:' in } c.pub \text{ bound to } \langle 3\rangle.eval(c)).up().down() \\
= & (\text{return } \langle 3\rangle.eval(c)).up().down() \\
= & (\text{return } \langle 3\rangle.up()).up().down() \\
= & \langle 3\rangle.up().up().down() \\
= & \langle 3\rangle.up() \text{ (i.e., the Agora object 3)}
\end{array}$$

Figure 4.11: Example of Evaluating `x VARIABLE:3`

One of the distuinguishing features of reifier messages is that they are dynamically scoped: the first argument of each reifier method is the context in which the reifier message occurs. This is in contrast to ordinary methods that are completely lexically scoped. Parameterising reifier messages with their context of invocation is not some 'dirty' trick but completely follows the spirit of special forms in Scheme. When evaluating the `(if ... ... ... )` special form, the expressions have to be evaluated in the context where the special form occurs, and not in the context of definition of the 'if' procedure.

4.5.3 Absorption of Ex-Nihilo Objects

In the previous sections we have seen that any 'down' level object can be reified in Agora by sending *up* and that an upped object can always be brought back to the 'down' level by sending it *down*. The final step for full reflection is to extend the *down* mechanism for ex nihilo created objects as well. This is called absorption. Absorption allows one to down an ex nihilo created Agora object into the implementation, such that the implementation can send (implementation level) messages to it. Each Agora implementation uses its own technical trick to accomplish this, depending on the implementation language. In most cases, this was far from trivial to realize. In Java for example, we have to generate class files dynamically in order to wrap an Agora object in a Java object. Each message sent to this native Java object must be mapped onto the corresponding message in Agora: if $o^d.m(a)$ is sent in Java, and o^d is a downed Agora object, the message must be resent in Agora yielding $o^d.up().send(m, a.up()).down()$.

Hence, when sending implementation level messages to downed Agora objects, the receiver and all arguments must be upped. Then the message must be sent to the Agora object using *send*. The resulting Agora object must be brought back to the implementation language using *down*. The technique is summarised in figure 4.12. It allows us to replace implementation level objects (i.e., objects of the evaluator) by our own objects written in Agora. We show how to use this technique in section 4.6.2.

$$\boxed{o^d.m(a_1, \ldots, a_n) = o^d.up().send(m, a_1.up(), \ldots, a_n.up()).down()}$$

Figure 4.12: Message Passing For Downed Agora Objects

Combining the message passing operators of figures 4.6, 4.9 and 4.12 yields the implementation of *up* and *down* as shown in figure 4.13. Upping a downed up level object o^u consists of returning the original up level object o^u. Upping a non downed object consists of creating a new up level Agora wrapper for it. The same mechanism is used for *down*. As already mentioned, each Agora implementation uses its own technical trick to implement these rules.

$$\forall\, o : o.up() = \begin{cases} \text{new } UpWrapper(o) & \text{if } o \text{ is not a downed object} \\ o^u & \text{if } o = o^u.down() \end{cases}$$

$$\forall\, o \text{ understanding } send : o.down() = \begin{cases} \text{new } DownWrapper(o) & \text{if } o \text{ is not a upped object} \\ o^d & \text{if } o = o^d.up() \end{cases}$$

Figure 4.13: The Implementation of Up and Down

This technique for establishing a language symbiosis between Agora and its implementation language has proven its usefulness in other settings as well. In [GBD02] and [GD02] we show its applicability in the context of a meta language (Prolog) that reasons about a base language (Smalltalk). The same configuration is used to connect a forward-chained meta language that reasons about Smalltalk in [DDGD01]. Finally, in [DDW99] we show its applicability in the context of aspect languages that reason about a base program. In all cases two language layers are involved and objects can float freely from one language to another.

4.5.4 The Agora MOP: Extreme Encapsulation

The system of 'upping' and 'downing' objects implies that *all objects* in the implementation understand *up*, and that *all Agora objects* understand *send*, *up* and *down*. These mechanisms form the *meta object protocol* of Agora and can be regarded as the object-oriented analogue of the meta functions used in Brown [WF88]. In Brown, function and special form application are seen as the only control structures[9], and the operators $\vee$ and $\wedge$ are the conversion operations between the base and the meta level. In the same way, message passing and reifier message passing (with hidden context argument) are the only control structures in Agora. *up* and *down* are used to switch objects consistently between the base level and the meta level[10].

4.6 Agora Model: Evaluation and Epilog

In the previous chapter we have shown that prototype-based languages are no panacea. Two critiques have been developed against classic prototype-based languages. From a software engineering point of view, we have identified many shortcoming which are more tangible and measurable criteria for the sentiment prototype-based languages usually evoque: they are too flexible. From a language theoretical analysis, we have learned that prototype-based languages suffer from inherent encapsulation and security problems. Both analyses have been traced back to the single fundamental problem of prototype-based languages in section 3.5: prototype-based languages feature only objects but define a plethora of language operators on these objects and this is what renders them too flexible and unsafe. We have therefore identified the extreme encapsulation principle which states that message passing should be the only language operation applicable to objects. The Agora model has shown that adhering to this principle does not imply that we are condemned to classes or uninteresting languages with only objects and methods. Now that we have described the model in deep detail, let us look at its implications for the problems we identified.

4.6.1 General Conclusion

In this chapter, we have shown how a 'very Scheme like' object-oriented prototype-based programming language can be constructed with a meta object protocol that only offers *send*, a property we referred to as extreme encapsulation. Although not possible at first sight, such a semantic architecture still allows features like cloning and object extension by a clever use of special purpose attributes that can access an object's internal details. As we have explained in section 3.5 other object-oriented programming language either introduce differ-

[9]In Brown, special forms are also called reifier functions.

[10]Note that implementing *down* directly on Agora objects is not strictly necessary because we can replace it by *send*('*down*') which is supposed to invoke a 'downing method'. However, for efficiency reasons we implemented *down* directly on Agora objects.

ent language *values* like classes to accomplish this, or, take only objects, but then define a multitude of meta level *operations* on these objects.

In the second part of the chapter we have shown how adding the *up* and *down* message to our meta object protocol extends the basic language to a fully reflective prototype-based language. To some people this seems strange: if one cannot do more at the meta level than at the base level (i.e., message passing), then why add a meta level in the first place? Agora refutes this criticism by consistently mapping every base level message onto a meta level message, and the other way around. As such, every implementation level message can be intercepted (and thus reprogrammed!) at the Agora level. This makes Agora a fully reflective language kernel in which both structural and behavioural reflection are possible. We will further elaborate on the reflective capabilities of Agora in the rest of this section.

4.6.2 Extension "from the outside"

In this dissertation, we will not go into the details of how reflection in Agora can be used to extend the language from within itself. This was discussed extensively in Steyaert's PhD thesis [Ste94]. Nevertheless an insight we have developed during our research is that the principle of extreme encapsulation does not necessarily preclude the extension of objects "from the outside".

Every Scheme programmer is familiar with quoting, a mechanism to transform a program into a data structure. In Agora, quoting an expression is accomplished by sending it the `QUOTE` reifier. The result thereof is the expression itself in the form of an Agora object. That is, sending `QUOTE` reifies the underlying parse tree as an object in Agora (i.e., an object understanding *send*). Internally, this is accomplished by sending *up* to the expression. Hence, internally, each expression object understands the reifier message $quote(c)$, just like identifiers understand the message $variable(c, expression)$. The implementation of *quote* is to return the receiver as an upped object, i.e., "$this.up()$" yielding an Agora object that understands *send*.

The opposite of quoting is called unquoting. In Agora, this is accomplished by sending `UNQUOTE` to an expression. The receiver of this reifier must evaluate to an expression object. The resulting expression object will be downed (yielding a real expression understanding *eval*) which can then be evaluated in the context of unquoting. Hence, the implementation of the $unquote(c)$ reifier on expressions is "$this.eval(c).down().eval(c)$".

These two reifiers are particularly interesting for extending objects "from the outside". Extension from the outside is accomplished by a view or mixin that does not list the new slots itself, but takes a parameter being the expression to be evaluated in the view. By unquoting this parameter, the parameter is evaluated (yielding a quoted expression), and this expression is then evaluated in the context of the view or mixin:

```
mySlots  LOCAL VARIABLE: ({ ...slots ...} QUOTE);
myObject LOCAL VARIABLE:
```

```
                 { ...
                   extend:slots VIEW: (slots UNQUOTE)
                 };
      myObject: (myObject extend:mySlots)
```

But if we can extend objects "from the outside", then what about extreme encapsulation? The answer is that objects are always extremely encapsulated, but *can* be extended from the outside if *they* want to. In other prototype-based languages, objects *cannot* avoid being subject to extension because the extension operators are applied *to* the object in such a way that the object cannot play a role in the process. Hence, the Agora model shows that a language can be created that is acceptable from a software engineering point of view and that adheres the extreme encapsulation property.

4.6.3 Reflection Protection

The ability of the language to guarantee reflection protection is extremely important. We speak about reflection protection when a language does not allow us to bypass "base level protection mechanisms" (such as extreme encapsulation) through the reflection operators. Agora has full reflection protection because the meta level essentially does not contain anything more than the base level does. Although meta level programming and reflection in Agora offer the same power other reflective systems have, the meta level does not allow programmers to bypass the restrictions of the base level. Note that this reflection protection is really important. In Java e.g. the fact that objects can be dissected at the meta level is the source of many security breaches.

4.6.4 Agora: Intersecting Classes and Prototypes

In the introduction of this chapter, we claimed to introduce an intersection between class-based languages and prototype-based languages. Let us now explain why we feel the Agora model is indeed such an intersection in the sense explained in section 1.2 of the introduction of the dissertation.

First there is the language theoretical point of view delivered by the denotational semantics we developed in section 4.2. It was shown by Cook that a satisfactory denotational semantics for class-based languages consists of a record based object model and a generator-based inheritance model. In class-based languages these are clearly separated from each other and generators never interfere with the realm of objects and their message-based way of accessing them. In prototype-based languages, on the other hand, the inheritance mechanism requires objects to be generators "with a self hole in them". The MiniMix and Agora model are indeed intersections of class and prototype-based language because they inherits the semantical possibilities of prototype-based languages (incorporated in the inheritance and other operators discussed in sections 3.4 and 4.4) and the record-based object model of class-based languages. This is possible by allowing objects to access and manipulate their own (but only their own!) generator. But objects themselves are the only language values and message passing is the only operation possible on objects. We therefore can say that the Agora model inherits the object model from class-based languages and the language operators of prototype-based languages.

Also from a software engineering point of view, the Agora model is the middle ground between class-based languages and prototype-based languages. Although the model does not have classes, a lot of the properties of class-based language are inherited due to the strict adherence to message-passing as the way to accomplish inheritance and object creation. As such, inheritance and object creation can only be accomplished if some "predefined constructions are installed at the right place". In conventional prototype-based languages this is not the case: because of the many language operators defined on objects and because objects have no active role in when and how these operators will be applied to them, inheritance and object creation are completely unstructured. Let us have a look at the implications of this on the software engineering problems of prototype-based programming languages we identified in section 3.3:

- We identified a flexibility problem due to the fact that objects can be cloned in the **absence of a construction plan** leading to unfinished objects. This is easily solved in Agora. The only way to create objects is by listing all their attributes, or by sending them a message that is handled by a cloning method or a view method. The cloning method or the view method that delivers the new object (i.e., the clone or the extension) can ensure that the delivered objects are always "complete" and meaningfully initialized. And since message passing is the only operation, this pre-programmed construction plan cannot be circumvented. Of course, programmers can still implement buggy cloning methods or view methods. This is the same as constructors with bugs in a class-based language. But "half cloning" an object is a bug that is not easily made in our model.

- The **prototype corruption problem** consists of the fact that it was possible to accidentally modify prototypes such that the modifications have repercussions on the clones delivered later on in the code. Our encapsulated cloning mechanism easily refutes this critique. By programming cloning methods in the right way, delivered clones are always correctly initialized. After all, cloning methods act like constructor methods in class-based languages. E.g., the cloning method to clone strings can ensure that the clone delivered is always initialized to be the empty string.

- The **varying templates problem** basically stated that it is very dangerous to change the parent of an object destructively, both from a flexibility as well as from a security point of view. In the Agora model, the parent object of an object is a frame (or generator) and not a real object. As such it is impossible to change the parent generator of an object because this would imply that such generators become first class in the language.

- Finally, we identified the problem of **reflection protection** which boils down to the fact that, by allowing reflection operators to be defined directly on objects, it is possible to circumvent the base level restrictions a language imposes on objects. As shown in section 3.6, this can have disastrous effects in objects in open networks. In Agora, it is impossible to access more at the meta level than at the base level. Nevertheless, our experiments with extensions from the outside and Steyaert's PhD thesis show that this does not mean that reflection is not useful when it adheres to the reflection protection criterion.

Note that two additional problems cited in section 3.3 are **not solved** by Agora. First, there was the problem of some concepts being **inherently abstract**. This is not solved more by Agora than by other pure prototype-based languages. It is inherent to the pure prototype paradigm. In the following chapter we will present a contemporary multi paradigm language that has both procedural and prototype-based characteristics. The presence of functions in this language will prove to be a solution to this problem as functions can act as constructors for objects belonging to inherently abstract concepts. Hence, these functions will play the role of classes. Nevertheless, the functions are not really an addition to the language as they are implicit self-sends. Another problem addressed in section 3.3 was the **re-entrancy problem** which boils down to the fact that code-reuse (re-entrancy of code) and the modeling hierarchy are often orthogonal to one another. This is also not solved by Agora. On the contrary, Agora makes the problem worse as it does not feature multiple inheritance, nor does it contain a complicated comb-like inheritance as in NewtonScript to combine the need for both traits and the modelling hierarchy. So programmers are to structure their hierarchies solely based on "the right" combination of view methods. The language proposed in the next chapter will solve this problem.

4.6.5 The Treaty of Orlando Revisited

In section 2.5 we have presented the Treaty of Orlando as a general taxonomy of object-oriented programming languages. The Treaty of Orlando classifies object-oriented languages according to how they fill in the notions of objects, message passing, templates and empathy. In a class-based language, templates are classes and empathy is class-based inheritance. Cookie-cutting is instantiation. In a prototype-based language, objects *are* templates and the two types coincide. We like to refer back here to figure 2.3 that reflects this schematically.

Now that we have explained the MiniMix theoretical model and the Agora language family derived from it, we have to adapt this picture slightly. The analysis of this chapter has clearly shown that a third concept is needed to correctly classify object-oriented programming languages. It appears that, apart from objects and templates, an object-oriented programming language is also largely determined by the way objects can relate to (and "perceive") themselves. We will refer to this as the *self representation model* and we argue that its understanding is crucial to object-orientation[11]. Indeed, existing class-based

[11]This can already be noticed from the way OOP languages handle recursion, i.e., by mes-

languages such as Java and C++ actually already reveal the fact that there is a need for a self representation model to classify languages: objects are granted different access rights to themselves than to other objects. Proof of this are different access modifiers (like `protected` and `private`) these languages are equipped with. They clearly show that the semantic domain behind `self` or `this` is not necessarily the same as the semantic domain used to model the actual objects. For example, in Java, `this` is not the actual object because calls such as `this.do()` will succeed even when `do` is a private method. Another example that shows the relevance of the self representation model is the distributed language Obliq discussed in section 7.5.3. It will appear that some Obliq objects can apply more operators on themselves than other objects.

The study of inheritance in prototype-based languages and MiniMix indicates that this self representation model can be made much more important than one would expect from existing languages. Including the self model in the Treaty Of Orlando gives us the model of figure 4.14. The figure shows an improved version of 2.3. The figure shows how the template model has a self reference model which it uses to define empathy. Empathy is defined on the self representation model and not on the empathy model. This is also the case in class-based languages where access to protected superclasses attributes is allowed during inheritance. The objects are still created by cooking cutting a template, but it will refer to its self representation model to accomplish this. Furthermore objects refer to their self reference model through the template from which they were cookie-cut. E.g. in a class-based language, an object can access more from itself than from other objects precisely because the access is funneled through the class declaration properties. The Revised Treaty of Orlando is a proper extension of the original one in the sense that it still classifies existing language in the same way. However, if we try to align the object domain with the template domain (which is what defines a prototype-based languages, according to the original Treaty), we now have two options:

- Either we also align the object domain with the self representation domain. In that case, we get a classical prototype-based language.

- Or we explicitly do not align the self representation domain with the object domain. In that case, we get the Agora model because the object model will coincide with the template model and thus the object model will *have* a self representation model without objects *being* their self representation model. It is exactly this insight that was used in Agora: objects adhere to the extreme encapsulation principle and only define message passing. However, objects can refer to their self representation model thereby delivering extensions (empathy) and clones (cookie-cutting).

sages to self. The self representation model is included in *all* formal treatments of OOP. Another hint is the importance of `MyType` in type-theoretical treatments of OOP.

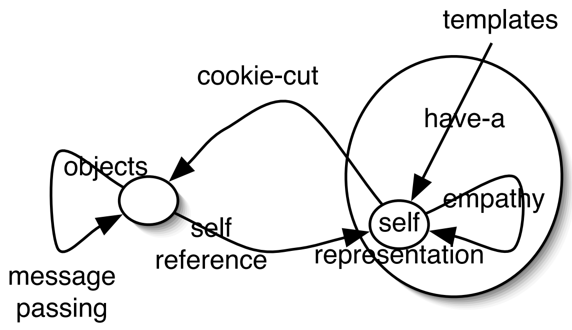


Figure 4.14: The Treaty of Orlando Revisited

4.7 Conclusion

The analysis presented in chapter 3 provided us with an apparent stalemate. On the one hand, class-based languages have fundamental paradigmatic problems in order for them to be deployable in the context of open networks and distribution. The reason is that classes establish an implicit relationship between objects that becomes painfully explicit. Prototype-based languages solve this by sticking to idiosyncratic objects. The consequence is that all the interesting language features (which we called language operators) are defined on objects. This yields unacceptable security problems. We have therefore introduced the extreme encapsulation and the reflection protection principles.

In this chapter, we have shown that the principles *can* be adhered to by consistently sticking to the "objects+message passing" boundaries, but by allowing the language processor to pass around hidden information about the "self" of an object behind scene. A number of well-designed types of methods (e.g. view methods, mixin methods and cloning methods) can access this hidden information to allow all kinds of special operations to be performed on the receiving object. But this information is only available for the "current receiver" and not for other objects. The result of those methods is always a fully encapsulated object that cannot be tinkered with. Even when moving to the meta level, objects only reveal a "send" method that can be used to send messages to them. This corresponds to reflection protection.

The Agora model shows that by cleverly rethinking the principles of prototype-based languages, a family of programming languages can be distilled that inherits the advantages from both paradigms but avoids there disadvantages. The overal result is a full-fledged secure prototype-based programming language with powerful language features like inheritance and cloning.

Chapter 5

Pic%: A Contemporary Agora Descendant

Chapter 4 presented the Agora model as an intersection of class-based and pure prototype-based programming. Agora is classless and does not suffer from the encapsulation problems outlined in chapter 3. However, Agora has many problems when it comes to the practical applicability of the language. We will present a descendant Pic% that overcomes them but still endorses the principles of extreme encapsulation and reflection protection principles that were presented as a yardstick to measure prototype-based languages

5.1 Introduction

This chapter introduces Pico and Pic%. Pico is a very tiny Scheme like language that was designed for educational purposes. Pico might be considered as a simplification of Scheme [DDD04]. Pic% is a prototype-based extension of Pico (called Pico - hence the name Pic-oo or Pic%). Central to the design and implementation of both Pico and Pic% is simplicity. Moreover, Pic% was designed following the Agora principles outlined in chapter 4. In chapter 8 we further extend Pic% with concurrency and distribution facilities and chapter 9 adds mobility constructs. The resulting language is called ChitChat. Figure 5.1 shows how the languages have fertilized each other.

Pic% is a contemporary prototype-based language in the sense that it incorporates the scientific knowledge of prototype-based languages and that it solves the problems and avoids the critiques put forward in the previous chapter. To this extent, Pic% is heavily based on the Agora model. However, Pic% also avoids a number of drawbacks of the Agora model. As we will see in the following section, most drawbacks of Agora can be reduced to the fact that Agora is all but a lightweight language. It contains quite a number of concepts and implementing Agora is a non trivial task mainly due to its uniform treatment of reflection.

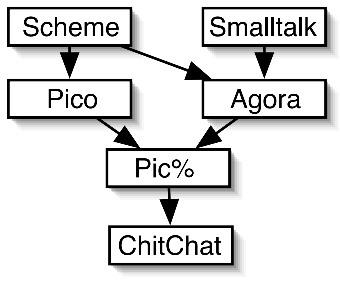


Figure 5.1: The Language Tree in this Dissertation

The following section starts by listing the major drawbacks concerning Agora's size and complexity. Section 5.3 clarifies the origin of Pico. Section 5.4 enumerates the design options we have at our disposal to simplify the Agora model. Section 5.5 introduces Pico which was the initial inspiration for Pic%. Section 5.8 explains how Pic% incorporates the Agora ideas in Pico.

5.2 Problems with Agora

The Agora model originally proposed Steyaert's [Ste94] and semantically analysed in chapter 4 at first sight seems to satisfy our requirements outlined in chapter 3 perfectly. Agora is powerful classless language that offers all the security needed. Furthermore, Agora's ideas are *deep*. This was shown in section 4.3.4 where we presented Agora as the Scheme of object-orientation. However, aside from this language theoretical positive news, Agora as a concrete programming language also has a number of important drawbacks:

- Agora has many **problems concerning syntax**. Most people find the Smalltalk-like syntax hard to grasp. Although language theorists are fond of regular syntactic schemes such as those of Lisp, Scheme, Smalltalk, Self and Agora, the average programmer seems to have a lot of problems in fully grasping it[1]. Because of cultural reasons, people seem to be better at reading infix-notation-based programs. We believe that syntax was one of the major reasons why Java was so easily accepted by industry: although Java has more in common with Smalltalk, it looks like C++.
- Agora requires **advanced programming environments** that support rich text as the lexical difference between ordinary messages and reifier messages is made based hereupon. For example, Steyaert distinguishes reifier messages from ordinary messages by their boldfaced appearance. Unfortunately, "richer text formats" such as colors and boldfaced are still not accepted as part of the lexical layer of the definition of a language.

[1]In our own computer science programme, it is our experience that it takes students more than a year to fully grasp and oversee the implications of Scheme's regular syntax.

- A true problem is that **Agora features no less than 12 types of messages** which all look alike but the semantics of which is subtly different. They vary along the three dimensions { receiverless, receiverfull }, { keyword, unary, operator } and { ordinary, reifier }. For example, in section 4.3.3 we explained how the interaction of receiverless messages, lexical scoping and the complex internal structure of objects yield scoping rules which are very hard to grasp, even for specialists in the field.

- One of the biggest problems of Agora is that it is **all but a lightweight** model. The reflection model based on a linguistic symbiosis between Agora and its implementation language using *up* and *down* puts high demands on the implementation. Indeed, the example in section 4.5.2 shows how a variable is declared. Without the technical housekeeping messages that were discarded for the sake fo clarity, the example shows that a simple operation such as declaring a variable requires at least 13 messages being sent by the evaluator. Also, the "pointer plumbing" of its object structure (see 4.3.4) requires a lot from the implementation, both from a memory consumption point of view and from a processor speed point of view during variable and method lookup.

- Despite the very powerful reflective architecture of Agora, as a language **it is extremely hard to extend**. As we have shown in [De 98b], writing new receiver*less* messages is not that hard because the new reifier message will be a lexically scoped "function". It is the same as programming a new macro in Lisp: when one uses the macro, it is looked for in the scope.

  However, it is not possible to program new receiver*ful* messages. For example, suppose we would like to design a new receiverful message `PRIVATEVAR:` to be used in expressions like `x PRIVATEVAR:4` to install a new private variable named `x` and initialized to 4. In order to do this, we have to express in Agora that "from now on, every identifier has to understand the `PRIVATEVAR:` message". But this is *really* problematic for two reasons. First, identifiers, like all objects, are "encapsulated implementation level objects" that cannot be extended. Indeed, e.g. in the Java implementation, identifiers are represented as Java objects which are not dynamically extensible[2]. So it is not possible to add behaviour to identifiers from within Agora. In [Ste94], Steyaert "solves" this problem by making a distinction between dynamic reflection and static reflection: the newly defined reifiers could statically be compiled into the implementation before it is run. The second problem we have with the construction of new reifier messages is that this is an inherently class-based activity! Indeed, when programming the meaning of `PRIVATEVAR:` we want to specify this code once *for all identifiers*. But since there is no such thing in the Agora language this is a really problematic thing to solve. Traits, e.g. are no option as there is no such thing at the implementation level.

[2]In Smalltalk, this is possible because of the reflective capabilities.

- In the previous chapter we have identified a number of problems of conventional prototype-based languages. We have solved a number of those problems (especially the encapsulation problem) with the Agora model, but as explained in section 4.6.4 two problems remain, namely the problem that some things are **inherently abstract** and thus fit poorly in the prototype-based philosophy, and **the re-entrancy** problem. Just as with Agora's reflection problem, these two problems seem to point towards an inherent need for some class-like construction.

To alleviate these problems, this chapter introduces Pic%, a prototype-based variant of Pico, a small Scheme derivative developed by Theo D'Hondt [DDD04]. Pic% is an extension of Pico in which we tried to incorporate as many properties of the Agora model as possible without ending up with a language hard to implement as Agora. Relaxing on purity was part of the solution.

5.3 Pico: the History and Rationale of Pic%

As explained, the roots of Pico lie in education. After years of teaching Pascal in a computer science introductory course for freshmen in exact sciences other than computer science[3] we had to face the fact that the overall results were deplorable. Many students kept on struggling with the syntax, static typing rules, the positioning of semicolons, the difference between procedures and functions, the necessity to compile and so on. We therefore started a project to redesign the entire course. One of the things we really wanted to get rid of was the laborious edit-compile-run cycle which we experienced to be an obstacle. Pure functional programming was ruled out for we want to accustom students with the notion of a changing memory, a notion that is still inherent to mainstream computer science practice. All these restrictions led us to languages like Scheme [KCE98], Smalltalk [GR89] and Agora. But we were left no other choice than to admit that their simple regular syntax and semantics even take our computer science students two years to master fully.

That is why we started to explore a simple language in which students were motivated to explore programming in a read-eval-print-loop as is the case with Scheme, but with a language that is:

- *easy to read*. The language had to look like calculus because that is about the only experience the intended audience has with formal languages.
- *as powerful and simple as Scheme*. We wanted a small number of powerful concepts instead of the baroque concept set of Pascal, Ada or Java.
- *extensible* in the same way Scheme, Smalltalk and Agora are. Instead of focussing on learning a fixed control structures suite by heart, we wanted that set to be easily extensible, much in the same way Smalltalk and Agora control structures are.

[3]Physics, Chemistry, Mathematics, Biology, Biotechnology, Geography and Geology.

- *lightweight* both in support and implementation. E.g. we did not want a language that requires complicated editing facilities like Smalltalk and Self. Nor did we want an interpretation model that requires the rather heavy message passing interplay the Agora semantics require.

At first sight, this seemed impossible. The extensibility of a language in the spirit of Smalltalk, Self, Agora or Scheme goes hand in hand with the simplicity and regularity of their syntax. However, the "easy to read" requirement seemed to be diametrically opposed to this. As we show, the exploration of an original suite of parameter passing techniques was the key to our quest. The outcome is called Pico.

5.4 The Acting Forces

In current day academic language design, syntax is no longer an exciting study field. But as explained in the previous sections, in our case this was one of the actual objectives of the project. The Pico experience taught us that it is not easy to come up with a syntax that is easy to read, very orthogonal, and, that allows the specification of a simple and regular extensible semantics.

When looking at existing languages, we can divide them into three flavors when it comes to their syntax:

An Irregular Keyword-based Syntax. The first kind of syntactic flavor is best known because so many popular languages have a syntactic system belonging to this flavor. The general idea is that the language is built around a number of keywords each with dedicated rules of how sub-expressions or sub-statements are to be centered around and grouped by those keywords. Examples of such languages are C, Pascal, Ada, Java, C++ and many more. These syntactic systems are irregular in the sense that every construct of the language has dedicated rules as to how instances of that construct are to be written down. Languages like these have the advantage of being easy to read. But they are harder to write programs in because one has to know their syntactic system thoroughly to construct programs correctly. As such, teaching the syntax in an encyclopedic, long-winded way is about the only option. Moreover, it is not easy to equip such a language with an extensible syntax and semantics.

A Regular Syntax without Special forms. In languages of the second flavor, the syntactic rules are extremely simple in the sense that everything is expressed using one single syntax rule. Examples of such languages are pure functional languages (where everything is a function application), Smalltalk and Self (where everything is a message send). In pure functional languages, it is possible to express everything as a function application because these languages usually support lazy evaluation allowing "keywords" to be functions whose arguments are not evaluated unless this is really necessary. However, in stateful languages this is not possible because laziness does not work together very well with imperative features. Therefore, languages like Smalltalk and Self use eager evaluation, meaning that arguments are always evaluated before a message is sent. Therefore, in Smalltalk, arguments of "language messages" like

`ifTrue:ifFalse:` must be manually wrapped in a lambda (called a block in Smalltalk) to delay them. This is also known as manual thunkification. The lambda, called a thunk, is passed around and calling it then causes its body expression to be evaluated. Important for our work is that the regularity of Smalltalk's syntax combined with its eager evaluation forces programmers to use its block system to clumsily prevent some expressions from being evaluated. Therefore, although languages in this second category are extremely easy to extend, programs written in them are often obscure, especially to novices.
A Regular Syntax with Special Forms. A third and final way of specifying syntax is a mixture of these keyword-based and regular syntax definition flavors. It is adopted by languages like Scheme, Agora, and Prolog. These languages are highly regular in the sense that all constructs are specified using exactly the same rules even though they semantically behave quite differently. In Scheme, almost every expression looks like a function call. Normal function calls follow eager evaluation, but for some "function names" (like `define`, `if`) a special -partially lazy- evaluation is defined. These "functions that bear a special status in the evaluator" are called *special forms* in Scheme. In Prolog we have the same situation where some predicate names behave differently from regular predicates. In Agora, reifiers (i.e., special forms) are recognized by their boldfaced appearance. Just as is the case for languages in the second category, languages in this third category are easy to extend as is illustrated by the Scheme macro system. Furthermore, constructing programs in languages of this third category is quite simple. However, students have to meticulously "parse" programs to read them. We found that even sophomores in computer science that have been extensively exposed to Scheme still find deciphering Scheme programs troublesome.

To the best of our knowledge an **intersection of these three flavors** that combines only their advantages is empty. What we were actually after was a language with a syntax as easy to read as keyword-based languages, as easy to write as ordinary calculus, yet as orthogonal in syntax and semantics as the languages with a regular syntax. Furthermore, instead of hard coding all control structures with ad hoc rules to be taught in an encyclopedic tedious way, we wanted them to be specifiable and extensible in the language itself as in Smalltalk. But simpler. It will seem that the key to solve our problem lies in a rather original parameter passing technique. Parameter passing used to be a hot topic in the seventies when differences between call-by-value and call-by-reference were exploited in programming languages. However, the distinction between these parameter passing schemes was not really a result of language design considerations, but was driven from an implementational point of view: should a thing or a pointer to the thing be passed around. As far as we know, Algo60 (with its call-by-name and call-by-value scheme) and Ada (with its `IN`, `OUT` and `IN OUT` parameter annotation system) were the only languages in which programmers were able to specify how parameters had to behave from a *conceptual* point of view.

In the following section, we will explain Pico and its innovative parameter passing technique and show how it is a simple, extensible lightweight Scheme variant that is also easy to read. Especially section 5.6 shows how the technique yields an extensible language, *without* having to introduce special forms. In section 5.8 we introduce Pic%, a Pico extension with object-oriented mechanisms that follow the extreme encapsulation principle explained in the previous chapter. We will do this by designing a Pico extension, called Pic%. However, in contrast to Agora, Pic% does *not* suffer from Agora's problems outlined in section 5.2.

5.5 Pico: The Original Language

Pico is best explained in two stages. First, in section 5.5.1 the raw foundations are presented. Second, some orthogonal additions turning Pico into a realistic language are discussed one by one, from section 5.5.2 onwards.

5.5.1 The Pico 3x4 Syntax System

Before moving on to an in-depth explanation of the language, the following Pico code snippet, meant as a teaser, gives a general flavor of the language. It is a straightforward implementation of the famous quick sort algorithm:

```
QuickSort(V, Low, High)::
  { Left: Low;
    Right: High;
    Pivot:: V[(Left + Right) // 2];
    until(Left > Right,
          { while(V[Left] < Pivot, Left:= Left+1);
            while(V[Right] > Pivot, Right:= Right-1);
            if(not(Left > Right),
               { Swap(V, Left, Right);
                 Left:= Left+1; Right:= Right-1 },
               false) });
    if(Low < Right, QuickSort(V, Low, Right), false);
    if(High > Left, QuickSort(V, Left, High), false) }
```

The first layer of the Pico syntax and semantics is explained by means of the three by four matrix depicted in table 5.1. This two dimensional matrix emerges from taking all possible combinations of **two design decision dimensions**. First, a Pico expression is always evaluated in the context of an environment (called a dictionary in Pico terminology) and one dimension of understanding Pico consists of viewing Pico in terms of manipulations of this dictionary. Therefore, each row in table 5.1 gives syntax to add something mutable or immutable to the dictionary (with `:`, resp. `::`), to refer to something in the dictionary, and to update something in the dictionary (with `:=`). Second, all Pico values (aside from literal values like integer numbers, fractions, texts and void) are described by what we call *invocations*. These invocations constitute the horizontal dimension of the Pico language design space. Invocations are used to refer to values in an atomic way, to refer to functions, and to refer

Table 5.1: Pico Basic Syntax

kind of invocation:	name invocations `nam`	table invocations $\texttt{nam[e}_1\texttt{]}$	function invocations $\texttt{nam(e}_1\texttt{, ... ,e}_n\texttt{)}$
reference	`nam`	`nam[e]`	$\texttt{nam(e}_1\texttt{, ... ,e}_n\texttt{)}$
definition	`nam: e`	$\texttt{nam[e}_1\texttt{]: e}_2$	$\texttt{nam(e}_1\texttt{, ... ,e}_n\texttt{): e}$
declaration	`nam:: e`	$\texttt{nam[e}_1\texttt{]:: e}_2$	$\texttt{nam(e}_1\texttt{, ... ,e}_n\texttt{):: e}$
assignment	`nam:= e`	$\texttt{nam[e}_1\texttt{]:= e}_2$	$\texttt{nam(e}_1\texttt{, ... ,e}_n\texttt{):= e}$

to tables. Tables are Pico terminology for what is usually known as arrays. The first kind of invocations, called name invocations, are ordinary references like `nam`. The second kind are of the form `nam[exp]` and the final ones look like $\texttt{nam(exp}_1\texttt{,...,exp}_n\texttt{)}$. Combined with the other dimension, this gives us syntax to declare variables with an initial value, to update them and to refer to them. Likewise, we can declare tables and functions, we can update a table position and we can change the body of the function. All these concepts are clarified in the explanation that follows[4].

Before doing so, let us emphasize the fact that in Pico, everything is a first class value: basic values, functions and tables can all be passed around as arguments, can be returned from functions, can be used as the right hand side of an assignment and so on. Another important thing to keep in mind is that Pico functions (in contrast to Scheme for example) always have a name. This decision was made because of the intended audience: indeed, mathematics does not feature something such as anonymous functions (in fact, functions are always named f, g and h in mathematics and many students already consider names like `fac` or `fib` as strange). Having said this, let us now run through the table:

- We start in the first column. A mutable variable is installed in the current dictionary using `nam: e` where `e` designates the initial value. `e` can be anything because expressions *always* yield a value. Hence, the expression `v: (n: (t: "hello"))` will install three mutable variables which are all initialized to `"hello"`. Referring to a variable simply happens by naming it. Finally, variable assignment[5] is almost the same as variable definition. The only difference is that the variable is expected to reside in the dictionary. Immutable variables (i.e., constants) are declared using the double colon syntax.

- Manipulating tables (i.e., arrays) is the topic of the second column. Table indexes run from 1 to the size of the table. Let us start with table definition which is the only expression type for which the semantics is not trivial from what one would normally expect. In an expression like $\texttt{t[e}_1\texttt{]: e}_2$, $\texttt{e}_1$ is

[4]In this dissertation, we give an informal explanation of the Pico concepts. A formal specification in the form of a meta-circular definition can be downloaded from `http://pico.vub.ac.be`.

[5]We use `:=` for assignment as we wanted to reserve `=` for equality tests.

evaluated to yield an integer acting as the size of the new table being installed in the dictionary under the name `t`. The entries of that table will be filled with the result of evaluating $\texttt{e}_2$. The unexpected behavior lies in the fact that $\texttt{e}_2$ is evaluated again and again *for every* entry of the table. This allows for expressions like `t[n]:(i:=i+1)` to create a table with `n` numbers in ascending order (provided that a variable `i` exists somewhere). Referring to a value in a table happens with an expression of the form `nam[e]` whose semantics is as expected. Finally, updating a table position happens with a $\texttt{nam[e}_1\texttt{]:=e}_2$ expression, the meaning of which is predictable as well. Just like ordinary variables can be declared constant with the double colon syntax, immutable tables can be created using the same declaration syntax. However, this does not mean that the entries of the table are immutable. It just means that the name associated with the table value is an immutable name.

- The final column in Table 5.1 is about function manipulation. Function scoping is lexical as in Scheme. But in contrast to Scheme, Pico functions always carry a name. A function with name `nam`, parameters $\texttt{p}_1, \ldots, \texttt{p}_n$ and body `e` is defined by the expression $\texttt{nam(p}_1\texttt{, ..., p}_n\texttt{):e}$. The expression can be anything. Hence, the expression `f(x,y):g(t,u):x+y+t+u` will define a function `f` with two parameters that will return another function of two parameters. An expression of the form $\texttt{nam(e}_1\texttt{, ... ,e}_n\texttt{)}$ will lookup `nam` and check it to be a function. After checking the number of arguments, the parameters will be bound from left to right to the arguments[6]. In the first layer of Pico we are describing here, eager evaluation is used such that all arguments will be evaluated. In the following sections we will see how to delay arguments from being evaluated. Last, function assignment happens in the same way as function definition. The name is looked up and its associated value (whether it is a function or not) will be garbage collected. The name will be (re)associated with a new function.

  Notice that Pico allows operators to be used in infix notation. This is syntactical sugar, though. An operator application like `x+y` is replaced by the parser by a regular function application `+(x,y)`. Operators are recognized by their name which consists of special symbols such as `+`, `*` and the like. But apart from that they are mere functions in all aspects.

This ends our explanation of the first level of understanding Pico. Notice that the curly braces used in the QuickSort teaser are not covered by this syntax. Furthermore, with what we know so far, it is not possible to define or even *use* a conditional `if` expression. Indeed, using an `if` with our syntax would force us to write it down as `if(condition, then, else)` causing all(!) arguments to be evaluated. In order to mend this, the following sections add some bells and whistles to this basic framework, which at first might seem rough edges to

[6]Notice, that in contrast to Scheme, we *do* specify the order (left to right) in which the arguments are evaluated and bound.

the proposal. However, as we will show, they are the key to turning Pico into a practical extensible language.

5.5.2 The Apply (@) operator

Much of the power of Scheme is due to the fact that it is list-based. Data structures basically consist of lists. Furthermore Scheme programs are also stored and represented as lists. This unification of the basic data structures and abstract grammar is at the basis of much of the power of Scheme. Features such as meta programming and a nice conceptual vision on functions of variable length arguments are direct consequences of the list orientation of Scheme. This is also supported by the implementation technology: the Scheme memory model is essentially list based, both in its abilities to store programs as in the way it stores data structures. In Pico this list-orientation was replaced by tables, i.e., arrays. This philosophy was then consistently applied to the rest of the language: tables are used to store programs, tables are the basic data structures of the language, variable size argument lists are implemented by table parameter passing, the memory model and garbage collector are optimized to handle variable sized tables and meta programming consist of manipulating tables (i.e., the parse trees).

A fact not known by many people is that the arguments of a lambda is actually also a list. This means that the following two definitions are equivalent in Scheme:

```
(lambda (x y) (+ x y))

(lambda r (+ (car r) (cadr r)))
```

I.e., Scheme formal parameters are actually lists. This enables functions of a variable number of parameters as shown by the second example. Transposing these ideas to a table driven language, we introduced a special notation for functions of a variable number of arguments:

```
f@arg: size(arg)
```

This function takes any number of arguments. Upon calling the function `f(1,2,3)`, the arguments are evaluated from left to right. The resulting values are collected into a table of appropriate size and the table is passed to the function as `arg`. In the example given, the result of calling `f(1,2,3)` will be three because `size(arg)` is three because there are three arguments passed to `f`.

This feature allows us to write our own `begin` function:

```
begin@args: args[size(args)]
```

This `begin` function can be called with any number of arguments. These will be bound, as always, from left to right. Of course, this is what one expects when writing an expression like `begin(`e_1`,`e_2`, ... ,`e_n`)`. Inside `begin`, the values of the expressions will reside in the `args` table. The body determines its size and returns its last entry (cf. the value of `begin` in Scheme). Another application of this technique is a function for inline table creation:

```
table@args: args
```

A call like `table(1, 2.0, 3, "hello world")` evaluates all arguments, and passes them on as a table; `table` simply returns that table.

Syntactic Sugar

Because `begin` and `tab` are used so often, the resulting number of parentheses starts to clutter up programs. Therefore the Pico parser allows some syntactic sugar:

- First, the parser allows expressions to be grouped between curly braces and separated by semicolons as in { $\texttt{e}_1$; $\texttt{e}_2$; ... ;$\texttt{e}_n$ }. It will replace this expression by `begin(`$\texttt{e}_1$, $\texttt{e}_2$, ... ,$\texttt{e}_n$`)`.
- Second, the inline table creation is supported by enumerating their elements between brackets and separated by commas. These expressions are syntactic sugar for the corresponding calls to `table`. E.g., `[[1,2], [3,4]]` is equivalent to `tab(tab(1,2),tab(3,4))`.

These two shorthands considerably improve the readability of Pico programs but do not add concepts to the language.

The Apply-operators

An issue related to having functions that accept any number of arguments is the ability to apply a function to a number of arguments that is not known in advance. In Scheme, this is accomplished through the `apply` function that consumes a function and a list of arguments. In Pico this role is played by the apply operator `@` that takes a function and a table. It applies the function to the arguments in the table. The following example shows how these features are combined:

```
accumulate@args:
      for(i:res:0,i:=i+1,i<size(args),
           res:=res+args[i+1])
sumto(n):accumulate@(nrs[(i:0)+n]:(i:=i+1))
```

The first function adds all the numbers passed on to the function `accumulate` no matter how many are passed over. The second function applies this, using the apply operator, to a newly created table (called `nrs`) filled with numbers from 1 till n.

5.5.3 Pico's Parameter Passing Semantics

Besides the apply operators, another addition to the Pico 3x4 syntax system is the way formal parameters are bound to the actual arguments upon function calling. During this process the lexical environment of the called function is progressively extended with new bindings yielding an extended environment in which the body of the function will be evaluated. This lexical scoping rule is exactly the same as in Scheme.

The gedankenexperiments we carried out when designing Pico led to a generalized notion of parameter passing. Referring back to the last column of Table 5.1 we notice that the formal parameters `e`$_1$`, ... , e`$_n$ of a function definition can, in fact, be arbitrary expressions! How a parameter passing semantics should be defined for each parameter type was an important part of our language design. Although there are many cases that do not make sense (e.g. it does not make sense to bind the parameter expression `3` to the argument expression `6`), we found some very interesting nontrivial cases which will be explained in this section. We start the presentation with the most basic case, namely parameter binding for parameters that are ordinary name invocations.

Binding for Name Invocations: call-by-value

When a formal parameter of a function is an ordinary name invocation (i.e., an identifier), the Pico semantics prescribes that the passed argument will be evaluated and the dictionary is extended with one new association mapping the identifier to the evaluated value:

```
{ f(a, b): ... ;  f(1, 2) }
```

This semantics of binding a value to an identifier, often referred to as *call-by-value*, is the most common in programming languages. The important point is that the actual arguments are evaluated before the binding is established.

Binding for Function Invocations: call-by-expression

Binding is more complex when parameter expressions are functions invocations, i.e., constructions of the form `nam(e`$_1$`, ... ,e`$_n$`)`. For a parameter like this, binding a given argument `exp` to it is achieved by creating a new function that corresponds to `nam(e1, ...  ,en): exp`. Hence, binding an expression to an function invocation parameter consists in constructing a new function with formal parameters `e`$_1$`, ... ,e`$_n$ and body `exp`. The closure of this function is the current evaluation environment, which is "the environment of definition of the function", thereby following the lexical scoping rules. Let us exemplify this using the function definition:

```
g(f(a,b),x,y): { if(f(x,y) > 0, x, y) }
```

`g` has three parameters, `f`, `x` and `y`, the first of which is an function invocation, the type we are discussing here. The other two parameters are ordinary name invocations discussed above. Hence for a function call `g(a+b, 1, 2)`, the bindings `f(a,b): a+b`, `x: 1` and `y: 2` are performed, as always from left to right, and `g` can use them as needed. The point to highlight here is that the body and scope of `f` is dynamically associated upon each invocation of `g`. We call this type of parameter passing *call-by-expression.* At first sight this technique seems very reminiscent to Algol60's call-by-name parameter passing technique. However in contrast to call-by-name, call-by-expression follows the standard lexical scoping rules: at all times variables are looked up in the lexical dictionary; i.e., the dictionary that was valid at the time the expression in which those variables occur was turned into a closure.

Although this might seem like an extremely obscure language feature at first, the parameter binding semantics explained here turns out to be very natural for people without prior programming experience (which was our intended audience). Consider for example the bisection method to find the zero of a numerical function. Because computer scientists are so much used to work with higher order functions, they will say(!) that a numeric procedure for this method takes "a function f", boundaries a and b and an accuracy *epsilon.* High school students, however, will rather say that the bisection method can be used to find the zero of "a function f of x" between a and b with precision *epsilon.* The subtle difference between "f" and "f of x" is often a source of problems when teaching higher order functions: *we* as computer scientists rather work with f because we are used to lexical scoping and local parameters. Students with only an education in high school mathematics will prefer the second interpretation. The binding semantics explained in this section allows this to be written down in a very natural way:

```
zero(a, b, f(x), epsilon):
  { c: (a+b)/2;
    if(abs(f(c)) < epsilon,
    c,
    if(f(a)*f(c) < 0,
      zero(a, c, f(x), epsilon),
      zero(c, b, f(x), epsilon))) }
```

A call of `zero` such as `zero(-3,3,x*x-6,0.01)` will bind the reference invocations `a`, `b` and `epsilon` to `-3`, `3` and `0.01` respectively. The application invocation `f(x)` will be bound to `x*x-6` which will internally create a function `f(x):x*x-6`.

As we will show in section 5.6, it is this feature that allows for Pico extensions to be written in Pico itself. Thanks to the original parameter passing mechanism, extending Pico does not require complicated macro facilities (as in Scheme), nor manually delaying some constituents (as in Smalltalk or Self).

5.5.4 First Class Dictionaries and Qualification

Pico is not an object-oriented programming language. Nevertheless, Pico features some basic object-based facilities such as object creation and late-bound message sending. More advanced object-oriented features like inheritance, classes or delegation are missing, though. These rudimentary object-based programming is enabled by rendering the Pico dictionary system first class. Getting a

Table 5.2: Pico Qualification Syntax

kind of	name invocations	table invocations	function invocations
...	see table 5.1	...	..
qualification	`exp.nam`	$\texttt{exp}_1\texttt{nam[e}_2\texttt{]}$	$\texttt{exp.nam(e}_1\texttt{, ... ,e}_n\texttt{)}$

reference to a dictionary is accomplished through a native function `capture()` which always returns the "current" dictionary of the evaluator. This dictionary is conceived and implemented as a linked list of name bindings. Every time a definition or a declaration is evaluated, a slot is added to the end of that list. Every time a name is looked for, the search starts at the end of the list and proceeds upward. This means that a newer definition shadows an older definition such that a mechanism like overriding is mimicked.

In order to use such environments, the syntax of table 5.1 is extended with an additional row resulting in the table shown in figure 5.2. Given an environment e, then *qualification expressions* of the form *e.inv* where *inv* is any invocation, can be used to denote reading a variable in an environment (using `e.v`), accessing a table that resides in an environment (using `e.t[i]` and calling a function that resides in an environment (using `e.f(arg)`). In every case, the name of the invocation is used to lookup the value in the environment.

The combination of `capture` and the way dictionaries are already structured according to the standard lexical scoping rules (i.e., for every "method" local variables have to be attached to the lexical scope residing in that method) allow for a view-method-like inheritance as explained in section 4.3.2. The following code excerpt illustrates this. The example shows a counter constructor function that returns a captured dictionary each time it is called. It declares three publically accessible constants `incr`, `decr` and `protect`. Once a counter `c` exists, we can send it a message `c.protect(5)`. As we can read from the code, this wil again declare two contants `incr` and `decr` that, as a consequence, shadow the existing ones. The result of sending `protect(5)` is again a dictionary that is actually an extension of the original environment. The resulting dictionary configuration is depicted in figure 5.2. Hence, `protect` can be seen as a mixin-method (more precisely, a view).

```
counter(n):
  { incr():: n:= n+1;
    decr():: n:=n-1;
    super: void;
    protect(limit)::
      { incr()::
          if(n=limit,error("overflow"),super.incr()) };
        decr()::
          if(n=-limit,error("underflow"),super.decr()) };
        capture() }
    super:= capture() }
```

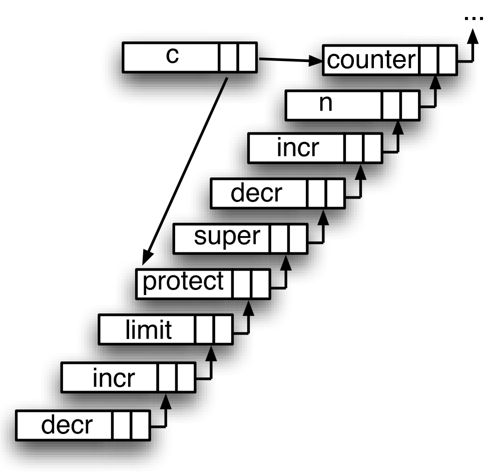


Figure 5.2: Pico Objects As First Class Dictionaries

A final note on scoping. As explained, Pico's dictionaries contain both mutable and immutable names. We chose to align immutable names with the accessible ones. Hence, the names accessed using the qualification must be names installed in the dictionary using `::`. Those names (whatever value they are bound to) can be read with a qualification. It is impossible to change them, however. Of course, variables that belong to one's "own" dictionary can be assigned, but this is to be done without a qualification. Simply stated, mutable variables (or functions or tables) are always private to the dictionary in which they reside. External code can only read immutable fields.

Notice that all the native functions of Pico are declared as constants which renders them publically available. This means that if `e` is a dictionary, than expressions like `e.sin(pi)` are meaningful. Moreover, dictionaries can "override" these primitives by simply redeclaring them. Indeed, if this `e` would have declared its own `sin` then the expression `e.sin(pi)` would have selected the latter one because lookup in dictionaries always proceeds from the most recent declared name, upward. This kind of qualification allows for modular programming.

5.5.5 Closing Pico's Syntax

Although the syntax defined by tables 5.1 and 5.2 constitutes a complete realistic programming language, there are some situations in which the syntax yields verbose and cumbersome programs, namely:

- The syntax has no provisions for multi-dimensional tables. Indeed, although there is nothing that prevents a table `t` from containing other

Table 5.3: Pico Syntax Closure

`inv`: kind of	names	table invocations	function invocations
simple invocation:	`nam`	`nam[`e_1`, ...]`	`nam(`e_1`, ...)`
definition	`nam: e`	`nam[`e_1`, ...]: e`	`nam(`e_1`, ...): e`
declaration	`nam:: e`	`nam[`e_1`, ...]:: e`	`nam(`e_1`, ...):: e`
assignment	`nam:= e`	`nam[`e_1`, ...]:= e`	`nam(`e_1`, ...):= e`
qualification	`e.inv`	`e.inv[`e_1`, ...]`	`e.inv(`e_1`, ...)`
'closed' invocation	`inv`	`inv[`e_1`, ...]`	`inv(`e_1`, ...)`

tables, there is no dedicated syntax to access such structures requiring verbose expressions like `t1:t[1];t1[2]` to access multi-dimensional tables. Therefore, the index expressions have been extended to include lists of expressions, with the expected semantics for expressions like `t[1,2]`. In the case of table definitions, the expressions for the higher dimensions are re-evaluated for every slot for the lower dimensions which enables one liners such as `triangle[3+(i:j:0),i:=i+1]:(j:=j+1)` to denote `[[1],[2,2],[3,3,3]]`.

- Another disturbing shortcoming of the basic syntax is that names are always required to denote functions and tables which excludes expressions like `t[1][3]`, `t[1](5)`, `f(3)[3]` and `f(3)(5)`.

This made us take the closure along `nam` of Pico's basic syntax defined in tables 5.1 and 5.2. The result is shown in table 5.3.

5.5.6 Meta-programming and Reflection in Pico

Just as in Scheme, the REPL functions `read`, `eval` and `print` have been reified. Applying `read` to any text value yields a parse tree which can be evaluated using `eval`. The result thereof is a printable Pico value. Hence, the following is a perfectly valid Pico meta program: `print(eval(read(accept())))`. It asks input from the user and prints the evaluated result on the screen.

As explained before, much of Scheme's power lies in the fact that both data and programs are represented as lists which means that Scheme's list processing capabilities can also be used to dissect programs. In Pico, this focus on lists has been consistently transposed to tables. This means that parse trees in Pico are data structures that can be manipulated using the primitive functions `get` and `set` which both consume a number as their first two arguments. `set` takes an additional value as the third parameter. Hence, `get` and `set` can be thought of as functions that can read and write the descendants of the parse tree. Of course, the number and type of these children depends on the type of the parse tree. This type can be determined by a primitive function `tag`. Conversely, `make`, which takes a tag (i.e., a number) and creates a parse tree of that type. As an example, evaluating `tag(read("x:3"))` results in 6, the tag for "definition expressions". Expressions like `get(read("x:3"),1)` return the first subexpression (i.e., `x`) and `get(read("x:3"),2)` the second one (i.e., `3`).

`eval` takes one argument and will evaluate it in the dictionary valid at the moment of the call. Since `eval` is a publically declared native, it is visible in every environment. So when called `e.eval(exp)`, this will evaluate `exp` in the environment that is valid during the call, which is precisely `e`. Of course, this is a potential breach of encapsulation because one can evaluate expressions like `e.eval(read("x"))` to read a private field `x` of `e`. But the point is that, an environment that wants to preclude this can always protect itself by overriding `eval` to do something safe.

5.5.7 Continuations

Just like Scheme, Pico features first class continuations. A design error of Scheme is to try to align these continuations with lambdas. Indeed, in Scheme, the primitive function `call-with-current-continuation` allows one to access the current continuation. Such a continuation can be activated through regular function application syntax. However, the semantics of activating a continuation is radically different from the semantics of function calling. This results in totally different behaviour of an expression of the form `(f x)` depending on the fact that `f` is bound to a regular function or to a continuation. In our opion, this is the source of much of the mystification and confusion around continuations. Scheme continuations *are not* functions and should not be used as if they were. Therefore, Pico uses the primitive functions `call(exp(continuation)):...` and `continue(c,v):...` to manipulate continuations. The idea is that `call` is parametrised by an expression that depends on `continuation`. The expression is immediately evaluated with `continuation` bound to the current continuation. Subsequently, this continuation can be passed around like any other first class value. The continuation can be reactivated using `continue` by giving it the continuation and the value that will be used as the result of the jump, i.e., the value that the continuation of `call` will receive. The following code, grabs a continuation and stores it in a variable `c`:

```
{ c:void;
  display(4+call({c:cont;5}))
}
```

The result of this code is that 9 is displayed on the screen and that, on the fly, the continuation of the `call` is being stored in `c`. Activating the continuation with e.g. `continue(c,6)` will display 10 on the screen. The following code illustrates how first class continuations in Pico can be used to devise a Pascal-like goto system:

```
{ make_label()::call(cont);
  goto(l)::continue(l,l);
  testgoto():{
    i:0;
```

```
        loop:void;
        loop:= make_label();
        i:=i+1;
        display(i);
        if(i<10,goto(loop),display(eoln)) }
    }
```

The `label` function takes the current execution status as a continuation. `goto` just continues this by "jumping" to the continuation. Note that we have to use an assignment and not a definition expression `loop:make_label()`, for otherwise we would be defining a new label `loop` each time we reactivate the continuation.

5.6 Extending Pico without Special Forms

As we discussed in section 5.3, easy extensibility was one of the main design goals of Pico. This section shows — by construction — that we have succeeded in this. Section 5.6.1 shows how the boolean system and a complete suite of Algol-like control flow primitives have been implemented in Pico itself. Subsequently, a Scheme like `cond` conditional will be presented in section 5.6.2. We will finish by tailoring Pico with a realistic exception handling system à la Java or C++.

5.6.1 Control Structures

An easy extension is a realistic implementation of the Church-booleans in Pico, together with the complete suite of control structures it induces. Pico's boolean system is actually an adaptation for imperative languages of the Church-booleans, in which avoiding evaluation is necessary because of side-efffects. The idea is to define booleans as functions that choose between two options passed on as delayed arguments because of the thunkification that results from the arguments being function invocations (without parameters). Based on these definitions, one of the core control functions of Pico, the `if` decision function, can be introduced in the same way it was proposed by λ-calculus:

```
true(t, f()):: t
false(t(), f):: f
if(cond, t(), f()):: cond(t(), f())
```

These declarations combine the ideas of the Church-booleans system with imperative programming. Indeed, the functional parameters `t()` and `f()` will automatically convert actual arguments into functions of zero arguments, usually called thunks. Indeed, the `()`-parameters delay evaluation of the argument expressions because calling the function will not evaluate the argument but will instead create a function of zero parameters whose body will be the argument. This allows us to perform evaluation in a controlled way, by means of function application. Operators like `and`, `or` and `not` are also implemented in Pico this way.

This boolean system allows for the definition of some "commonly known" Algol-like control structures such as the `while` construct which implements a leading decision loop by means of proper tail recursion. The automatic thunkification of the arguments is essential.

```
while(cond(), exp()):
  { loop(value, pred): pred(loop(exp(), cond()), value);
    loop(void, cond()) }
```

In all these examples, thunks (i.e., functions of zero arguments) are used to achieve what Smalltalk does with blocks. However, remember that in Smalltalk, 'thunkification' is requested from the *user* of the control structure yielding programs that are sometimes hard to read. We believe that we have obtained a much cleaner syntax and semantics with the same power. Furthermore it is not necessary to enrich the language with extra concepts such as macros and quasiquoting as was needed to render Scheme's special form suite extensible.

5.6.2 A Scheme like `cond`

.

A Scheme like conditional can be programmed by using a combination of automatic thunkification and functions that consume a variable number of arguments:

```
{ cnd() ==> act() : [cnd,act];
  else:true;
  cond@clauses:{
    siz:size(clauses);
    idx:0;
    end:false;
    until(end,
          end:=or((idx:=idx+1)>siz,clauses[idx,1]));
    if(idx>siz,
      void,
      clauses[idx,2]) }
}
```

The following small example shows how the conditional can be used:

```
fib(n):
    cond( (n=0) ==> 1,
          (n=1) ==> 1,
          (n>1) ==> (fib(n-1)+fib(n-2)))
```

The condition system defines three components: the function `cond` that consumes a variable number of condition/action pairs. These pairs are constructed using the `==>` operator which takes two arguments, thunkifies them automatically and collects the two thunks in a table of size 2. `cond` runs through the table and runs the action that belongs to the first condition whose evaluation (i.e., call the thunk) yields `true`.

5.6.3 An Exception Handling System

The following code excerpt is a more adult extension of Pico. It combines the special argument passing style, first class continuations and clever usage of scoping to implement a true Java or C++ like exception handling mechanism in Pico. However, whereas Java and C++ (thanks to their static typing) identify an exception based on its static type (the `catch`-clause names the type of the exception it can catch), this is impossible in Pico. That is why a call to `trycatch` requires a dynamic `filter(exception)` expression which it will evaluate every time an exception is raised. Whenever this expression yields `true` it will evaluate the expression that belongs to `catch(exception,value)` in the context of the exception (the first argument for `raise`) and an additional return value (the second argument for `raise`). For every call to `trycatch`, the corresponding continuation is stored in "its" `raise` which is turned global, and a local reference to the "previous" raise is kept aside for restoration upon successfully returning from the `try` expression.

```
{ raise(exc,val):error("UNCAUGHT EXCEPTION");
  trycatch(try(), filter(exception),catch(exception,value)):
    call({ keep:raise;
           raise(id,retval):={
             raise:=keep;
             if(filter(id),
               continue(cont,catch(id,retval)),
               raise(id,retval))};
           res:try();
           raise:=keep;
           res}) }
```

As an example of using this exception handler, consider the following code to calculate the roots of a quadratic equation where a negative discrimant exception is cleanly caught. The exception is identified by the text value `noRoots` and the second argument of `trycatch` implements the exception identification filter appropriately.

```
{ root(a, b, c)::
    { d:: b^2 - 4*a*c;
      if(d = 0,
        -b/2/a,
        if(d > 0,
          [ (-b + sqrt(d))/2/a, (-b - sqrt(d))/2/a ],
          raise("noRoots", d))) };
safe_root(a, b, c)::
  trycatch(root(a, b, c),
           exception = "noRoots",
           display("discriminant negative: ",value)) }
```

5.7 Pico: Evaluation and Epilog

As explained in section 1.2, measuring expressiveness and quality of programming languages is not an exact science. Nevertheless we have the feeling of having established a consistent language that is easy to learn and implement. This was confirmed by the results we obtained in the introductory computer science course we were talking about in section 5.1. While the course of the Pascal era mainly consisted of explaining Pascal based on simplistic examples like determining greatest common divisors, the size and simplicity of Pico actually allowed us to focus on programming. In the 30-hours course, we exposed the students (after explaining Pico itself) to four experiments designed to stimulate their appetite for further exploration of computer science in their respective fields: simulation of population growth (biology), triangularization of matrices (mathematics), simulation of forced oscillations (physics) and querying of a small database representing the periodic table of elements (chemistry). Again, all this was done with 18-year olds in 30 hours! But apart from these astounding results, Pico is also interesting from an academic point of view due to the parameter binding system explained in the previous section. The parameter binding mechanism for function application invocations allows for an easy extension of the language in the same way Smalltalk and Scheme can be extended, but much simpler.

Referring back to the criteria put forward in section 5.3, we can conclude that Pico is a language that is

- **lightweight**. This is shown by the fact that 90 percent (appart form function arguments and the apply construction) of the syntax can be explained by a regular grid as shown in figure 5.3. It renders the implementation very simple as shown by the fact that we have a *complete* meta circular definition that fits in 1000 lines of Pico code.

- **easy to read**. Although this is a subjective matter that is highly culturally determined, we think that most people will find the above examples easy to decipher, even without prior knowledge of Pico. This is in sharp contrast to languages like Scheme and Smalltalk which require a good knowledge of their semantics to understand quite mundane programs.

- as **powerful and extensible as Scheme**. Pico features everything Scheme does ranging from higher order functions to first class tables and continuations. Moreover by cleverly fitting the qualification syntax into the rest of the language, Pico also supports first class dictionaries which enables a form of modular programming missing from Scheme.

- **easily extensible**. This has been illustrated in the previous section with the boolean system, the Algol-like control structures, the Scheme-like conditional and the exception handling system. This is mostly due to the innovative argument passing technique explained in section 5.5.3. It allows one to write "new special forms" without having to resort to quasi quoting and macro programming, nor requiring the cumbersome manual thunkification demanded by Smalltalk and Self.

5.8 Pic%: A Lightweight Agora Scion

Although Pico was originally designed for educational purposes, over the years, it has also become a small language-lab for experimentation with language features. [VD00] e.g. shows how Pico is nearly trivially extended with simplistic language features that enable internet-agent programming. This section presents Pic%, a prototype-based extension of Pico that incorporates the ideas of the Agora model explained in chapter 4 but that avoids the pitfalls listed in section 5.2.

A prelude for a true object-oriented extension of Pico was already given in section 5.5.4 in which we have shown how a very primitive object-system can be added to Pico by rendering dictionaries first class. Combined with the syntactic notion of qualification, Pico might arguably be called an object-oriented language because it features encapsulation and message passing. However, this is quite uninteresting a language because it does not include well-accepted notions such as late-binding of self, cloning and super sends. The goal of Pic% was to take the extremely simple notions of Pico and to add constructions to obtain a lightweight Agora variant. But Pico is not a pure object-oriented programming language in which everything is an object. Apart from objects (i.e., first class dictionaries) and basic values (like numbers) it also has the notion of first class functions which is, as we will demonstrate in section 5.8.4, not a simple feature to reconcile with object-oriented programming because of scoping problems. From a scientific point of view, this has been a very interesting excercise. Whereas languages like Self and Smalltalk can be thought of as languages in which first class dictionaries are omnipresent (i.e., objects), and whereas Scheme and Pico are languages in which first class functions are the norm, a language that combines these features apparently does not appear to exist. Pic% does!

5.8.1 Problems with Adding Cloning to Pico

However, moving on from Pico to Pic% was not a trivial exercise that merely involved the addition of some missing language features. In order to see the problem, imagine adding a cloning operator to Pico. Conforming to the extreme encapsulation limitations explained in chapter 4, this has to be a construction with which objects clone *themselves*. A straightforward option might be to add a native function `clone()` that takes a clone of the "current" dictionary, much in the spirit of the way `capture()` returns a reference to the current dictionary as explained in section 5.5.4. This has a number of problems:

- First, since dictionaries are linked lists of associations, there are no easily definable boundaries to denote how far such a list needs to be cloned. A complete copy is totally unacceptable because it would exclude any form of code reuse. This problem has forced us to think about *frames* as groups of names that have been defined or declared in a single procedure activation. This way, our cloning mechanism could be defined to clone the current frame only, resorting to parent sharing between the current frame and its clone. However,
- Second, since Pico functions are closures in the Scheme sense, they contain a reference to their enclosing dictionary which also contains a self-reference, leading to circular structures. We must therefore make sure that the dictionaries contained in those functions are cloned as well. Not doing this would result in functions whose scope is not the object in which they reside. Calling them would thus operate on the original object instead of the clone! Hence, functions have to be "deep cloned" to yield the correct scoping leading to an object model in which not a single method is shared, unless we manually use the traits technique which is practically not usable in the absence of multiple inheritance or other advanced inheritance schemes like the comb-inheritance of NewtonScript (see section 3.3). We have called this the re-entrancy problem in section 3.3 and we have shown in section 4.6.4 that Agora does not solve it.

These problems illustrate very well that adding a seemingly trivial feature to a well-polished language can result in unforeseen interactions with unacceptable semantics. Adding an operator as mundane as cloning seems to ruin Pico's entire notion of simple dictionary-based objects!

5.8.2 The Pic% Object Model

The considerations of section 5.8.1 have made us completely rethink the way dictionaries and objects are composed, and how objects are created. The first problem forced us to divide Pic% dictionaries into chains of frames, exactly like in Scheme. The second problem has made us reconsider the notion of a function such that it no longer contains a reference to its lexical dictionary such that we do not need to deep-copy them yet retain reentrant code. But of course this means that functions (or maybe better, methods, from know on) will have to be provided with a scope reference every time we *use* them, be it as a result of message passing or as a first class return value because one simply refers to the function just as Pico prescribes. It is the latter that turns Pic% into an prototype-based programming language with first class methods. These considerations have led us to the following definition of the Pic% object model:

- Every Pic% object is a linked list of frames resulting from successive object creations and method invocations (such as in Scheme or Pico).

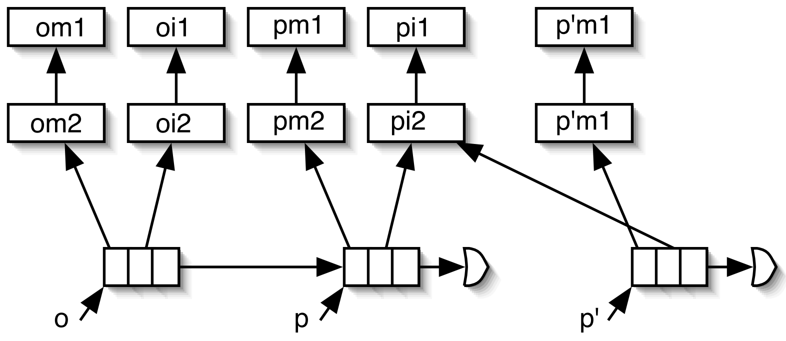


Figure 5.3: Pic% Object Structure

- Every frame has two parts: a list of declared bindings (immutable) and a list of defined bindings (mutable).
- Upon cloning a frame, a new frame is constructed with a reference to the same immutable bindings as the original and a clone of the mutable bindings.

In order to illustrate Pic%'s object structure, consider a single-frame object `p` with two mutable variables `pm1` and `pm2` and two immutable variables `pi1` and `pi2`. Suppose `o` is a descendant object (consisting of two frames) that also has two mutable variables `om1`, `om2` and two immutable ones `oi1` and `oi2`. Now let `p'` be a clone of `p` then the resulting object structure is the one depicted in figure 5.3.

5.8.3 Pic% Scoping and Closure Creation

As explained, the fact that Pic% methods no longer explicitly refer to their lexical scope, requires us to provide these methods with a scope as soon as they are referenced. Referencing a method can happen in the context of a message send of the form $o.m(a)$ or by simply referring to $o.m$ or by m in the context of the object itself. In some way, these references need to create a closure onto which a new frame binding formals to actuals can be attached, such that the code inside the method is correctly scoped. This opens up two interesting options:

- Either we attach the frame that binds formals to actuals to the object that received the message. This is the semantics outlined in [DD03b] and it seems to open up some interesting possibilities such as variable shadowing. We can regard it as *controlled dynamic scoping* because the parameter bindings are added to the dynamically determined dictionary (i.e., the receiver). In this case a newly created closure simply consists of the method together with the receiver.

- Another possibility is to attach the formals-actuals frame to the frame in which the method itself resides, i.e., the frame that results from the method lookup. This semantics is defended in [VM04] and corresponds to *classical lexical scoping*. However, since the native function `this()` should at all times yield a reference to the receiver of the message, this implies that the closure will consist of three parts: the method, the lexical scope (i.e., the frame of the method) and the dynamic receiver.

Although the implications of the second option are far less exciting than the ones induced by the first one (see our analysis in [DD03b] about Pic%) ChitChat uses the lexical one. In the context of distribution and mobility the static scoping mechanism seems to prevail, mainly because dynamic scoping does not seem to be easily alignable with the kind of state access control required by synchronisation (see section 8.4.1).

5.8.4 First Class Methods in Pic%

Our treatment of scoping above already sheds some light on the way functions are seen as closures emerging from selected methods. To the best of our knowledge there is no object-oriented programming language that supports this. Languages like Self and NewtonScript allow methods to be manipulated at runtime but only through the meta level and reflective capabilities which is quite cumbersome. The fact of having first class methods is actually less weird than the fact that (to the best or our knowledge) we are the first ones to come up with it. There are a indeed a number of good reasons for incorporating them in an object-oriented programming language:

- In Smalltalk and Self, the absence of first class methods is covered by blocks. A block is an expression between square brackets that can be sent a variant of the `value:` message. At first sight, a block fits well in the object-oriented paradigm: it is an object with a single method and calling the function corresponds to sending the corresponding message. However, blocks are not objects in the classical sense because - see section 2.4.2 - their pseudo variable `self` does not refer to the block itself, but to the object in which the block was created. Hence, a block is an object *whose `self` is does not refer to itself*!

- For exactly the same reasons, the designers of Agora94 [CDDS94] noticed that ex-nihilo object creation is not sufficient to mimic first class methods or lambdas. That's why Agora94 featured special syntax to create so called "single slot nested objects"' (SSNO), objects with a single method that inherits the self from the surrounding object.

- Maybe Java most painfully shows the need for first-class methods. Whereas Java is implemented using standard stack-based techniques (and not with heap-allocated closures like Scheme, Self or Smalltalk), it cannot handle closures. This would render first class methods really useful as painfully illustrated by Java's anonymous class restrictions. Consider the following Java code excerpt:

```
class OuterClass {
  private final int v=5;
  public Lambda m() {
    final int i=7;
    return new Lambda() {
      public void doit() {
        System.out.println(v+i) }
    }
}}
```

This example shows a method that returns an instance of an anonymous nested object implementing the interface `Lambda` *each time* it is called. Since Java does not have closures, all the variables of the outer scope must be *duplicated* into the new instance because the outer scope will no longer exist after `m` has finished and `doit` is invoked. Therefore, the variables used in the nested scope have to be declared `final` to preclude them from being assigned. After all, assigning them in one of the nested objects would make all the other duplicates obsolete. This could be solved much more expressively by allowing Java objects to return a first class version of a local method that transparently encapsulates the receiver in some way. This is precisely what Pic% does. We will come back to the implications thereof in chapter 6.

5.8.5 Adding Agora's Features to Pic%

Remember from section 5.8.1 that problems with cloning caused us to revise Pico's dictionary to represent objects. Having done this, let us return to cloning. In earlier Pic% experiments (such as presented in [DD03b]) cloning took the form of an external `clone` operator that had to be applied *to* an object. Our analysis of chapter 4 and [DDD03b] rules out this option because of the principle we called Extreme Encapsulation. This made us redesign this into a native `clone()` that takes a copy of the receiver in which that primitive is called, i.e., copy *ones own* state. In either case, the cloning model depicted in figure 5.3 is supported, yielding an object model in which immutable slots are automatically shared between clones. Recall from section 5.5.1 that these immutable slots coincide with the publically available slots. Hence, Pic%'s objects roughly consist of a local part and a public part. The public part can never be changed by means of assignments and is therefore shared by all the clones ever made. The local part is only visible and mutable by the object itself and is deep copied on every clone.

However, some experimentation with native functions like `clone()`[7] have

[7]We called them `view` and `mixin` to try to mimic the Agora constructions outlined in section 4.3.2

led us to the conclusion that it is very hard, if not impossible, to really have the full power of Agora by merely adding some natives. Indeed, a viewing method in Agora has to make an extension of its receiver *before* any expressions in the viewing method are executed for otherwise the new slots will be added to the receiver itself. The same holds for cloning methods and mixin methods. We really need a different type of methods besides the ordinary methods expressed in Pic% so far. Unfortunately it is not easy to come up with a syntax for such new kinds of methods without considerable changes to Pico's syntactic framework. In [VM04], however, a satisfying syntax extension proposal is made. It is based on the insight that, in Pico, the left hand side of a definition or a declaration can be a name, a function invocation or a table invocation, but not a qualification (e.g. it is not allowed to write `x.m` or `f.m(x)` on the left hand side of `::` or `:`). This made us add the following syntactic sugar[8]:

$$\begin{aligned} f.m(args) : body &\equiv m : f(\lambda args.body) \\ f.x : exp &\equiv x : f(exp) \\ f.t[i] : exp &\equiv t : f(?[i] : exp) \end{aligned}$$

These rules are easier to understand than one might expect. They allow an invocation to be "prefixed" by the name of a function f that will be applied to the entity being defined (or declared) *before* it is actually defined (or declared). This way f can "lift" an attribute before it is actually being installed. We have provided Pic% with three native lifting functions `view`, `cloning` and `mixin` that will transform their argument into one of Agora's "special" attributes as was extensively explained in section 4.4. The following example illustrates this. The view method `Point` will deliver a point with two methods: a regular method `print` and a cloning method `copy` whose body is executed in the context of a copy of the receiver upon invocation. This conforms to the Agora model explained in section 4.3.2. The view is used by simply calling it as in `Point(1,2)` or `this().Point(1,2)`.

```
view.Point(x,y)::{
  cloning.copy(xx,yy)::{
    x:=xx;
    y:=yy}
  print():: display("(x=",x,",y=",y,")") }
```

The final issue that needs to be resolved to yield a realistic prototype-based programming language is a notion of super sends. As explained in section 3.4.1, there is a huge difference between a super object (that has its own "self") and a super send (whose self should refer back to the one that performed it). The former could be achieved by adding some native `super()`. The latter, however,

[8]We list the transformations for definitions using `:`. Obviously they are also valid for declarations using `::`.

Table 5.4: Pic% Complete Syntax

`inv`: kind of	names	table invocations	function invocations
simple invocation:	`nam`	`nam[`e_1`, ...]`	`nam(`e_1`, ...)`
definition	`nam: e`	`nam[`e_1`, ...]: e`	`nam(`e_1`, ...): e`
declaration	`nam:: e`	`nam[`e_1`, ...]:: e`	`nam(`e_1`, ...):: e`
assignment	`nam:= e`	`nam[`e_1`, ...]:= e`	`nam(`e_1`, ...):= e`
super sends	`.nam`	`.nam[`e_1`, ...]`	`.nam(`e_1`, ...)`
qualification	`e.inv`	`e.inv[`e_1`, ...]`	`e.inv(`e_1`, ...)`
'closed' invocation	`inv`	`inv[`e_1`, ... ,`e_n`]`	`inv(`e_1`, ... ,`e_n`)`

requires dedicated syntax such as in Java. This is shown in the bottom row of figure 5.4 which now also summarizes the entire syntax of Pic% in which syntactic sugar such as the one presented above, the operators and the curly braces construction are omitted.

Note that some hygiene is required with the semantics of super sends in views, cloning methods and mixin methods. The rules are quite simple: when performing a super send in one of these, the resulting object will be selected to be the parent of the object (i.e., clone or extension) created. This way programmers can finetune precisely how far an object will be cloned.

5.9 Pic%: Evaluation and Epilog

In this section we take a step back and evaluate Pic% in the context of what has been said so far. Of course, the issues that will interest us most are the way Pic% respects the notions of extreme encapsulation and reflection protection explained in the previous chapter, and how well Pic% succeeds in mending Agora's problems outlined in section 5.2.

5.9.1 Agora Problems

Let us first start with reviewing some of Agora's problems described in section 5.2 that are very successfully solved by Pico and by Pic%:

- First, we have criticised Agora for the fact that it was all but a lightweight model. Indeed, evaluating Agora requires complicated machinery partly because of its extensive reflective capabilities. This is not the case for Pico, nor for Pic%. Proof of this is the fact that both models have been implemented numerous times in several languages varying from Scheme via Java to Smalltalk. Purely recursive versions of the evaluator exist as well as sophisticated CPS versions. But maybe the most overwhelming evidence that the model really is lightweight is the fact that a full implementation of Pic% for MacOSX, shipped with a programming environment, is an application of only 212K.

- Another problem with Agora is that although it is fully based on message passing but that, upon closer inspection, there are no less than 12 different message categories whose evaluation rules are fundamentally different and interact in extremely subtle ways. In Pico this is partially solved by the fact that is has much more surface syntax. The table shown in figure 5.4 explains the entire Pic% language in an extremely regular grid which is - in our opinion - simple to fully grasp. Again, our experience with undergraduates confirms this.

- Finally, and probably one of the most fundamental critiques we raised against the Agora model is that, despite of its sophisticated reflection machinery, it is not really an extensible language. In section 5.2 we have explained that the very model merely allows for the implementation of additional receiver*less* messages and not ordinary reifiers unless one changes the evaluator using complex pre-loading Steyaert calls "static reflection" [Ste94]. From a semantical point of view, the reflection operators in Pico and Pic% are much simpler than the ones of Agora. Nevertheless, section 5.6 illustrates that Pico does a pretty good job at writing extensions in itself. But from a certain point of view, Pic%'s reflection model is also much simpler than the one Agora has. This is the topic of section 5.9.3.

5.9.2 First Class Methods and Extreme Encapsulation

One of the important differences between the Agora model and Pic% are the first class methods. In the light of extreme encapsulation, a first reflex might be to treat this feature with a healthy dose of scepticism. After all, we are tampering with methods, one of the internal details of a an object. The answer of course is that with the object model designed for Pic%, the methods themselves do not contain any information about the object. They are totally stateless and it is exactly their invocation or explicit reference that wraps them in a closure. Moreover, every reference yields a new closure. Hence, first class methods cannnot breach the extreme encapsulation principle because an object is extremely encapsulated entity and, conceptually, there is no sharing.

In section 4.6.5 we have presented Agora as a unique model in the context of the Treaty of Orlando. We showed that it is a model of object-orientation that *does* align its object model with its template model (turning it into a prototype-based language) *without* aligning the object model to the self representation model. This characteristic is preserved in Pic%. Indeed, apart from message sending, Pic% features functional calling. However, just like in procedural languages, this is actually nothing but a message send to the current dictionary and since dictionaries and objects are aligned, this means that function calls are just self-sends. In this respect, we can refer back to table 4.8 which explains the essence of extreme encapsulation in Agora by the fact that different types of methods each implement their own *do* method which causes different effects to happen depending on the type of method at hand. In Pic% this is reflected by the fact that (first class) methods are applied to arguments and that each type of method "knows what to do" when it is applied. Applying an ordinary method causes the same effect as applying a Scheme lambda. Applying a cloning method will cause the method to create a clone of the encapsulated receiver before the body is run. These observations show that Pic% is completely consistent with Agora's properties discussed in chapter 4.

5.9.3 Pic%: Reflection Protection

The only Pico feature which seems unreconcilable with what we desire from a prototype-based programming model that avoids Agora's disadvantages is its reflection model. As we explained in section 5.5.6, Pico's reflection model basically consists of a set of accessor methods on the table-based memory model. This way, any parse tree can be manipulated by a Pico program. However, this also requires us to define `get` and `set` on objects and this is extremely undesired because it is a blunt violation of the reflection protection principle which is sacrosanct in the context of open networks and mobility as explained in section 3.6. This forced us to remove these primitives from Pic%. Does this turn Pic% into a non-reflective language? In the context of the research conducted by the Agora principle (see Steyaert's PhD [Ste94] for details) the answer to this question is a yes. However, because of the specific Pico features, the full-fledged reflection of Agora (which was at the same time pretty useless when it comes to extending Agora, see 5.2) does not seem to be needed that much:

- The special parameter passing technique explained in section 5.5.3 covers many uses of reflection in other systems. This was shown in section 5.6 where we have extended Pico with several constructions.

- One of the most useful applications of reflection in Agora was to extend objects from the outside. Indeed, in section 4.6.2 we have shown how the `QUOTE` and `UNQUOTE` reifiers were used to grab code and hand it over to an object which can subsequently evaluate it in its own context. This does not violate extreme encapsulation because it is the object itself that decides to unquote (thus evaluate) that expression in its own context. This technique is still available in Pic% as shown in the following example.

  ```
  view.object(x)::{
    write(xx)::(x:=xx);
    extend(g(anX))::view(g)(x)
    }
  ```

  This code shows how an object `o:object(3)` can be created that understands `write` and `extend`. `extend` accepts any expression depending on `anX`. It will execute that expression in the context of `x` after having transformed the expression into a true view method. This can be used to make an extension such as `o.extend(read()::anX)` that adds a `read` method to the object. Notice that this Pic% solution is even better than the solution of Agora explained in section 4.6.2 because an object does not have to expose the name of its internal variables. The `extend` method prescribes the names future extensions are allowed to see and use (here `x` called `anX` for external clients). Again, the point of this "extension from the outside" mechanism is that objects are extremely encapsulated but *can* provide extension if they want to.

- We did not explore the entire design space to re-introduce reflection for Pic% in different ways. For example, one might associate a mirror to every object one wants to reflect upon such as in Self [M. 03]. This does not necessarily have to breach encapsulation as long as every object is fully in charge of independently deciding whether or not to deliver these mirrors. This is totally feasible given our extreme encapsulation model.

5.9.4 The Re-Entrancy Problem Revisited

At the end of chapter 2 (more precisely in section 2.4.5) we have explained that prototype-based programming languages suffer from a reentrancy problem because methods conceptually reside in idiosyncratic objects. Either one chooses to copy the methods upon cloning the objects (thereby sacrificing code-reuse and reentrancy), or one has to resort to techniques such as comb-inheritance of traits (which induces the need for multiple inheritance).

In Pic% we have come up with a nice solution to this problem. By aligning public attributes with immutable attributes, these can be shared by all clones. This is exactly the kind of reentrancy class-based languages know, *without* adding additional language features! Indeed, upon cloning an object, only the dictionary holding the mutable slots is cloned. The one holding a reference to the constants is shared between all clones. Hence, this dictionary acts as a class-pointer in a class-based programming language. But the implementation can freely choose to what extent it really reuses this pointer! Because the attributes are totally immutable, they do not induce a conceptual sharing relation between clones. As explained in section 3.2, this is important in the context of open networks and mobility as it allows an implementation to copy the constant dictionary over the network without introducing hidden dependencies between objects the way classes do.

Let us illustrate all this by means of an example. The code excerpt below shows a constructor for points and circles. Given that code, a point is created with the expression `p:Point(0,0)` and a circle view can be defined on it using `c:p.Circle(1)`. The nice thing is that this is all very natural. All clones made of the point and all clones delivered by the circle will share the same `prn` method pointers. Just like in a class-based language, this code is completely reentrant because it will never be copied. Nevertheless we do not have to resort to traits or comb-inheritance for this.

```
view.Point(x,y)::{
    cloning.new(xx,yy)::{
      x:=xx;
      y:=yy};
    prn()::{ display(x,"@",y) };
    view.Circle(r)::{
      cloning.new(xx,yy,rr)::{
        .new(xx,yy);
        r:=rr };
      prn()::{ display("("); .prn();display(").",r)}}
  }
```

5.9.5 Some Concepts are Inherently Abstract

Another problem that we associated with prototype-based programming in section 3.3 and which the Agora model did not solve (see section 4.6.4) is that some notions are inherently abstract and do not fit in the prototype-based way of thinking at all. E.g., when one is writing the code for a stack, one is writing the code for all possible stacks, not just one particular stack. Hence, writing stack abstractions is an inherently class-based activity. The following code excerpt show how this is typically solved in Pic%. It shows a view method `Stack` that returns a new stack. Such view methods can be seen as the constructors in a class-based language. They encode a prescribed construction plan that is executed to make new prototypes. However, no language concept like classes is need. Moreover the constructor "functions" are mere methods belonging to a surrounding scope (read:object).

```
view.Stack(n):
  { T[n]: void;
    t: 0;
    empty():: t=0;
    full():: t=n;
    push(x)::
      { T[t:=t+1]:=x;
        this() };
    pop()::
      { x: T[t];
        t:=t-1; x };
    cloning.copy(nn)::{
      t:=0;
      T:=(T[nn]:void) };
    view.makeProtected()::
      { push(x)::
          if(full(),
            error("overflow"),
            .push(x) };
        pop()::
          if(empty(),
          error("underflow"),
          .pop()} }}
```

5.10 Conclusion

In this chapter we have shown that although the Agora model has some particularly nice language theoretical properties in the context of the problems outlined in chapter 3, it has several remaining problems related to its practical applicability. We have proposed Pic% as a lightweight projection of this model on top of a small experimental language called Pico. We have argued that Pic% shares the Agora properties so highly desired in the context of distribution and mobility without incorporating its problems. Pic% is a lightweight extensible object-oriented programming language. It is classless and adheres to the extreme encapsulation property.

Chapter 6

Pic% Idioms and Techniques

The proof of the pudding is in the eating. This chapter presents a few idioms that have been distilled over the years by programming in Agora and Pic%. The idioms presented have been selected because they are well-known but have a specific solution in Pic%, or, because they are an elegant application of Pic%'s specific features (to wit overriding of views and first class methods). We consider the idioms presented in this chapter as a partial validation of Pic%. They show that a number of high level object-oriented programming techniques are extremely elegantly formulated in the language and that Pic% is indeed an expressive medium.

6.1 Introduction

In the previous chapter we have presented the Pic% model as a scion of the Agora family that respects Agora's characteristics but which avoids its disadvantages. Surely defining a new programming language family is not enough and experience with its use has to be gathered. This is a difficult process in the context of academic research as it is hard to come up with a user group that can evaluate the expressivity of a language to a sufficient extent. This is certainly the case with the ChitChat model to be presented in chapter 8. However, Pic% is sufficiently developed in order to have gained some experience with it. It is mainly our own's but also that of some undergraduate and graduate students. Usually the techniques and idioms are manifestations of well-known idioms in object-orientation. However, there are also some extremely elegant solutions that are a consequence of the particular Pic% features.

The following sections are ordered arbitrarily and illustrate the techniques and idioms in a quite encyclopedic way because there is — as far as we know — no real deeper structural insight to be discovered behind them. The presented techniques where especially chosen because of their originality or because of their elegance. It remains to be said that, apart from the "Pic% pearls" presented here, a large part of the evaluation of Pico and Pic% is the existence of *complete* meta circular evaluators for both languages[1]. The fact that we stress this is — as explained in the introduction — a reflection of our view on programming languages which was highly inspired by the famous MIT Abelson&Sussman course [AS85] which puts a lot of emphasis on this theme.

6.2 Pic% at work: Design Patterns

Peter Norvig has shown that many design patterns, especially the creational ones, become superfluous in a language without static typing [Nor98]. The reason is that dynamically typed languages actually turn types into first class entities. This renders creational patterns such as factories and builders trivial: one simply passes along the type or the function that constructs objects of the type. This is not different in Pic% because (first class) view methods can be passed around like any other value. Moreover, declaring view methods privately allows one to assign them (using a "setter" indirection that is publically declared). This will cause users of the view to be given different kinds of objects depending on the implementation of the view.

In the same vein, many behavioural patterns are actually object-oriented designs that cover an absence of a particular combination of first class functions and higher order functions as featured by functional languages. But as we will show in the following sections, the fact that Pic%'s first class functions are actually first class methods with a (hidden) receiver wrapped in their closure enables some particularly elegant applications.

6.3 Overriding Views and Mixins: Factories

A technique that can be used to render object creation much more flexible than in other languages is the fact that view methods can be overridden just like any other method. This is illustrated in the following example:

```
view.GUIMaker()::{
  view.newWindow():: error("GUI is abstract");
  view.newButton():: error("GUI is abstract");
  view.MacGUIMaker()::{
    view.newWindow():: { thisIsA:: "Mac Window" };
    view.newButton():: { thisIsA:: "Mac Button" }};
  view.WinGUIMaker()::{
    view.newWindow():: { thisIsA:: "Win Window" };
    view.newButton():: { thisIsA:: "Win Button" }} }
```

The example shows an abstract factory with two concrete factories. Evaluating the expression `GUIMaker().MacGUIMaker()` constructs an object whose

[1]See `http://pico.vub.ac.be/`.

view methods will return Macintosh user interface components. The factory `GUIMaker().WinGUIMaker()` delivers Windows components on the other hand. Since object creation in Pic% de facto happens by method calling the above code is quite mundane Pic% code.

Notice that this factory technique combines very elegantly with the flexibility offered by the fact that Pic%'s object creation technique is mixin-based [BL92] (in [LS94] this was called modular inheritance). The idea is that an object can provide ordinary methods that deliver newly created objects based on the outcome of the algorithm they implemented. The method internally calls view methods to compose the desired object structure. This can also be used to preclude illogical objects from being created (e.g. a person cannot be a male and a female at the same time). The method might raise an error. The point of our argument is that all object creation is performed through message sending. Because of extreme encapsulation, the receiver of that message is entitled to implement the message in whatever way it wants to.

6.4 Cloning with Cloning Methods

Cloning methods were already included in Agora94 [CDDS94] but their implications where never really explored.

Cloning methods are actually extremely expressive in contrast to a simple `clone` operator that is to be applied *to* objects and in contrast with constructors as found in many class-based languages. The reason is that cloning methods interact nicely with delegation and overriding. Indeed, invocation of a cloning method will execute the body of that cloning method in a shallow copy of the receiver as explained in section 4.3.2. In the following code excerpt, we illustrate this.

```
view.Point(x,y)::{
  cloning.copy(xx,yy)::{
    x:=xx; y:=yy}}
```

The code shows a `Point` constructor that can be sent `copy`. This will result in creating a copy of the receiving point and will cause the method body (i.e., the assignments) to be evaluated in the context of the clone. This yields correctly initialized clones.

Cloning methods can, just like any other method, be overriden by more specialized cloning method or by ordinary methods, thereby preventing some objects from being cloned. The semantics of a cloning method is that it will *clone all the frames from the original receiving object, upward the delegation hierarchy including the frame in which the cloning method is found.* The result is an object whose parent is the same as the object from which the clone was made; i.e., the frame preceding the frame containing the cloning method. A super-send in a cloning method will yield an object that will be set as the super of the clone being constructed by the cloning method. This way, cloning methods can precisely specify how much of an object (as a list of frames) is being cloned and what the parent of the clone will be.

Cloning methods solve an old problem of prototype-based languages. In many class-based languages, there is often a notion of class-based encapsulation which means that objects of the same class are allowed to access each others private variables. The absence of classes requires prototype-based languages to mimic this using getter and setter methods, thereby also revealing the internal state to anyone else. Imaging a class `ComplexNumber` that specifies a method `add` to make the sum of the receiver and a single argument. In a class-based language the code can initialize the result's instance variables because it has access to both arguments because they are of the same class. In prototype-based languages this requires getter and setter methods. In Pic% cloning methods solve this as follows. Note that no accessors are needed.

```
view.Complex(re,im)::{
  ...
  cloning.new(rr,ii)::{
    re:=re+rr;im:=im+ii};
  add(c)::
    c.new(re,im)
  }
```

A rough edge to cloning methods is that they offer no provisions at all for cloning cyclic structures. Surely cloning methods can invoke (cloning) methods for the objects stored in their slots, these objects might be cloned twice if programmers do not take the necessary precautions such as passing along a clone map, a table that associates objects with their clone.

6.5 The Proxy Pattern

It was already been mentioned in the GoF book [GHJV95] that the proxy pattern is actually presented as a pattern simply because of the absence of dynamic delegation-based object extensions. The GoF proxy pattern wraps an existing object in an object the methods of which delegate their work to the original object. In a language with dynamic object extension such a proxy is simply a descendant object that extends the original object, overrides some of its methods but which automatically delegates all the other ones to the original object. This is not new, but we decided to mention the proxy pattern anyway because of two reasons:

- First, the name *delegation* is not well-chosen in the GoF book. As explained in section 2.3, the difference between real delegation and mere message forwarding as presented in the GoF book is late binding of self.
- Second, the proxy pattern is one of the fundamental techniques currently used in distributed software. The idea is that objects are represented

by a proxy on a different machine and that messages sent to the proxy are delegated, over the network, to the object the proxy represents. In ChitChat we extend Pic%'s views and parent sharing to networked views and networked parent sharing, thereby rendering the proxy pattern for distribution purposes superfluous too. It is the topic of chapter 8.

In brief, dynamic object extension using views renders the proxy pattern, both in the GoF sense as well as in the distribution sense superfluous.

6.6 True Singletons: Destructive Constructors

The singleton pattern was presented in the famous GoF book [GHJV95]. The idea is that some situations require "one of a kind objects" (hence the name singleton. Classically, this boils down to making sure that a class delivers just a single instance of itself, even if the application logic tries two make two such instances. Examples of this are `true` and `false`. In Smalltalk, the classes `True` and `False` each have their own idiosyncratic behaviour. In order to be safe, it has to be guaranteed that they deliver only a single instance of themselves, for otherwise the consistency of the system could not be guaranteed if e.g. "two trues" would exist. Other reasons for enforcing singleton objects might be efficiency (one does not want to have thousands of "a"-objects in a word-processor) but, more important for our research, security as argumented in section 3.6.3.

As already briefly touched upon in chapter 2, it is a merit of prototype-based languages in general that they are good at representing singletons. One "just makes a singleton object" and that is it. An extra merit of the Pic% model is that this singleton can really be be guaranteed to be unique by using the technique of *destructive constructors*. Destructive constructors are constructor functions that, upon invocation, overwrite themselves by their single instance. This is illustrated by the following code excerpt which shows an implementation of the classic Church booleans as view methods `True` and `False` which, when called, immediately overwrite themselves. As such, the view methods itself is lost upon first invokation and the names `True` and `False` get bound to the only objects that will ever be generated by that function.

```
{ view.False():{
    ifTrue(th(),el()):: el() ;
    ifFalse(th(),el()):: th() ;
    and(arg()):: this() ;
    or(arg())::  arg() ;
    not():: True;
    False:=this() };
  view.True():{
    ifTrue(th(),el()):: th() ;
    ifFalse(th(),el()):: el() ;
    and(arg()):: arg() ;
    or(arg()):: this() ;
    not():: False;
    True:=this() };
  True();
  False()
}
```

Notice that this is an idiom that is really unique in Pic%. Other prototype-based languages also allow one to declare singleton objects but because of their lack of extreme encapsulation and their lack of reflection protection, they cannot guarantee that users erroneously or maliciously copy or destructively extend that singleton. Pic% really allows for the construction of a secure singleton that cannot be tampered with. As we saw in section 3.6.3 this guarantee is important in the context of open networks. Consider the i-ticket with which Harry gets his butter in the scenario given in the introduction. A malicious user could create several copies of this i-ticket and could use this feature to pay once and collect butter for free endlessly with copies of the i-ticket. This example demonstrates the importance of controlling the number of objects of a certain type. But these problems are not unique to prototype-based languages. Reflection in Java e.g. allows one to simply bypass the object-construction facilities and create unwanted instances. Code like `iTicket.class.newInstance()` shows that it is easy to bypass the singleton property of objects. In Pic% however, this is impossible as reflection protection is guaranteed.

6.7 The Iterator Pattern

An extremely popular design pattern is the GoF-book is the iterator (or observer) pattern. The idea is to have an encapsulated data structure that has to be traversed in order for a certain algorithm to be applied to all the elements it contains. In a language with first class functions, this pattern is implemented as a `map` or `foreach` function. In a language without lambda closures, this is particularly cumbersome to implement as witnessed by the pattern. The pattern makes the data structure create an object (called `Iterator`) whose sole purpose it is to access the elements of the object one after the other. This made the Java designers overload the meaning of the `for` loop in order to have language support for this problem. The new "iterator for" was included in version 1.5 of the language. As with many things in Java, the new construct is actually an ad hoc feature to cover a particular situation where first class lambda's are missing.

In Pic% this is typically accomplished by "sending" ones algorithm to the data structure by an ordinary message. Pic%'s parameter passing semantics takes care of automatic thunkification. This is shown in the following code excerpt in which `iterate(algorithm(element))::...` accepts an algorithm (depending on `element`) which it internally applies to all its elements.

```
view.List()::{
  elements: ...;
  insert(el):: ...;
  iterate(algorithm(element))::
       ... apply algorithm to the elements ...
  ...}
```

The `iterate` method is used as one would expect. The following example shows how to print all the elements in a list.

```
l::List()

l.iterate(display(element))
```

The example shows how the innovative parameter passing technique of Pic% combines the best of two worlds: we do not need the clumsy iterator construction proposed by the GoF-pattern nor do we have to thunkify algorithms manually with block closures, which are, as we explained in section 5.8.4 "weird citizens in the object-oriented paradigm" because they are objects whose "self" is not itself. Furthermore, because of the first class methods of Pic% it is even possible to select the `iterate` method from the data structure and pass it along as an iterator in the GoF sense. But since this iterator is a higher order function it opens the way for the complete arsenal of higher order function composition techniques, know from functional programming, to be applied.

6.8 The MVC Pattern

A commonly used technique in Pic% is a variant of the famous model-view-controller pattern. Because of extreme encapsulation, an object can only be extended or cloned by sending it a message that is required to be implemented by a view method or a cloning method. The key of the technique discussed here is to let an object store a reference to all newly created objects it spawns.

The following code excerpts illustrates this. The `Model` method is used to create a model which contains one method `tick` that changes the internal state of the model and subsequently calls `this().changed()`. This will traverse the list `us` to send `notify()` to all its elements. These elements are added to the list every time the view method `View` is called to create a view on the model. This method registers the view with the model so that it gets notified everytime the model changes. Needless to say, the fact that no extension views can be created on the model without sending a message to the model is crucial here.

```
view.Model()::{
  var:0;
  us:void;
  changed():{
    fst:us;
    while(not(is_void(fst)),
      { fst[1].notify();
        fst:=fst[2] })};

  tick()::{                                ‘ example method
    var:=var+1;
    this().changed()};

  view.View(viewCode(var))::{
    notify()::viewCode(var);          ‘ the view’s algorithm
    us:=[this(),us]}
```

We will use this technique in our distribution examples when a server spawns clients by special distributed view methods. The server can keep references to its clients this way. We refer to section 8.6 for more details.

6.9 First Class Methods and Connectors

One of the drawbacks of object-oriented programming when it comes to higher order composition technology — as required by component-oriented programming — is that all communication between objects has to happen by message sending, and that these messages have a name that is usually hardwired in the program. This was e.g. the reason why introspection operators where added to Java pretty early to support the Bean component model and make it less fragile due to explicit dependencies resulting from hardcoded message names.

Consider the following programming problem: the goal is to write a generic higher order constructor `BroadCaster` that accepts any number of components and that implements a single method `broadCast`. The method should accept any number of arguments and should call a certain method in all the components it groups. Since we want `BroadCaster` to be a generic higher order construct that is to be used with different kinds of components that might look totally different, we do not want the name of the messages to be hardwired.

In Java and Smalltalk, such constructs are only possible because of their meta programming facilities. And even though they are possible, they easily result in pretty cumbersome code:

- Since the name of messages is hardwired in Java, we have to select the methods out of the components using `java.lang.reflect` and hand over the components and the methods in two different arrays to the `BroadCaster` constructor. The implementation of the `broadcast` method then runs pairwise through both arrays and invokes every method on the corresponding receiver.

- In Smalltalk the situation is a little bit cleaner but still requires the double array solution. One hands over the components and the names of the messages that will have to be sent.

Pic%'s first class methods offer a particularly elegant solution to this component wiring problem. It is shown below:

```
view.BroadCast@components:{
  broadCast@args::
     results[(i:0)+size(components)]:components[i:=i+1]@args }
```

The `BroadCast` object constructor takes any number of components — represented as first class methods — which it can internally access as a table. Upon sending `broadCast` with any number of arguments (internally called `args`), a new table of results is created that collects the results of applying each `component[i]` to those `args` using the apply operator `@`.

Here is how it can be used. Two totally independent components `c1` and `c2` are created. Their methods are selected and handed over to the broadcaster constructor. The result is an object that can be sent `broadCast` with the three arguments that will be distributed over all components. Of course, the broadcaster also works correctly for methods with any other number of arguments.

```
view.ComponentType1()::{
  do1(x,y,z)::display("One says",x,y,z,eoln)}

view.ComponentType2()::{
  do2(x,y,z)::display("Two says",x,y,z,eoln)}

{  c1:ComponentType1();
   c2:ComponentType2();
   bc:BroadCast(c1.do1 , c2.do2);
   bc.broadCast(1,2,3) }
```

Notice how the first class methods `c1.do1` and `c2.do2` are handed over to the connector. Because of the semantics of first class methods, these contain the receivers `c1` and `c2` in their closure. This is just a small initial example of how first class methods could be used to wire up components. Investigating how far the feature can help us in this is an interesting topic of future research. Another elegant application of first class methods is their use as listeners, the topic of the next section.

6.10 Listeners are First Class Methods

A popular use of blocks in Smalltalk is to associate widgets of a user interface with the action that has to be undertaken upon activation of those widgets. The typical example is to associate the action that has to be undertaken upon clicking a button. A block or lambda is an easy solution because it can be created in the same context that will be needed when invoking it. A lambda that is attached to a widget this way is called a *listener*. Since Java does not feature lambdas one typically has to create a special listener objects and hand it over all the information that it will need upon activation by the widget. To alleviate this problem, starting from Java version 1.3, it was decided to add anonymous objects to Java. These are objects that can — like lambdas — be created ex nihilo in the very context that will be needed when invoking them. This was already briefly touched upon in section 5.8.4. However, as explained there, the JVM's architecture puts severe restrictions on the usability of outer scope variables from within the inner scope determined by the listener. This often renders the listener technique cumbersome to use. Again, the anonymous objects with their restricted scoping rules (they cannot modify variables residing in their lexical scope) is an ad hoc technique to cover the absence of lambdas.

In Pic% this is elegantly resolved by selecting a first class method (which might even be private) from its context and hand it over to the listener. Because of the semantics of first-class methods as explained in section 5.8.4, selecting a method also encapsulates the receiver itself which automatically makes the first class method — when invoked — refer to the right object. Hence, first-class methods as designed in Pic% could be a very elegant solution to cover for Java's lack of lambdas, without having to resort to a full-fledged closure system: the method simply encapsulates the "this". The following Pic% example illustrates the technique. Upon creation of the dialog box in `makeDialog`, the buttons are created and are handed over the methods `uponCancel` and `uponOK` that will be invoked when one of the buttons is pressed. Whenever this happens, they will naturally run in the context of the model from which they where selected.

```
view.Button(listener)::{
  push()::listener() }

view.DialogBox(message,b1,b2)::{
  show():: display(message);
  ok()::b1.push();
  cancel()::b2.push()}

view.AModel()::{
   uponCancel():{ display("Cancel was pushed") };
   uponOK():{ display("OK was pushed") };

   makeDialog()::{
     ok:Button(uponOK);
     ca:Button(uponCancel);
     DialogBox("OK or Cancel?",ok,ca)} }
```

Notice that this technique also elegantly interacts with inheritance. By replacing `uponOK` in the creation of the button by `this().uponOK`, the first class method will be searched for in the descendant of `AModel` in which the dialog box is created.

6.11 Conclusions

This small “interludium chapter” has presented some of the techniques and idioms that have emerged from using Pic% over the years. Some of these idioms were “mere” elegant implementations of existing patterns in Pic%. However, we have also illustrated that two innovative language features, to wit the functional parameter passing and the first class methods yield particularly powerful higher order programming techniques without having to resort to lambda closures. We have shown that the first class methods could even be turned into a satisfactory (and relatively easy to implement) feature for languages like Java, the designers of which obviously want to avoid the full-fledged closure system that is needed to implement lambdas.

In chapter 1 we have argued that the expressiveness of a programming language is hard, if not impossible, to measure. One could argue that many of the design patterns presented in GoF are actually “workarounds” developed by practitioners to cover for the absence of language features (such as first class lambdas and dynamic types) they need to express their designs. In this sense, the more elegant and concise a language allows these patterns to be expressed, the more expressive it is. From this point of view, this chapter “shows“ that Pic% can be considered quite expressive.

Chapter 7

OOP Concurrency and Distribution for Open Networks

After having presented our prototype-based programming languages in the previous chapters, we now turn our attention to distribution and mobility, the main topic of this dissertation. Chapter 8 integrates a concurrency and distribution model in Pic%. Therefore this chapter reviews the basic notions of concurrency and distribution in object-oriented systems and gives a taxonomy of the language design options that we have at our disposal in this field. We will review existing concurrent and distributed languages (notably ABCL, Argus, Emerald and Obliq) and evaluate them in our context of open networks set in chapter 1.

7.1 Introduction

At this point in the dissertation it is useful to take a step back and recall the problem statements we formulated in section 1.3 of the introduction. There we stated that it is not our goal to come up with a full-fledged production language for open networks but to formulate clearly and offer an initial solution to four fundamental problems such languages will have to address. Let us review them with the knowledge about prototype-based languages built up so far:

1. **The Ambient Object Paradigm Problem.** In chapter 3, we argued that programming such networks in class-based languages is virtually a lost battle. On the other hand, a reformulation of the encapsulation problems of prototype-based languages in the context of security rendered them non-viable as well. We resolve this apparent stalemate by applying our Pic% model in the context of open networks. Pic% neither has classes, nor suffers from the drawbacks of classic prototype-based languages.

2. **The Distributed Sharing Problem.** Although distribution is all about sharing resources, current languages offer practically no features for sharing. The object is usually considered as *the* unit of distribution and networked object referencing is the only sharing technique offered. We take the position that unfortunately objects are the *only* unit of distribution and that structural network-based sharing mechanisms such as those offered by prototypes are needed.

3. **The Concurrent Parent Sharing Problem.** This argument also holds for concurrency. Whereas sharing is a fundamental part of distribution, concurrency models obsessively avoid sharing because of the tremendous problems the combination of sharing and concurrency poses.

4. **Move Considered Harmful.** Current mobile programming languages are at the software engineering level that sequential programming languages were in the early sixties. In chapter 9, we will argue that current mobile features correspond to the "goto" and inevitably lead to distributed object-soups that no human reader is able to comprehend by merely looking at the source code.

Chapter 8 iterates over the Pic% proposal of chapter 5 and enriches it with language constructs for distribution and concurrency. The resulting distributed programming language, called ChitChat, is unique in that it does not shun sharing mutable state. On the contrary, state sharing between concurrently running distributed entities is extolled by the model. Roughly spoken, it exploits the prototype-based notion of parent sharing to control state sharing between concurrently running (distributed) entities. We extensively discuss the ChitChat model in chapter 8.

In order to put our proposal into scientific context and in order to justify our design choices, this chapter reviews the vast body of literature about concurrency and distribution, especially in the context of open networks and from a programming language point of view.

Theoretically, concurrency and distribution are two separate fields because a distributed system might be entirely sequential in nature. In practice however, this is not the case as even client server architectures are written such that servers can handle multiple requests from clients concurrently. Hence, in practice distribution often "implies" concurrency. This is even more so in our context where devices are independent computers and the network connecting them is not predetermined. The resulting configuration is almost naturally concurrent. The converse does not hold because of a different notion of failure that distributed and "mere" concurrent systems have. Whereas a "mere" concurrent program succeeds or fails as a whole, a distributed program can fail partially because one node might crash. This "partial failure" is extremely important for distribution and is even more important in the context of open networks as machines can leave the network (making other machines failing partially) and re-appear a few moments later in order to resume contact. This aspect is not treated by our research at all. We will get back to this in our future work section of chapter 10.

The chapter is structured along five sections. Section 7.2 explains some general issues about concurrent object-oriented programming. Section 7.3 looks at the options at our disposal in object-oriented concurrent language design. Especially the ABCL model [YBS86] will receive a lot of attention. Subsequently, sections 7.4 and 7.5 do the same exercise for distribution. Finally, section 7.6 evaluates the most important languages in the context of open networks.

7.2 Concurrency in Object-Orientation

We start by giving a birds eye view on the major schools that exist in combining concurrency with object-orientation. But to evaluate the available options, it is important to understand the kind of concurrency we are after.

7.2.1 Reasons for Concurrency

In our opinion there are at least two reasons for using concurrency in an object-oriented language:

- A first form of concurrency arises in the context of parallel systems where networks of processors that are connected to speed up computations. This view of concurrency gave rise to concurrency models such as CSP [Hoa78] and actors [Agh86]. This "speed-driven stance", often associated with number crunching, is not our focus of research.

- The second manifestation of concurrency is more recent. E.g. in Java many user interface components actually produce threads that operate on a shared state because the application itself is sequential. This "unintentional source of concurrency" is driven by software engineering. It is a consequence of the way we structure applications. In the same vein, our AmI context requires concurrently running devices to cooperate smoothly. This requires a lot of resource sharing and the software that runs on them will have to deal with this. The problems resulting from this concurrency, albeit different in goal, are thus also a consequence of the way the applications are structured.

We deliberately stress this distinction because an important reason for shunning state sharing in speed-driven concurrency is to improve their mutual independence to gain speed. For our purposes however, concurrency is a consequence of the way we *model* the software as (distributed) processes that share state. This relaxes the efficiency requirements for the targeted language features somewhat. Instead, the concurrency model will be steered by the way sharing and distributing is conceived.

7.2.2 Issues in Concurrency

Concurrent programming is hard. The difference between sequential programs and concurrent ones is arguably of the same order of magnitude as the difference between functional programs and imperative ones. Two problems recur:

Race Conditions

The property of functional programs of being easy to reason about completely vanishes when mutable state is added because the temporal status of that state partially determines the semantics of the program. Having two programs that concurrently operate on that state makes the problem twice as large because the program's semantics can depend on whichever program performed its changes first. These *race conditions* form *the* problem of concurrent programming and avoiding them usually renders things extremely complicated.

The most frequently occurring problem is that of a *ghost write*. Suppose two travel agencies access the same shared airplane that has one free seat left. Both agencies concurrently check whether a seat is available. Upon getting a positive answer, they subsequently both book the seat. One of the two write operations will have no effect and is thus called a ghost write.

The Inheritance Anomaly

Although it is completely outside the scope of our research, any text that is about combining objects and concurrency is at least expected to mention the inheritance anomaly. Obviously, not all methods of a concurrent object can be invoked at all times. E.g., a consumer of an object cannot consume if the object was not previously filled by a producer. Therefore, special synchronization code in the consumption method has to delay that consumer. The inheritance anomaly boils down to the fact that this code is generally not reusable. Overriding one single method often requires *all* methods to be overridden. We refer to [MY93] for an overview of the problem and ways to solve it.

7.2.3 Schools of OO Concurrency

In their excellent overview article [BGL98], the authors identify three object-oriented concurrency schools: the applicative (or middleware) approach, the reflective approach and the integrative (or language) approach.

Applicative Approach

An example of the applicative approach is Smalltalk. Although Smalltalk is not a concurrent programming language, it contains predefined classes that implement, in Smalltalk, concepts such as processes, and monitors. Smalltalk composition technology is then used to "wire up" a concurrency program. Block closures are heavily used to "disguise" sequential code as processes or threads.

This approach to concurrency is cumbersome. As [BGL98] put it, *"the programmer faces at least two different major issues: programming with objects and managing concurrency and distribution of the program, also with objects but not the same objects!"* Unfortunately, this is how most concurrent programs are written nowadays. We highly doubt that this approach scales up to distributed and mobile software that has to deal with the complexity of open networks.

Reflective Approach

The second school in combining object-orientation and concurrency is known as the reflective approach. The idea here is to "open up" a sequential programming language with reflection operators in order to adapt and enhance it with concurrency concepts. At the meta level, reified language concepts are rendered concurrent and absorbed back into the language processor. An example is the CodA system [McA95] which uses the Smalltalk meta-level to introduce concurrency and distribution. At the meta-level, the applicative approach is used to compose the concurrent behaviour attributed to base level programs.

Integrative Approach

The integrative school of combining objects with concurrency is the one adhered to by our research. The integrative approach is about aligning object-oriented notions (like objects, classes, inheritance and methods) with concepts from concurrent programming, resulting in *concurrent object-oriented programming languages*. Proposals vary from straightforward language unions that add "threads" to a sequential language, to carefully designed language intersections (like the Actors [Agh86] model) in which object-oriented concepts like objects and messages were carefully integrated with the notions of concurrency. In [BGL98] the languages are classified along three design dimensions:

- The first dimension is the level to which objects and threads are aligned.
- The second is how good messages align with synchronisation boundaries.
- Third, there is the alignment of objects and the units of distribution.

In the following section, we explore the options in this design space.

7.3 Design Issues in Concurrent OOP

The three axes that make up the design space of the integrative approach are covered by sections 7.3.1, 7.3.2 and 7.3.3 respectively. Section 7.3.4 looks at the restrictions that open networks put on concurrency. Section 7.3.5 focuses on ABCL, a language in the space that seems to meet quite a lot of those restrictions.

7.3.1 Axis #1: Threads vs. Active Objects

The first language design dimension is about the extent to which objects and threads are aligned. The are two extremes to be reviewed.

Some languages, like Java, do not align these notions at all, resulting in language unions containing both **threads and objects**. Every thread is an independent "processor" that sends messages and runs methods. As such, different threads can operate on the same object resulting in race conditions. In order to prevent this from happening a flag-driven synchronization system is added to objects which indicates to what extent the object is in use by one or more threads. Java's `synchronized` keyword and method modifier states that only one thread can operate on the object declared synchronized. Unfortunately, this is not very high a level of abstraction. It appears to be very simple at first sight, but is extremely error-prone when used on a large scale because it makes programmers completely responsible for managing state sharing problems by handcrafting primitive monitor mechanisms on objects [Lea99].

At the other extreme of the spectrum are languages that fully align objects with threads such that objects *are* threads that consume messages. These are called **active objects** (as opposed to passive ones). In the "pure actor model" [Agh86] objects (called actors) are constantly running entities that have a queue to gather messages. Whenever ready, an actor consumes the next message and runs its associated method. After processing some instructions, the actor will typically replace itself using a `become` instruction. It can henceforth handle a new message, although the method is still running because an actor has no mutable state.

There seems to be a continuum between the thread model and the active object model, usually adding mutable state to the latter. For example, in ABCL, messages arrive in a queue but the object basically processes them one by one thereby avoiding internal race conditions. In others, active objects handle messages concurrently which requires internal synchronisation code for mutable state [BGL98].

7.3.2 Axis #2: Synchronization and Message Passing

Apart from the pure actor model, concurrently running entities manipulate shared state and will thus require synchronization to avoid race conditions. Synchronization requires synchronization points and it is natural to consider method boundaries for this because sending a message is where "control is handed over from one logical entity to another". The level to which synchronization and message sending are aligned is the second dimension along which concurrent object-oriented programming languages are classified.

Combining concurrency with message sending engenders three forms of synchronisation. If more than one thread can enter an object, we have to deal with *intra-object synchronisation* to protect internal state from being corrupted. One speaks of *inter-object synchronisation* if a message spawns concurrency (because the receiver handles it autonomously) and if the sender and receiver need to

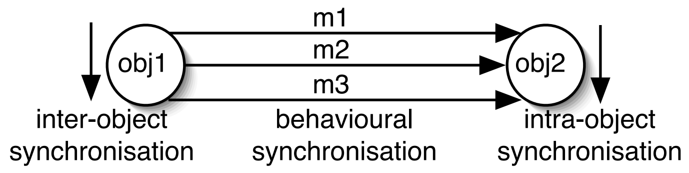

m1
m2
m3
obj1
obj2
inter-object synchronisation
behavioural synchronisation
intra-object synchronisation

Figure 7.1: Synchronization Schemes Schematically

synchronize afterwards. Finally, the messages an object understands define a communication protocol, also from a concurrency point of view. E.g. a read in a buffer is only possible after processing a write. Determining which methods can be invoked when is called *behavioural synchronization* or *conditional synchronization*. This is schematically depicted in figure 7.1.

As explained above, stateful languages require **intra-object synchronization** between multiple threads or methods to avoid race conditions. A primitive example is Java's `synchronized` keyword combined with its `wait` and `notify` natives. Intra-object synchronization appears to be extremely difficult and error-prone [Lea99]. The only viable solutions seem to be abandoning mutable state (like pure actors do) or abandoning intra-object concurrency (which is what ABCL does - see section 7.3.5).

Depending on the internal state of an object, some operations can temporarily be disabled. There are many mechanisms to achieve such **behavioural synchronization**. They fall into *centralised behavioural synchronization schemes* and *decentralised behavioural synchronization schemes*. In centralized schemes, there is a centralized expression in the program text (an AOP-like declaration) that specifies the rules for enabling and disabling methods. Decentralized behavioural synchronization means that the heading (or even body) of every method has the provisions necessary to make sure it is disabled whenever necessary. In [BGL98], a good overview is given about behavioural synchronization mechanisms. Examples are

- Flags combined with synchronization functions like `wait` and `signal`.
- Behaviour sets are sets of methods an object enables at a certain moment. Methods can change the contents of the sets.
- Path expressions describing which interleaving of messages are valid.
- Guards which are boolean expressions that block methods and trigger them anew when they become true.
- Chords in C# which require multiple messages to arrive at an object before a "shared" method body is executed.
- Guardians in Argus [Lis88] which put delayed methods into "guardian actors" and reactivate them back later on.

- Scheme-like “current continuations” which are grabbed and stored for later (when the conditions are met) re-activation.

Finally, figure 7.1 shows how **inter-object synchronization** is necessary when messages spawn concurrency which is the case in active object models. Whereas most thread-based systems (in which concurrency is spawned by thread creation) use *synchronous message sending* (i.e., the thread “flows” from the sender into the receiver back to the sender), the approaches based on active objects use *asynchronous message sending* which entails that sending a message is a source of concurrency. Since the sender continues, sender and receiver will need synchronization when the reply is ready. In some proposals, there is never a reply at all. Others work with a callback system. A particularly nice solution to the problem was developed in ABCL in which asynchronous message passing immediately returns a *promise*. This can be the final result, or, a placeholder that will change itself to the result upon completion of the method. In that case, the promise is said to be *fulfilled* by the receiving active object. The sender of the message will receive the promise and “think” that it is the answer to the message. It can store the promise, pass it as argument to other message etc. This maximizes concurrency. Only when the sender actually wants to do something with the value for which it is a promise, then it will actually block (and thus synchronize with the receiver). We will return to this in section 7.3.5 that treats ABCL in detail.

7.3.3 Axis #3: Objects as Unit of Distribution

The third dimension along which object-oriented concurrent languages can vary is determined by the extent to which objects are the units of distribution. Because distribution takes such an important part in our research we have devoted entire sections on this. Section 7.4 treats distribution issues in general and section 7.5 reviews distributed programming languages. But let us first have a look at how existing work on concurrency in object-orientation fits in our research domain. Section 7.3.4 looks at the specific requirements for concurrency in open networks and section 7.3.5 takes a closer look at one language (to wit ABCL) from the above design space that has some promising features in this context.

7.3.4 Requirements of Concurrency in Open Networks

As said in section 7.1, the goal is to enhance the Pic% model with a concurrency model that is applicable to distribute its objects across open networks. In addition to the more general wish list drafted in [CR93] about the features of a contemporary concurrency model, we now evaluate the options offered by the design space explored in the previous sections for open networks.

1. The communicating devices are not divided into predetermined clients and servers because they are independent computers. This rules out synchronous communication because it inherently follows a client-server strat-

egy in which a client explicitly waits for the result. Hence, we will focus on asynchronous message passing and adopt an actor-based model.

2. Historically, pure actors were a reaction to the tremendous difficulties that come with threads that share state. From a language theoretical point of view, it is undoubtedly the best marriage of objects and concurrency. However, we have two objections against pure actors. First, our very modeling domain is about smoothly *cooperating* distributed applications. This seems to imply that there is a notion of (at least temporal) sharing between the concurrently operating devices. Examples such as virtual white boards inherently contain the idea of a state that is shared between concurrently operating actors. In the same vain, a PAN "knows" the surname and mood of its user. This is very hard to program in pure actor systems. The only way cooperating actors can conceive of a "shared" state is to send messages back and forth all the time, using explicitly encoded session information[1]. This would be so much easier if the cooperating parties could "simply" share some state.

3. Another huge drawback of the actor model is that it puts an unreasonably high cognitive load on developers because its pure asynchronous message passing renders the sender and receiver of a message constantly out of sync. Circumventing this problem is done by passing along, with every message, a continuation actor that knows how to handle the rest of the computation [Lie87]. This continuation passing style works fine for small examples but does not seem to scale up. The problem is even worse in open networks. When two communicating actors temporarily move out of earshot they would need "service discovery" each time communication is required. Although this will always be needed at the system software level, we surely cannot expect application programmers to deal with "keeping the connection alive" for every message they want to send *and* for every result they want to get back. There needs to be some "rubberband" tying senders and receivers of messages together. In section 7.3.1 we discuss ABCL's promise system as a good candidate to solve this problem.

4. More than two devices might be involved in a communication. If one device requests a result from a second one which forwards the request (in a tail position) to a third one, then the second process does not have to be blocked unnecessarily, merely to pass on the result from the third object back to the first. Instead, the promise of a promise should immediately be returned to the first process. This "chaining of promises" will maximise the availability of devices and the minimise coupling between them which is important in case the second device (i.e., the middleman) goes out of range.

5. In developing software for open networks it is impossible for a programmer to foresee all possible interactions a networked actor might undergo

[1]Such as http-cookies.

with newly encountered devices. It will therefore be extremely hard to guarantee internal consistency of mutable state if that networked object allows intra-object concurrency. Even in applications where the two communicating parties were "written for each other", intra-object concurrency seems hard to manage. We will thus avoid it.

6. The model should be easily alignable with our distribution requirements which explicitly extoll the idea that cooperating distributed parties share information with each other. In chapter 8 we will model such shared information as shared parents of distributed objects.

7. Finally, the model has to work in a mobile setting where objects hop from one device to another. Again, the model of explicitly late bound message passing (enabling rerouting) combined with an implicit "connection" established between the receiver and the promise that awaits the result, will make things much easier.

As we will see in section 7.3.5, apart from the sharing we are after, these restrictions combined with the insights of previous analyses such as [CR93] heavily point towards ABCL which is a realistic variant of the actor model with mutable state. Enhancing Pic% with an ABCL like concurrency model for distribution, based on shared parents, is the topic of chapter 8.

7.3.5 The Actor Based Concurrent Language (ABCL)

ABCL [YBS86] tries to reconcile actor languages with the notion of mutable state. We discuss ABCL in detail because it is a prototype-based language that was the intellectual basis of the ChitChat model explained in chapter 8. ABCL supports cloning and class-like object constructor functions, but, to the best of our knowledge, no delegation that involves late binding of self.

Objects in ABCL

In order to explain how ABCL works, consider the following code excerpt which is a code skeleton for a bounded buffer. ABCL is a language whose concurrency emerges solely by means of sending messages to active objects (without intra-object concurrency). These are always in one of the states: *dormant*, *active* or *waiting*. A dormant object can be activated by sending it a message and becomes dormant again as soon as it has no more messages in its queue. An active object can put itself into the waiting state using the `(select ...)` statement. The meaning is to become active again only after having processed the message referred to by that `select`. This is one of ABCL's behavioural synchronization mechanisms. Another one is to attach constraints to the message pattern, a feature not illustrated by the code [YBS86]. In the code, each `=>` entry shows a message pattern and its associated method.

```
[ object Buffer
   (state ... )
   (script
     (=> [:put obj]
         (if full?
            then (select (=> [:get] ... )))
          ... store obj ...)
     (=> [:get]
         (if empty?
            then (select (=> [:put obj] ...))
          else
          ... remove obj ... )))]
```

Messages Passing Semantics

Sending a message to an object puts the message in the object's queue. There are two types of messages. Ordinary messages are added to the end of queue. Express messages have higher priority and can even temporarily interrupt the execution of another message. This means that message handling loses its atomicity which can introduce race conditions. Crucial messages can be precluded from being interrupted however.

ABCL features three kinds of messages: *past*, *now* and *future*. "Past" messages correspond to pure actor-like messages that never block their sender. "Now" type messages correspond to synchronous message passing causing their sender to wait until the receiver explicitly returns control. Finally, "future" type messages allow both sender and receiver to run concurrently but will cause the sender to wait as soon as it really needs the result but which was not returned by the receiver yet. To ensure this, upon reception of a "future" message, the receiver immediately returns a so called *promise* to the sender. In ABCL, promises are not transparent but represented by a so called "future object" whose value can be claimed by the sender. If the receiver is not finished yet, the sender will block until the receiver fulfils the promise by filling the future object with some value.

In contrast to Multilisp's futures [Hal85] or Argus's promises [LS88], ABCL allows for a future to result in more than one return value. Therefore, a future object is actually a queue accumulating all values returned by the receiver. This allows a sender and a receiver to exploit maximal concurrency and to return values back gradually. Synchronization (i.e., blocking the sender) occurs only when necessary. The concept is sometimes referred to as *wait-by-necessity.*

As said above, an important advantage of promises over the continuation passing style (CPS) advocated by the pure actors model is that they allow programmers to express their algorithms much more naturally. It does not force a program to be written in CPS because a promise is an (asynchronous) connection between the sender and receiver of a message. CPS is experienced by many people to "turn their algorithms inside out" because code that logically belongs together has to be put in separate continuations.

Summary

ABCL is a prototype-based concurrent language featuring active objects, cloning and constructor functions. It has no inheritance, nor delegation. Concurrency is maximized by (a)synchronous message passing with different flavours (now, future, past) and promises. Intra-object serialization is achieved using a queue collecting all messages and behavioural synchronization by constraints and the `select` delay instruction. Futures regulate the inter-object concurrency.

7.4 Distributed Objects

In section 7.2.1 we have explained to be investigating concurrency not in the speedup doctrine, but in the light of programming open distribution. Now we look at distribution itself. This section covers some generalities. Section 7.5 discusses existing languages in our context. We will not try to give a detailed discussion of what exactly is a distributed program because such discussions easily degrade into an attempt to determine the gender of the angels. We call a program distributed if it is executed by geographically dispersed machines (also called *hosts* or *computational environments*) such that the link connecting them is somehow detectable[2] to the running program.

As with concurrency (see section 7.2.3), there are three approaches to combine object-orientation and distribution: the applicative, the integrative and the reflective approach. Writing a distributed object-oriented application is not an easy task. The reason is that a combination of the applicative and reflective approach, called *middleware*, seems to be the omnipresent, but arguably also the toughest way to construct distributed applications. It requires the non trivial interactions between the distribution and "ordinary" object-oriented concepts to be dealt with manually which usually results in insurmountable problems as shown in section 7.4.1. That is why this dissertation promotes the integrative approach, especially for the dynamics induced by open networks.

7.4.1 Problems with the Middleware Approach

Neither practitioners nor academics spontaneously come up with distributed *languages* when thinking of distribution. Indeed, the bulk of distributed programming is nowadays done in mainstream sequential languages like C++ or Java using an intermediate middleware that combines libraries and AOP or other reflection techniques.

An important concept in object-oriented treatments of distribution is a *proxy*. A proxy is a local object that represents another remote object. Proxies are responsible for taking the actions necessary to forward local messages they receive to the remote object they represent. Issues such as (not) copying arguments of messages across the network and deciding what to do with the result

[2]In speed, in time, in memory consumption to take care of a message send, etc.

(copy it to the sender, move it to the sender, or creating a local proxy for the remote result) have to be taken into account. Middleware solutions typically have tools for generating proxy code that contains the machinery to forward messages to the objects they represent. Middleware often consists of a precompiler that transforms a program annotated with compiler directives indicating which parts and how the program has to be made "network-aware". The precompiler typically generates stubs for proxies and the underlying distribution machinery. This all sounds very nice and therefore many practioners claim that there is no need for dedicated language support for distribution. They promote the use of a pointerless programming language like Java combined with middleware technology such as e.g. Voyager [voy].

However, reality is not so simple. On top of the problems with concurrency which practically immediately pop up when distributing an application, middleware solutions, no matter how solid and complete, have many problems.

Referring to Remote Objects

One of the issues with distribution is the way objects find each other on the network, i.e., how local code gets references to remote objects. This problem is sometimes neglected because it seems to be "but a bootstrap problem": once a "first" reference to a remote object is obtained, it can be sent messages that return other references. But still, a protocol has to be designed to get a first reference. Surely, the poor "solution" consisting of a hardcoded network topology combined with "simply" referring to remote objects by a hardcoded (IP) addresses plus a hardcoded object descriptor, is not really scaleable and justifiable a solution. This is especially true in the context of open networks.

When striving for a solution that is more scaleable and manageable there are currently a number of alternatives.

a) Centralized Name Servers. The most popular solution, especially in middleware proposals, is that of a name server known to all the locations of the distributed application topology. Network references to objects that are registered with the name server, are acquired by querying the name server locally. Conceptually the name server is a central authority but robustness and performance can require name servers to be replicated on different machines. Name servers have the advantage of being simple text-based solutions. But this is also their biggest drawback as it requires objects to bear a unique name across the network. This is related to the fact that name servers are conceived as central authorities in charge of getting locations in touch with each other. It makes name servers practically unusable in the context of open networks.

b) Broadcasting Protocols. Broadcasting protocols are inherently decentralised. They assume that devices constantly "beam around" information that can be picked up by other devices. Examples are JXTA [Sun03] and M2MI [KB02]. JXTA is a set of protocols developed by Sun Microsystems in the context of Java, which allow devices in peer-to-peer topologies to communicate with each other. One of the protocols is the "Peer Discovery Protocol" which allows nodes to discover each other based on XML descriptions of broadcasted services. Once discovered, other protocols can be used for opening connections. An approach that is particularly interesting for dynamically defined processor clouds is the M2MI proposal (Many-to-many-invocation) [KB02], also implemented in Java. M2MI allows objects to send messages to each other by means of multicasts; messages that will be received by all objects that implement a certain Java interface. This mechanism - albeit pretty low level - was designed for exactly the kind of communication we are after.

The number of steps involved in "just getting a reference to an object on a different machine" in middleware solutions strengthens our opinion that this **should be offered by the programming language**. Obliq [Car95] for example uses built-in name server primitives in order to name objects across the network.

Representing Remote Objects: Proxies

Middleware solutions usually implement a remote object reference by means of a *proxy*, a placeholder that represents the remote object and that correctly forwards local messages to that object[3]. One of the tasks of middleware solutions usually is to generate proxy code. They offer tools that generate a proxy class on the basis of the signatures of the input class. The need for distribution transparency requires local and remote objects to be interchangeable which, in statically typed languages like Java, requires their types to be compatible. In Java the interface solution is often adopted: the original class is renamed and its original name is turned into an interface. Both the original class and the generated proxy class then implement the same interface. However, the proxy solution is often a source of problems. Generally spoken, they all boil down to the fact that the idea of a proxy breaks object identity: from a semantic point of view, proxies are perfect stand-ins for the objects they represent and the application logic will consider them the same. However from a technical point of view (i.e., the actual pointers) they are different objects and are thus not the same. This resulting problems are not easy to solve (see [PSH04]):

a) Language Operators vs. Proxies. Although at first sight it may seem an unimportant technical problem, the presence of language operators that are *not* subject to late binding polymorphism (such as `==` and `!=` in Java) can yield subtle bugs and insurmountable problems because, semantically an object and its proxy are equal but technically they are represented as two different objects. This even gets worse if "proxy management" does not guarantee an object to be represented by the same proxy every time it crosses network boundaries: an object arriving at a location along two different calling paths will be represented by two different proxies and will, as a consequence, not be equal. The basic problem is that many operators offered by languages like Java are not subject to late binding polymorphism. This is related to the problem of abundant language operators discussed in section 3.5. It is an extra argument to impose a strict adherence to message passing. As a matter of fact, some middleware

[3]This process is called *remote method invocation* and is treated below.

solutions will replace the usage of `==` by `equals` messages but this changes the semantics as illustrated by the following code excerpt:

```
Object canBeNull = random(true,false) ? null : object1;
if(canBeNull == object2) ... else ...
```

Bluntly replacing `==` by an invocation of `equals` will break the code every time the random generator returns `true`. `equals` is then sent to `null` which is not an object.

b) Static Typing vs. Proxies. A problem for which no simple solution exists is that the middleware proxy solution heavily interferes with static typing as featured by the bulk of languages used by middleware solutions. Imagine a library application that can transparently manipulate both local as well as remote books. It can display information about and reserve books irrespective of whether they are local or remote books. The solution is to have a `Book` class and a `BookProxy` class that represents remote books. The substitutability requirement for local and remote books constrains the way the classes are organised in a statically typed language.

- The `BookProxy` is a subclass of `Book`. This has the disadvantage that every proxy also contains the useless data fields of a `Book`.
- The class `Book` is refactored to an empty superclass of which both the original class as well as the proxy class are subclasses. This works fine as long as `Book` is an independent class that is not part of a hierarchy. If `Book` has a subclass, say `DigiBook`, then the substitutability requirements between books, digital books, proxies for books and proxies for digital books impose a multiple inheritance hierarchy with a diamond.
- The interface solution (for Java!) explained above avoids multiple inheritance but has other drawbacks. Although [ACFG01] shows that it can be improved significantly, the worst estimates for an interface invocation is that it is 50 times as expensive as a regular invocation. More severe difficulties of the interface solution is that it interferes heavily with features such as `instanceof`, `new` and certainly reflection!

c) Reflection vs. Proxies. The interface solution is only possible if one has access to the sources, for otherwise instructions like `new Book()` cannot be replaced unless dynamic class loaders are deployed. But even if we do have complete access to the sources, the interface solution easily breaks in combination with reflection. Indeed, no precompiler will be able to replace `Class.getClass("B"+"ook").newInstance()` correctly. This code will break unless the middleware also changes class loaders which is only possible in Java. Reflection operators can also be used to circumvent message passing. For example, compare `book.clone()` with `book.class.newInstance()`. When sent to a proxy, the first message will (if correctly forwarded) return a copy of the book. The second one yields a copy of the proxy. Such examples show that proxies do not interact smoothly with reflection, even reflection as weak as Java's.

d) Late Binding of Self vs. Proxies. One of the key notions of object-orientation is the notion of recursion through `self`. However, upon forwarding a message from a proxy to the object it represents, `self` will refer to the object itself and not to the proxy. This too causes technical problems. Imagine distributing an application that processes "model" objects referring to "view" objects in the MVC-sense. Unless *all logic* is kept strictly centralized on a server, we might want to store local views locally to avoid unnecessary network trafic. However, the MVC mechanism will never cause the local views to be notified. The problem is that, once the proxy delegates a message to the server, the proxy "loses control". When the model on the server sends `self changed` messages, they will not "come back" to the proxy rendering the local views stored in the proxy inconsistent.

e) Conclusion. The fundamental problem that is the root of all these technical incarnations is that a proxy semantically "represents" an object and should thus be fully interchangeable with that object. Unfortunately, technically, they are different objects and this seriously interacts with the rest of the language features. It should be clear that **programming language support for a transparent representation of "objects", whether local or remote is indispensable** to avoid these technical intricacies.

RMI, Argument Passing and Result Delivery

After having discussed the problems middleware solutions suffer from for getting a "first reference" to an object and associated to representing objects by proxies, let us now look at message passing in the context of distribution. A message sent to a remote object is referred to as a *Remote Method Invocation* (or RMI for short) which is the equivalent of Remote Procedure Call (or RPC) for procedural languages. The RMI mechanism is responsible for forwarding a message across the network upon reception by a proxy. The middleware takes care of correct message delivery, of correctly handing over arguments and results, and of propagation of thrown exceptions. Shougaard [Sch03] identifies six steps in executing a remote method invocation:

1. Get a proxy for the remote object.
2. Invoke the method on the proxy.
3. Take the necessary provisions to handle the arguments of the message. We will discuss this below.
4. Upon receiving the call and the arguments, the message has to be bound to the right method on the object being represented by the proxy, and provisions for sending back the result have to be made.
5. Run the actual method.
6. Return the (reference to the) result to the original caller.

Depending on the middleware, several possibilities arise for transferring arguments and results back and forth between the sender and the receiver. One might decide to copy the arguments over the network, to really move them back and forth or to simply give the receiver a proxy for the arguments that remain with the sender. The difference between copying and moving is subtle though important. An argument copied is not affected in case the receiver inflicts changes to the copy (possibly by sending it messages). For arguments truly moved back and forth, such changes are also perceived by the sender. In any case, moving or copying objects is not easy. They are actually graphs, which means that it has to be decided how "deep" every argument has to be moved or copied. Furthermore, technology is needed to *serialize* or *marshall* objects into a sequential representation that can be sent over the network after which the receiver can *deserialize* or *unmarshall* them, which is a costly operation. Java RMI spends 25 to 50 percent of its time in marshalling and unmarshalling arguments [PH99]. Other middleware solutions such as SOAP and XML seem to worsen this problem *a lot* [VMD$^+$04]. An alternative for moving or copying objects is to give the receiver proxies for the arguments which actually remain with the sender. This will reduce the performance of the system if the receiver interacts with them a lot, causing a lot of network traffic. These considerations cause most middleware to offer the choice, rendering them complex to use.

In an RMI, one will typically want exceptions thrown at the remote location to be propagated back to the sender. In some statically typed languages, like Java, this raises new typing problems. A method signature is required to declare the exceptions it possibly throws. If objects and their proxies are to be mutually substitutable and if the proxies can also throw network related exceptions, then this causes the exception types for the original method signature to be adjusted. This is problematic, if not impossible, if the source code is not available.

Last, but not least, sending messages over the network entails the concurrency issues described in section 7.3 because both parties are independent machines and thus continue execution. The position of many middleware solutions is that a concurrency model cannot be fixed upfront and therefore they offer different message passing frameworks. The flip side of this flexibility is complexity.

Conclusion

We conclude that building decent middleware is not an easy thing to do. Many design issues cannot be decided by the middleware because there is not a single solution that neatly fits in with the programming language at hand. Therefore, many middleware solutions will offer frameworks of solutions in which programmers can compose their solutions on a per-use basis. Needless to say, this raises the complexity of middleware. We therefore promote the integrative school of distribution in which programming languages can offer well-integrated distribution features that are cleanly integrated with the rest of the language.

7.4.2 Advanced Distribution Issues

The issues raised in the previous section are a strict minimum when considering distribution. This section sheds some light on more advanced issues which are beyond the scope of our work or which are deferred to later chapters.

Mobility

A topic often addressed in one breath with distribution is mobility. This is not surprising when referring back to the discussion on parameter passing. When sending a message to a remote object, the fact that parameters might need to be sent back and forth immediately raises the notion of mobility. Since mobility plays a central role in our work, we have devoted an entire chapter to it.

Replication

Another notion frequently recurring in the context of distribution is replication. The idea is to make copies of objects across a network in order to speed up object accessor to support failure recovery. Keeping replicas consistent upon state changes is not easy to accomplish when the replica management is not supported by the programming language. State changes (i.e., assignment instructions) have to be manually reported to the replica manager in order to get them propagated over the network. Another possibility is to align one's design to restrictions imposed by a middleware solution[4]. Replica management is beyond the scope of this dissertation, but we do have some ideas about this matter for future research. We will briefly touch upon them in section 10.3.2.

Partial Failure

A big difference between concurrent systems and distributed systems is the notion of partial failure. That is because a concurrent system will succeed or fail in its task as a whole, whereas a distributed system might partially fail while parts of it are still working correctly: although one device might crash, the rest of the system might still perform meaningful computations. Dealing with partial failure is therefore also one of the most important problems in distributed systems. It is also an *extremely* difficult problem as even its detection is not easy because the difference between a failure and a very long network delay is not clearly defined. In spite of its importance, partial failure was beyond the scope of our research in the context of this dissertation. In order to render the ChitChat model presented in chapters 8 and 9 practically applicable in a world of open networks from which devices can disappear without warning, partial failure will have to be taken in account. We have therefore categorized it as top priority in our future work section.

[4]A common technique is to require every class that can be subject to replication to be a subclass of a predestined "replication class".

7.5 Object-Oriented Distributed Languages

The previous section presented a general overview of distribution related problems and the way middleware solutions (do not) solve them. Just as with concurrency, the basic problem is that a middleware solution introduces a number of objects and/or concepts that solve the distribution aspects but which have little to do with the objects that structure the application semantically. This discrepancy poses many interaction problems which render middleware solutions very hard to use and extremely error prone. As with concurrency, our solution will therefore be to come up with a language dedicated to solve them. Of course we are not the first one to do this. We therefore give an overview of the most important existing languages in order to situate our work presented in chapter 8.

7.5.1 Emerald

Emerald [HRB+91] is a distributed programming language designed and implemented in the late eighties. Emerald is important in the context of our research because some of its goals such as the facilitation of distribution, were very close to ours. However, instead of being targeted at open networks with a dynamically defined topology, Emerald was mainly designed to support applications that have to run on a fixed and well-defined network topology. But even though Emerald's final target is different from ours, the philosophy behind Emerald is very close to the one we advocate, to wit that of a simple distributed and mobile programming language in which software developers can write "network aware" software without having to deal too much with the technical burden of things conceptually as simple as sending a message to an object that happens to be located on another machine. Of course this is not a tutorial on Emerald, so we merely focus on some important points of the language. In this section we focus on the distribution and concurrency aspects. Mobility in Emerald is treated in section 9.5.3.

Though Emerald does not feature any of the advanced features discussed in chapter 2, it is a prototype-based language in the sense that it does not feature classes and structures programs using objects only. Emerald is statically typed and its types are also first class objects. The combination of prototypes and static typing is possible because of the absence of dynamic features like delegation. The following code excerpt shows a small teaser taken from [RTL+91] that can be used to synchronize a software clock with a hardware timer interrupt.

```
const System <-
  object S
    monitor
      const timing <- Condition.Create
      % Tick is invoked by a hardware clock
      operation Tick
        signal timing
      end Tick

      operation Tock
        wait timing
      end Tock
    end monitor
  end S
```

Emerald has no class concept. Objects are created by evaluating so-called object constructors, expressions that evaluate to an object. Nesting such an expression inside a loop will generate objects that are similar. Although an Emerald implementation is allowed to optimise the structure of those "copies", there is no such thing as class-based sharing of code or state that can be detected from within the language. Emerald objects are completely self-contained. Their internal state can only be affected by message passing. However, as we will see in chapter 9, Emerald does not adhere to the extreme encapsulation principle because of the mobility operators defined on objects. Emerald does not feature inheritance. Its polymorphism is, amongst others, a consequence of the fact that radically different objects can *conform* to the same type. Classes can be mimicked in Emerald by nesting an object constructor inside an operation of another object constructor. Each time the operation is called, the inner object constructor is evaluated yielding an object. Variables "inherited" from the outer scope are treated as immutable constants, reminiscent to the way Java's `final` keyword avoids shared mutable memory between nested anonymous objects.

Emerald's concurrency model is based on a combination of active objects and threads. An optional *process block* can be added to an object. This is a chunk of code executing autonomously. By declaring an object as a monitor (as in the example), mutual exclusion of its operations is assured. Explicit synchronization is achieved by `wait` and `signal` which are similar to Java's concurrency control primitives `wait` and `notify`. Emerald threads are not limited to one object because it features synchronous message passing: the thread of control "flows" from the sender into and back out of the receiver. As such, Emerald features both inter-object concurrency as well as intra-object concurrency.

Emerald has features to deal with partial failures. One such feature is the `checkpoint` statement which saves the state of an object to stable storage to facilitate recovery after a potential crash. Another way to handle failures are *failure handlers*, comparable to the catch clause of the try-catch construction in Java or C++. However failure handlers are not associated with a particular method call or instruction group as in Java. Failures can be caused by network problems, division by zero, nil reference etc. Upon encountering such a failure, the appropriate failure handler is called. A special kind of handler is the "unavailable handler" which is used whenever an object is needed that can no longer be found on the network

Apart from the language, a good deal of effort was spent on the underlying implementation [BHJL86, JLHB88] which is responsible for ensuring Emerald's location transparent message passing. It is the responsibility of the kernel to locate the receiver and to handle the message properly after reception. One of Emerald's important characteristics facilitating this is the fact that Emerald objects carry a name that must be unique within the network. Referring to an object is done by mentioning its name. Apart from distribution transparency, Emerald also features object mobility. As we will explain in chapter 9 objects in Emerald can move around on the network both on the initiative of the Emerald program as well as on the initiative of the kernel. Although location is kept transparent, one can query an object for its location using the `locate` instruction. One reason for this is performance in the sense of reducing network traffic between two heavily cooperating objects. Another one is availability, an important concept in the proximity of partial failure. As we shall see in chapter 9, these locations are used in combination with Emerald's mobility instructions.

In order to finish the overview of Emeralds distribution model it remains to be said that mobility is used to improve argument passing semantics in Emerald as well. The sender of a message can decide to send the actual arguments by reference, "by move" or "by visit" via an appropriate annotation.

7.5.2 Argus

Argus [Lis88] is a distributed programming language that was explicitly designed to support programs that are supposed to maintain online data for long periods of time and for which reliability is a major concern. Examples are file systems, mail systems and inventory control systems. The following code excerpt taken from [Lis88] is meant as a teaser for Argus. It shows a procedure that transfers money from one account to another one. The accounts reside on different branches of the bank, which possibly reside on different nodes in the network. As can be seen from the code, Argus is statically typed and it features exception handling. Upon invoking `withdraw`, an exception might be thrown. The signature of the `transfer` operation and the code specify that the exception is simply propagated to whoever called `transfer`.

```
transfer = proc(from, to: account_nr, amnt: int)
                                 signals (insufficient_funds)
  f: branch := get_branch(from)
  t: branch := get_branch(to)
  f.withdraw(from, amnt)
       except when insufficient_funds:
           signal insufficient_funds
       end
  t.deposit(to, amnt)
end transfer
```

Argus is not object-oriented all the way down in the sense that it features primitive types, classes but no inheritance. One of Argus' main concerns is to deal with partial failures. To this end, two important concepts were added to the language, to wit *guardians* and *actions*.

Guardians are a special kind of objects that reside at one node at a time (but which may be moved). A guardian is in charge of encapsulating a set of resources which can be manipulated through special procedures called *handlers*. Handlers are to be thought of as procedures that can be called by other guardians over the network. Guardians can create other guardians over a network. The creator of a guardian decides its location which allows for dynamic updating of systems. Upon calling a handler, a new process is created that handles the call. Calling handlers is location independent: knowing the location of a guardian is not needed to call its handlers. Guardians manipulate several data objects which cannot be published on the network unless they are also guardians.

Argus is focused towards systems which have to be available for extremely long periods of times. To enable this, it features two kinds of variables, stable ones and volatile ones. The stable ones are replicated onto stable storage. Upon a crash, the volatile ones are lost but a recovery procedure that is spawned will restore the stable ones and recompute the volatile ones.

Communication between guardians happens in a message-based RPC way. Guardians are multithreaded which means that there has to be a synchronization mechanism to keep their internal state consistent. Communication between a caller and a called guardian proceeds via *promises* [LS88] which are not transparent but have to be explicitly fulfilled by the called guardian and be claimed by the caller. Nevertheless, they allow for maximal concurrency because the caller and the called guardian can continue their task as long as the calling one does not need the results of the called one to proceed correctly. This is very much the same as in ABCL, the concurrent object-oriented language we discussed in section 7.3.5.

As said, calling a handler of a guardian causes a new process to be forked. This means that there is quite a strong intra-object concurrency as well as inter-object concurrency. Argus does not feature any automatic synchronization mechanism. The programmer is completely in charge of keeping guardians consistent. To this extent, Argus offers the ability of using atomic transactions, also known as *actions*. These are chunks of code that are guaranteed to run total (i.e., all or nothing) and which are guaranteed to be serialized. The synchronization mechanism is implemented via locks.

7.5.3 Obliq

Obliq [Car95] is a distributed language developed with the purpose of writing computations that can roam networks. Obliq objects are records of named slots of the form { `x => ..., m => ..., ...`} where each slot is either a value, a method or an alias to a slot of another object. Obliq is dynamically typed prototype-based in the sense that it does not feature classes. However it does not feature delegation or object-based inheritance either! Instead it has many of Kevo's characteristics (see section 2.4). Such languages in which new objects are created by adding attributes to copies of existing objects are called *concatenation-based languages*. In Obliq this takes the form of quite a powerful cloning operator which clones an object and possibly adds new slots to the clone. The following Obliq teaser defines two objects, the second one of which is based on the first one by concatenation.

```
let unidirectional =
  { x    => 3,
    inc  => meth(self,y) self.x := self.x+y; self end,
    next => meth(self)   self.inc(1).x end };

let bidirectional =
  clone( unidirectional,
  { dec  => meth(self, y) self.x := self.x-y; self end,
    prev => meth(self)    self.dec(1).x end });
```

Obliq heavily revolves around lexical scoping. A unique feature is that lexical scope is preserved when an object is copied across network nodes such that its free variables keep on referring to the same resources as the ones they referred to in the scope in which the object was created. A common critique of Obliq is that this entails a lot of network overhead when computations refer to variables that belong to their lexical scope residing on a different machine.

Besides cloning, Obliq has selection/invocation, assignment and aliasing operators to manipulate objects. The selection/invocation operator selects the value from an object's slot and invokes it in the case of a method. Notice that methods always have at least one parameter which is the name of the receiver itself, allowing programmers to use a different identifier than `this` or `self`. The assignment operator overwrites a slot of an object with a new value. This allows for a very limited form of inheritance that excludes common things such as super sends and late binding of self. The aliasing operator takes the form `x => alias y of b` to specify that the slot `x` shall henceforth be an alias of slot `y` in object `b`. This redirection can be defined recursively which allows for automatic message forwarding. However, it is important to notice in the light of the discussions of chapter 2 that this forwarding mechanism does *not* take late binding of self into account. Hence, Obliq features no real delegation.

Obliq is important in the context of our research in the sense that is fully confirms our amendement to the Treaty of Orlando presented in section 4.6.5. Remember that a self-representation model was added to the Treaty of Orlando to correctly classify languages. Obliq endorses this by defining *self-inflictedness* and *self serialization*, two dynamically verified concepts conceived on the basis of the self representation of objects. The idea is to call any of the four operations defined above self-inflicted if it is applied to an object which is the same as the "self" of the currently executing method. The reason for defining this is that objects can declare themselves as protected {`protected,...`}, serialized {`serialized,...` } or both {`protected, serialized,...` }. The idea of a protected object is that, apart from selection/invocation, it will only allow self-inflicted operations. This means that by declaring itself protected, an object can preclude other external users from cloning it, from updating it and from aliasing it. Only message passing is possible! This is precisely what was called extreme encapsulation in section 3.4.4. Hence, Obliq endorses this principle. However, it does this in the absence of delegation with late binding of self.

Based on the notion of self-inflictedness is the notion of self serialization which is an important concept in Obliq's thread-based concurrency model. It uses the operations `fork` and `join` to spawn new threads and to make a thread waiting for another one to end. As we have explained in section 7.3, a thread-based model results in race conditions when multiple threads concurrently "enter" the same object. In order to render such operations mutually exclusive, the object can use the `serialized` modifier as explained above. In order to avoid deadlock upon recursive method calls, serialization is defined as self serialization based on the notion of self-inflictedness. The idea is that self-inflicted operations will not try to acquire a lock and will hence not cause deadlock.

Obliq's behavioural synchronisation (see section 7.3.2) is based on so-called condition variables `c` and allows programmers to attach guards to them using the `watch c until guard end` statement which will cause the current thread to block as long as the `guard` is false. `signal(c)` causes the guards to be reconsidered which may result in a continuation of their thread.

In contrast to Emerald and Argus, Obliq's distribution model is applicable for networks the topology of which is not known upfront. Obliq locates objects using a central name server. Threads can register objects with the name server under a certain name using `net_export("objectName",DNS,object)` and threads located elsewhere can query the name server to see whether it knows about objects that have been registered under a certain name, using the operation `net_import("aCertainName",DNS)` whose result is an object. Based on these primitives [Car95] explains the notion of *execution engines* which are objects that accepts a lexically scoped procedure and that will call the procedure with local resources such as a local database or file system. Since this is the only way Obliq computations can access remote resources, security is guaranteed.

Obliq is not a mobile language in that it does not have built-in mobility primitives and that arguments of a remote message are passed by reference upon. However, the combination of cloning and aliasing allows protected serialized objects to make a remote copy of themselves and redirect all operations to that copy by means of one single atomically executed method which is entitled to do this because of self-inflictedness. We will get back to this in chapter 9.

7.6 Evaluation And Epilog: Open Networks

Now that we have presented the most important distributed object-oriented programming languages we can evaluate them in the context of the open networks we envisioned in chapter 1.

7.6.1 Emerald

As we saw in section 7.5.1, Emerald is an object-based language that features distribution transparency and object mobility. In section 3.2 explained that

Emerald's absence of class-based implicit sharing relationships between objects is indeed an absolute must in our research context. But there are also a few language features of Emerald that render it hard to use in the context of open networks without a predefined topology. Although it may be possible, we currently believe that devising a type system to statically type applications with the dynamics described in section 1.1.1 will be extremely hard. The idea of AmI is that devices cooperate with other devices the interface of which was not known in advance. But a more fundamental problem with Emerald in our context is that its concurrency model is essentially based on synchronous message passing. This will be very hard to defend in dynamic networks because it is very hard to detect the difference between devices that are answering late and devices that have moved out of earshot. An asynchronous communication mechanism will clearly outperform this because the independence of devices does not extoll one device to wait for another one if this is not strictly necessary.

7.6.2 ABCL

The ABCL model described in section 7.3.5 is extremely well-suited for our purposes, and as we will see in chapter 8, much of its features have found their way in our ChitChat model. Its most important features that we have incorporated as well is its clean concurrency model based on asynchronous message passing and futures, although these futures are not transparent to the programmer. On the downside, ABCL is a concurrent programming language and not a distributed one. The interaction of its features with the explicit notion of dispersed computation with shared resources has never been investigated. This is one of the contributions of this dissertation.

7.6.3 Argus

Just like Emerald, Argus is a statically typed language which is arguably problematic in the context of open networks. Apart from the concurrency primitives that allow programmers to use ABCL-like promises, the concurrency features in Argus are pretty cumbersome to use. Argus gives the feeling of being much more focused towards systems that have to deal with huge amounts of data kept on external storage media, instead of dynamically defined network nodes. Nevertheless, Argus does allow guardians on one node to specify the spawning of guardians on other nodes which enables Argus to be used in dynamically configurable environments. An innovative feature in Argus is the all-or-nothing transactions it offers to a programmer to guarantee the atomic execution of some given code fragment in the context of data shared between different concurrency handlers of a guardian. In section 8.4.3 we will incorporate this feature in our own ChitChat model to regulate atomicity of code transactions from descendants in a shared parent.

7.6.4 Obliq

Obliq is very interesting in the context of our research. Just like the ChitChat model proposed in chapter 8, Obliq is a prototype-based language. However, instead of featuring advanced sharing mechanisms based on delegation or object-based inheritance, it features stand-alone objects that can only share resources through message passing. Moreover, the notion of self-inflictedness aligns really well with our own observations about the relevance of the self representation model in our improvement of the Treaty of Orlando presented in section 4.6.5. The basic idea is the same: protected objects are only subject to message passing and field selection but can still apply a multitude of operations to themselves. On the downside, Obliq will not be applicable for dynamically defined networks because of its centralized name server approach used for initial object referencing and, more importantly, because of its thread-based concurrency mechanism with synchronous message passing between remote objects. If it comes to the concurrency model, it shares the problems of Emerald.

7.7 Conclusion

The purpose of this chapter was to give an overview of the enormous domain of concurrency and distribution in object-oriented programming languages. After having given an overview of the general difficulties introduced by concurrency, we have presented the three fundamentally different ways of reconciling concurrency with object-oriented programming, to wit the applicative (i.e., middleware), the reflective and the integrative (i.e., languages). The middleware approach seems to be extremely popular among practitioners these days which is quite painful a situation as it is arguably also the toughest way one can imagine writing stable concurrent and distributed applications. Especially in the field of distribution, the applicative approach performs poorly as explained in this chapter. That is why we have argued in favour of the integrative approach. For both concurrency and distribution, an overview of the most important languages was presented. It seems that, with the exception of Argus[5], many such languages are prototype-based. Neither ABCL, nor Emerald and Obliq have classes.

However, chapter 2 has shown that prototype-based languages have a lot more to offer than mere objects and message passing, notably that idiosyncratic objects can share data and behaviour through shared parents in a delegation relation. None of the languages discussed here promote this. Maybe this is not accidental: in chapter 4 we showed that conventional prototype-based languages suffer from inherent encapsulation problems, which renders them unusable in dangerous environments which the open personal area networks described in chapter 1 will be. To counter this, chapter 4 formulated the extreme encapsulation principle and chapters 4 and 5 have shown that, even with this principle in mind, a wide range of delegation-like techniques are possible. The chapters

[5]Argus has no inheritance.

to come will exploit this in the context of distribution and mobility. Chapter 8 presents a highly expressive secure distributed version for Pic% that features techniques such as networked inheritance and networked parent sharing to enhance sharing of resources between distributed parties.

Chapter 8

ChitChat: Delegation-Based Concurrency & Distribution

In chapter 7, we have evaluated existing concurrent and distributed object-oriented programming languages in the context of the open networks as envisioned in chapter 1. The majority of these languages is classless, but as summarized in section 7.7 none of them exploits the full power of the prototype-based languages reviewed in chapter 2. This chapter introduces ChitChat, a distributed version of Pic% that explicitly extols prototype-based parent sharing in the context of concurrency and distribution. ChitChat is a powerful prototype-based language that fullfils the extreme encapsulation requirement put forward in chapter 3 and that renders the techniques distilled in chapters 4, 5 and 6 beneficial to concurrent and distributed programming.

8.1 Introduction

In section 1.4 of the introduction, we have formulated four fundamental issues with regard to concurrency, distribution and mobility in the context of open networks. Surely, solving these issues is not enough to serve as a production language for AmI programming but we claimed they *will* have to be solved.

One of the most fundamental problems was the Ambient Object Paradigm Problem, presented in chapter 3. It basically boils down to the fact that class-based programming languages have fundamental paradigmatic problems for mobility and open networks, and that prototype-based languages have encapsulation problems that lead to unacceptable security breaches. In chapter 4 we have resolved this stalemate by distilling a full-fledged programming language that adheres to our extreme encapsulation principle but which still features the full power of prototype-based languages described in chapter 2. In chapter 5 we have evaluated Pic% as a lightweight contemporary incarnation of this model which is — in many respects — even more powerful than the original Agora model.

In this chapter we take up the thread of the narrative started in section 3.2 where we have recommended prototype-based languages in the context of open networks and mobility. We do so by taking the Pic% model and augment it with distribution in this chapter and mobility in chapter 9. However, as explained in chapter 7 distributing an object model across open networks is nearly impossible without a decent concurrency model to support it. So we both need a concurrency and a distribution model for Pic%.

In the spirit of our vision on language design outlined in section 1.2, we did not boldly build a notion of threads together with some synchronization constructs *on top of* Pic%. Instead we have redesigned Pic% to align its peculiarities with the notions of concurrency and distribution. The outcome of this research is a unique distributed programming model, called ChitChat, that has the following properties:

- ChitChat is a concurrent and distributed extension of Pic%. It stays faithful to Pic% in the sense that is is a classless object-oriented programming language with the full power of prototype-based languages explained in chapter 2, but that respects the extreme encapsulation principle.

- The concurrency and distribution features are an intersection of the desirable properties of the "champions" Emerald, ABCL, Argus and Obliq we identified in chapter 7. It supports asynchronous concurrency and distribution transparency in the sense of section 7.4.1.

- The model is the first proposal that reconciles concurrency and distribution with delegation in prototype-based object-oriented programming language, a marriage that was considered impossible until now [BY87].

- Our model shows that objects themselves can be shared over different nodes of a network. This intra-object distribution, combined with the concurrency model, will be shown to have some very powerful properties with respect to sharing resources over a network.

- In chapter 9, it will be shown that mobility fits very well with the model of concurrency and distribution presented here.

We have fragmented the description of ChitChat along logically delimited sections. In section 8.3 we explain how concurrency was introduced in ChitChat by adding ABCL's active objects to Pic%'s passive object model. Section 8.5 describes the (sometimes subtle) interactions between active and passive objects both in the light of message passing as well as in how they interact in the presence of inheritance hierarchies that can be distributed over different machines. The distribution model that naturally follows from this concurrency model is the topic of section 8.6. Section 8.7 evaluates the proposal. However, before we delve into the technicalities, we offer the reader an understanding of the big picture of ChitChat in the section 8.2.

8.2 ChitChat in a Nutshell

The distribution and mobility related problems of class-based and classic prototype-based languages in the context of open networks outlined in sections 3.2 and 3.6 made us look for distributed prototype-based languages that strictly adhere to the extreme encapsulation principle. As shown in chapter 7, Obliq is such a language. However, Obliq uses centralized name servers, works with synchronous message passing and has very poor sharing mechanisms in the sense of chapter 2. As explained above, ChitChat is a highly integrated distributed and mobile programming language that has no classes, adheres to the extreme encapsulation model but still features the powerful mechanisms outlined in chapter 2.

The ChitChat concurrency model was strongly inspired by the one from ABCL. It features active objects that process queued messages one after the other, excluding intra-object concurrency. Message passing is asynchronous, but just as in ABCL, concurrency is maximised by immediately returning a promise to the sender of a message. In contrast with ABCL, promises are transparent. Neither sender, nor receiver has to claim and/or fulfill promises explicitly. The sender simply blocks whenever it tries to use the promise's value before it is known. ChitChat's behavioural synchronization is taken care of by an innovative mechanism called *call-with-current-promise*, much in the spirit of Scheme's continuations. Active objects are capable of grabbing the promise they are about to fullfil, store it and fullfil the promise manually, at a later moment, possibly after some conditions have been met.

In contrast to ABCL, ChitChat combines active objects with a delegation-based inheritance scheme. Moreover, as in the prototype-based languages of chapter 2, an active parent can be shared by many active descendants. The philosophy is to let the shared parent be the representative of the state they share. The delegating active objects can access their shared state in their parent. But in order to avoid race conditions that might occur if two or more active descendants change the shared state, Pic%'s scoping was slightly restricted to prevent descendants from accessing the parent state in uncontrolled ways. To compensate for this visibility restriction, a "serialized super send" construct allows expressions to be executed atomically in the context of the shared parent.

This model of concurrency in which active objects hold state shared by active descendants very naturally fits distribution as well. The idea is to allow the active descendants to reside on machines that are different from the one the parent resides on. Hence, the ChitChat model of distribution allows parts of objects to reside in different locations. Message sending to a distributed object can give rise to a delegation process that proceeds across the network. In what comes we describe this model in detail and give extensive motivation for it. For the sceptic reader ("You don't want to generate method lookup over a network!") we already mention that classic proxies as promoted by middleware solutions are objects that *always* delegate *every* message over the network. They are the poor man's version of the scheme we propose.

The ChitChat distribution model can be thought of being centred around *Connected Applets* where a server can be remotely asked for an applet by sending it a message. If this message is implemented as a view, this gives rise to an object being created (the "applet") that resides on the client and that delegates all its non-implemented messages to the server. At all times, the client object is an active object that is completely independent from the other active objects (i.e., connected applets) that were spawned by that server. All applets are running concurrently and have controlled access to the state they share (on the server). We claim that this model of connected applets is *extremely* useful considering the percentage of Java applets that, upon arrival, immediately establish a connection with their server in order to transfer data in one or two directions. A real life example is the currently extremely popular Volano-chat [vol] applet which immediately connects to its spawning server upon arrival. In section 8.6.5 we will show how ChitChat allows this chat to be implemented as a local view on a shared chat server object.

8.3 Active Objects & Synchronization

As was already mentioned several times, the ChitChat model was strongly inspired by ABCL [YBS86]. A ChitChat active object is conceived as the combination of a "passive object", a waiting queue in which its received messages are scheduled and an eternally running thread that consumes the messages in the queue *one after the other*. This means that ChitChat does not know intra-object concurrency: inside an active object, only one method can be running at a time. Figure 8.1 shows how we depict such an active object in the remainder of the dissertation. The thread is depicted by a spool. For the sake of the argument, the role of delegation is postponed. The passive object defining the behaviour of the active object can be thought of as a regular Pic% object as discussed in chapter 5. The following code excerpt shows a ChitChat program that calculates fibonacci numbers concurrently.

```
aview.fib(n)::{
  do()::if(n<2,
           1,
           athis().fib(n-1).do()+athis().fib(n-2).do())}
```

The code shows an "active view method" called `fib` (in analogy with `view` as discussed in chapter 5). Invoking it yields an active object which contains a single method `do`. In analogy to the self reference `this()` used to cause recursion in passive objects, active objects send themselves an (asynchronous) message by referring to `athis()`. Hence in the above code excerpt, every invocation of `do` will cause two new active objects to be created which immediately receive `do` in turn. Every such invocation of `do` returns a promise to its caller. Synchronization is entirely accomplished by the `+` operator in this example. It will block until the promises making up its arguments are both fulfilled.

Hence, as in the actor model, messages are sent asynchronously and the sender does not wait for the result of the message. But in contrast with the bare actor model, a connection between the sender and the receiver is established

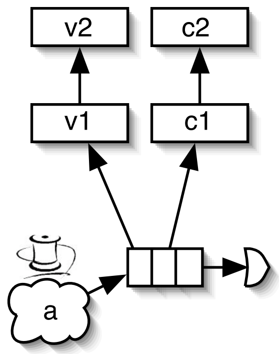


Figure 8.1: ChitChat Active Object

by means of a *promise* that is immediately returned by the receiver. This can be thought of as a placeholder that represents the return value. In case the method delivering the promise has finished, the promise *is* the return value. Otherwise, anyone trying to manipulate the promise directly is blocked until the promise is fulfilled. In other words, promises are completely transparent in ChitChat.

Because of the combination of asynchronicity and actual arguments evaluation semantics, the queue does not really contain the names of the message but method activations instead. At the time of the message sending, the message is immediately searched for in the object. The corresponding method is selected and enqueued together with the binding of actuals to the formals. Subsequently a promise is returned to the sender. Another issue in the context of asynchronicity is that message ordering becomes relevant. To ensure "logical" message ordering we have adopted Yonezawa's *Assumption of Preservation of Transmission Ordering* [YBS86] which basically states that messages ordering between two objects is preserved.

8.3.1 Behavioural Synchronization in ChitChat

Referring back to figure 7.1, after having explained ChitChat's intra-object synchronization (i.e., full serialization) and its inter-object synchronization (i.e., synchronization upon promise consumption) we still have to focus on its behavioural synchronization mechanism. Behavioural synchronization is what makes objects temporarily enable or disable some of their methods depending on their internal state. In ChitChat, this is handled via an innovative technique called **call with current promise**, developed by Van Cutsem and Mostinckx in their graduation thesis [VM04]. The idea is very similar to Scheme's "call with current continuation" and is based on the fact that the interpreter always has a "current promise" at hand, to wit the promise of the last sent asynchronous message. In passive objects, this is the promise of the "last" active object whose method gave rise to sending a message to that passive object. We postpone a full description of the interaction between active and passive objects to section 8.5. For now it suffices to say the message sending semantics of Pic% does not change. A message sent to a passive object is always synchronous and does not affect the "current promise".

In analogy with `call` and `continue` needed to grab and restore "current continuations" as discussed in section 5.5.7, ChitChat features the primitives `delay` and `fulfill`. The formal function heading of delay is `delay(exp(promise)):...` so that it is to be called with an expression in which the variable `promise` is dynamically bound to the current promise. `delay` never returns and the promise of the method in which it occurs is thus not fulfilled. Hence the object waiting for that promise to be fulfilled can be (temporarily) waiting. The philosophy is that the expression used in the call to `delay` stores the promise somewhere for later usage when the conditions to continue the execution are met. At that time, a method can "pick up" the promise and fulfill it manually by calling `fulfill(p,v)` where `p` is the stored promise and `v` is the value the promise will be fulfilled with.

Van Cutsem and Mostinckx [VM04] show how this mechanism can be used to implement semaphores, CSP-like [Hoa78] rendez-vous primitives and how ParLog's [Cla88] or-parallellism can be implemented. In the following example we show how Pic%'s language extension mechanisms explained in section 5.6 is used to implement an Obliq-like guard-based behavioural synchronisation mechanism using current promises.

```
view.guard()::{
  prom:void;
  t:void;
  e:void;
  waitIf(test(),effect())::
    if(!test(),
      effect(),
      { e:=effect; t:=test;
        delay(prom:=promise) });
  signal()::
    if(is_void(prom),
      void,
      if(!t(), { tmp:prom;
                 prom:=void;
                 fulfill(tmp,e())}))}

aview.buffer(g)::{
  elms:void;
  produce(el)::{
    elms:=[el,elms];
    g.signal()};
  consume()::
    g.waitIf(is_void(elms),
{ tmp:elms[1];
  elms:=elms[2];
  tmp})};
```

A (regular) view method `guard` is shown whose invocation will yield a (passive) object in the sense of chapter 5. A guard `g` is passed along upon constructing a buffer with the active view method `buffer`. Every time some object tries to consume from an empty buffer, the promise is captured in the guard. Upon producing a value by another object, that promise is fulfilled so that the consumer can proceed. Although this example is pretty simplistic, the guard could be made much more complicated (e.g. keep a list of promises). We have shown it because it effectively shows how ChitChat's language extension mechanisms (i.e., Pico's parameter passing semantics) is combined with the very general promise capturing mechanism to build behavioural synchronisation abstractions such as Obliq's `waitIf` and `signal` illustrated above (see section 7.5.3 for a description).

8.3.2 Active Closures and First Class Methods

In section 5.8.3 we have thoroughly discussed that the combination of late binding polymorphism, lexical scoping and first-class methods gives rise to an object model in which the methods reside "contextless" in objects. Upon method lookup, a method (i.e., the body code together with formal arguments) is turned into a closure consisting of the lexical object frame in which that method resides, the method itself and the "receiver" from which the method was selected (which will be the value of `this()` upon invocation). In short, Pic% functions are actually closures emerging from first-class method selection.

This model is naturally transposed to the concurrent setup discussed here. The only difference is that, as we will discuss in section 8.5, the delegation hierarchy consists of chains of active objects. The result of selecting a first-class method from such an active object is called an *active closure*. It consists of the method itself, the active object into which the method was found and a reference to the active object that was the actual "receiver" of the method selection expression of the form `active.m`. Upon calling the function formed by this active closure, the actual arguments are bound to the formal ones following the regular Pic% semantics. The resulting activation is enqueued in the queue of the active object that contains the method in the first place. Hence, this is a direct generalisation of Pic% semantics to the active object model.

8.4 Pic% Object Model and Scoping Revisited

Although ChitChat's scope rules are quite natural and usually behave as one would expect from looking at the source, they are actually quite subtle.

8.4.1 Return to Lexical Scope

In section 5.8.3 we have discussed two scoping options that fit well in the Pic% object model because of the alignment of dictionaries (i.e., lists of frames containing name bindings) with objects. We saw that, when calling a method on an object, the frame binding formals to actuals can be either attached to the receiver (yielding some form of dynamic scoping because the receiver is dynamically determined), or to the frame in which the method was found. In the "original" Pic% proposal published in [DD03b] the first option was defended. However, the model is not reconcilable with the concurrency model based on parent sharing we are about to propose. Indeed, imagine two active objects that share a parent with mutable state. In order to prevent race conditions caused by those two descendants, we have to restrict the visibility of mutable state in the parent from within methods running in the descendants. The rule that a method residing in a descendant can always "automatically see" any variable along the delegation chain is simply too flexible to enable concurrency control. Now suppose that we adopt the dynamic scoping scheme in which a method that resides in the parent is executed in the context of an arguments binding frame that is attached to the dynamic receiver. Because of the scoping restrictions in between descendant and parent frames this would imply that the method (found in the parent) is no longer entitled to see "its own scope" because this scope is accessed by following the chain of frames consisting of the arguments binding frame, the descendant frame (i.e., the receiver) and the parent frame. We have therefore decided to **re-adopt a variant of the classic lexical scoping scheme** in which the scope of a method is the frame in which the method was found during method lookup and not, as in the original Pic%, the frame corresponding to the receiver. Of course, the meaning of `athis()` refers to the dynamic receiver as expected.

8.4.2 Internal Object Scope: Frames $\neq$ Objects

Apart from dynamic scoping, another track we had to leave due to concurrency control is the complete alignment of dictionary frames with objects. As explained above, potential race conditions cause us to restrict the visibility of variables between active parent objects and active descendants. For constants (i.e., names declared with a `::`) scoping can be completely free because these can only be read and can therefore never give rise to race conditions. Hence, all possible **frames** in a list of frames can safely **delegate all** requests to lookup **constants**. For variables the situation is more complicated. Note that in the Pic% model, the existence of a frame can actually be traced back to the result of an object creation[1] or to the result of parameter bindings. Since the scope of a method is a combination of both, care has to be taken because a method has to be able to "see" its slots. Hence, **variables** have to be **delegated** by the actuals-formals **frame**. However, they may **not be delegated** by the descendant **objects** towards the parent for otherwise race conditions could arise.

[1]In Pic% this is done by calling `capture()`

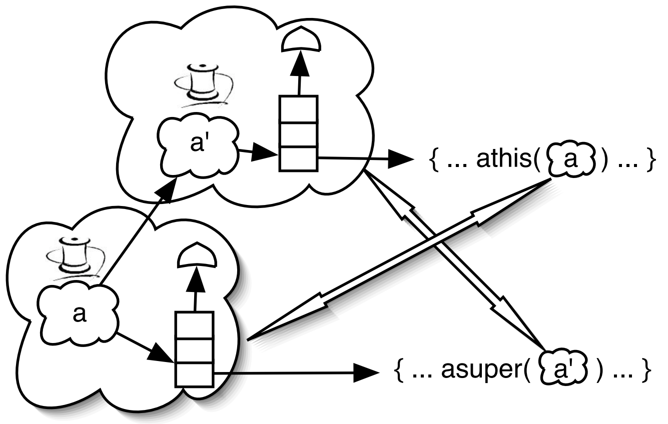


Figure 8.2: ChitChat Active Scope Functions

This means that there is a conceptual difference between frames and objects: frames delegate both the lookup for variables and constants to their parent while objects never delegate the lookup for variables. Hence, in Java terminology, ChitChat variables are "private" and not "protected" as in Pic%. The reason is race condition prevention.

8.4.3 Scope Functions and Parent Sharing

In the course of the dissertation, we already promised repeatedly that it was our explicit goal to exploit parent sharing to manage data needed by two concurrently running descendant processes, and, to physically share conceptually shared data between distributed entities. To make this possible, ChitChat features native functions `asuper(...)` (resp. `athis(...)`). In contrast to the native functions `asuper()` and `athis()` which simply *return* an object, these functions *expect* an expression that is evaluated **in the context of the parent object** (resp. in the context of the receiver) in an atomic way. Because of the scope rules defined above, descendants (resp. parents) cannot cause race conditions on variables residing in a shared parent (resp. descendant), simply because they have no access to them. But they *can* send code "upwards" (resp. "downwards") that will be run by the parent (resp. descendant) atomically. This is schematically depicted in figure 8.2. For example, suppose that a shared parent contains a variable `x`. A descendant can use the expression `asuper(x:=x+1)` to make sure the parent variable gets incremented. It will be scheduled in the queue of the parent and will be run in its turn, without interruption from any other message or any other such super call.

8.4.4 An Experiment: The Dining Philosophers

The model is illustrated by the following experiment. It is an implementation of the famous dining philosophers example in ChitChat. The code uses the guard implementation of section 8.3.1 to make active objects wait and continue the execution of a method based on condition variables.

```
aview.Table(n)::{
  forksDown[n]:true;
  guards[n]:this().guard();
  aview.Seat(i)::{
   sitDown()::
     asuper(guards[athis(i)]).waitIf(
           asuper(if(forksDown[athis(i)]&
                     forksDown[athis(i\\n+1)],
                    { forksDown[athis(i)]:=false;
                      forksDown[athis(i\\n+1)]:=false;
                      true})),
                    void);
   getUp()::
     asuper({forksDown[i]:=true;
             forksDown[i\\n+1]:=true;
             guards[i].signal()})}  }

aview.Philo(i,t)::{
  s:t.Seat(i);
  work()::{
    display(i," is thinking");s.sitDown();
    display(i," is eating");s.getUp();
    athis().work()}  };

table:athis().Table(n)                          ‘create table

philos[(i:0)+n]:athis().Philo(i:=i+1,table)      ‘create philos
for(i:1,i<n+1,i:=i+1,philos[i].work())       ‘make philos work
```

The `Table` active object constructor is called to create a table of `n` seats. Every seat is a view on the table that implements two methods `sitDown` and `getUp`. The table contains an array of forks which are just booleans, and a Pico table of guards that keep philosophers from sitting down if both forks are not available. A philosopher is an active object that has a reference to its seat. Its work consists of thinking and eating till eternity.

The interesting part of the code is the body of `sitDown`. The seat takes a reference to "its" guard and evaluates the test for that guard. If the test yields true, the method blocks. The test is the atomic execution of the code consisting of the availability tests of both forks, and, if positive, the code to pick them up. Since this code is delimited by `asuper(...)`, it is atomically executed in the context of the table. In the course of its execution no other process can run inside the table. Notice that, in the opposite direction, `athis(...)` has to be used to access variables residing in the seat (such as `i` and `n`) while running code in the parent.

This example illustrates the ChitChat concurrency model very clearly. The idea is that the shared state of a system is placed in the parent object of the constituents of the system. In our case, the seats are objects which are a philosopher's view on the table parent. Every seat works independently but accesses the shared state in the parent using the scope functions `asuper(...)` and, the other way around `athis(...)` if necessary.

8.4.5 Deadlock and Parent Sharing

One of the rough edges to the proposal is that combinations of `asuper(...)` and `athis(...)` can easily lead to deadlocks. The situation occurs when using `asuper` in a method that actually needs the result of the returned promise, and if the call to `athis` also needs the result to be able to fulfill the promise associated with the super call. Hence, the problem is likely to occur if both the call to `asuper` and the call to `athis` do not occur in tail position in the code. If both calls occur in a tail position it is guaranteed to be no problem because the actual result is not needed in either case then.

8.5 Active vs. Passive Objects

ChitChat has both active and passive objects. Chapter 7 has extensively motivated active objects in the context of open networks. A concurrency model based on passive objects and thread synchronization is not a viable option. It would impose client-server schemes in which device independency would have to be encoded manually by forking special communication-specific threads as is implemented by Java RMI. However, turning *every* object into an active one requires too heavyweight a machinery because of the representational and computational cost induced by active objects [BGL98]. That is why we have opted for both active and passive objects. This section described how their semantics interact.

8.5.1 General Object Structure

The philosophy of the ChitChat distribution and concurrency paradigm is that passive objects should[2] only be used within the boundaries of an active one, pretty much in the spirit Argus distinguishes between "ordinary" data objects and guardians. Since we decided not to provide all passive objects with internal

[2]Note that this is not forced by the language. We explain this as the philosophy of the language. Some problems in ChitChat require heavy hand-coded solutions but are easily avoided by sticking to this philosophy. It is just like Smalltalk classes are also objects but no one would use them actually as "real objects" used to model a domain.

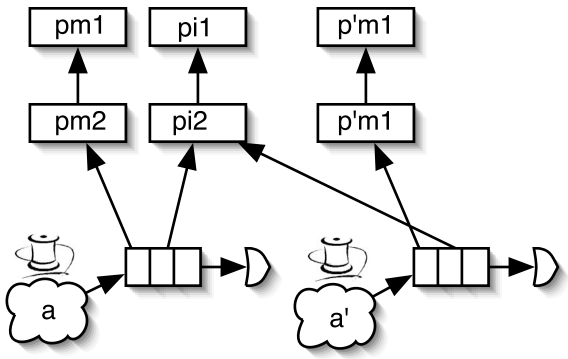


Figure 8.3: ChitChat Active Object Structure

serialization machinery, making a passive object accessible from within two different active ones is asking for race conditions. To prevent this, passive objects *can* be turned into a monitor by sending them the `serialize()` message which associates them with a reentrant mutex. The result is a serialized object, very much in the spirit of Obliq's self-serialization with the {`serialize,...`} construct as explained in section 7.5.3. In conclusion, ChitChat active objects are always serialized (i.e., no intra-object concurrency) and passive ones are never serialized unless they are explicitly sent the `serialize()` message. The latter can be done by sending `this().serialize()` inside the view method that creates the passive object. This is exactly the self-serialization as promoted by Obliq.

A passive object is a Pic% object: it is a list of frames consisting of a public immutable part and a private part in which the mutable state resides. The frames emerge from successive method invocations and view constructs, as in Pic%. We refer back to figure 5.3 that gives a schematic overview of a typical passive object. As explained in section 8.3, an active object can be understood as a passive object together with a queue and a computational thread. Furthermore, cloning semantics is the same as in Pic%. This is depicted in figure 8.3[3].

8.5.2 Distribution Driven Hierarchy Restrictions

One of the fundamental assumptions made by our research is that ChitChat **never introduces networked links that refer to passive objects**. That is because passive objects do not have thread machinery such that any message sent to them would be sent synchronously, an option we have ruled out in section 7.3.4 for our context of open networks. Therefore, *all* network trafic

[3]Some proofreaders were tempted to think that an active object somehow "wraps" a passive one. This is not the case. An active object contains a passive part but one type of object can never be converted to another one. Active objects are active and passive ones are passive.

that causes passive objects to cross network boundaries will always (deep!) copy those passive objects. This will be the case for parameter passing, return value delivery, as well as for the strong mobility provisions we will introduce in chapter 9. Our model does not allow references to passive objects over a network. This is very fundamental an assumption that has shaped the entire object model of ChitChat. Investigating the consequences of removing this assumption has not been thoroughly undertaken. It would probable change the object model as radically as Euclidean Geometry changes to Saddle Geometry by removing the assumption that two parallel lines have no point of intersection. It is an interesting topic for future research.

The fact that network references never refer to passive objects has important implications for the way delegation hierarchies can be structured by ChitChat's language features. Remember that it is our explicit goal to have multiple active objects on different machins to share the same parent. This means that active objects should not have passive parents. We might have the reaction to relax this restriction somewhat and allow active objects to have passive parents as long as they reside on the same machine. But this would not lead to a clean semantics when combined with mobility in chapter 9. Indeed, moving the active object would either have to copy the passive parent, or, to create a network link that points to a passive parent. We therefore postulate that

active objects should never have passive parents

8.5.3 Delegation Driven Hierarchy Restrictions

Another restrictive force in the design of ChitChat's object model is the delegation semantics caused by method lookup. Suppose we have an object o and we send it a message $o.m()$. Now let us consider what would happen if the delegation chain of o would consist of several parts (resulting from successive object extensions) such that the parts are arbitrary combinations of active and passive objects. For the sake of the analysis, say o has a parent p and suppose that m sent to o is found in p. Let us look at the three remaining combinations that can occur:

- Both o and p are passive. This is the standard Pic% semantics and does not give any problems.
- o is active and p is active. Again, this is not a problem. The message is looked for along the delegation chain and the activation is put in p's queue. One of the reasons for choosing p's queue is that o and p might reside on different machines.
- o is passive and p is active. Now we have to wonder whether this (as a whole) represents an active or a passive object. I.e., do we want m to run immediately (because it was actually sent to o, a passive object), or do we have to schedule it in p's queue (because it was found in p!). Neither is satisfactory semantics. Surely, we cannot run the method immediately

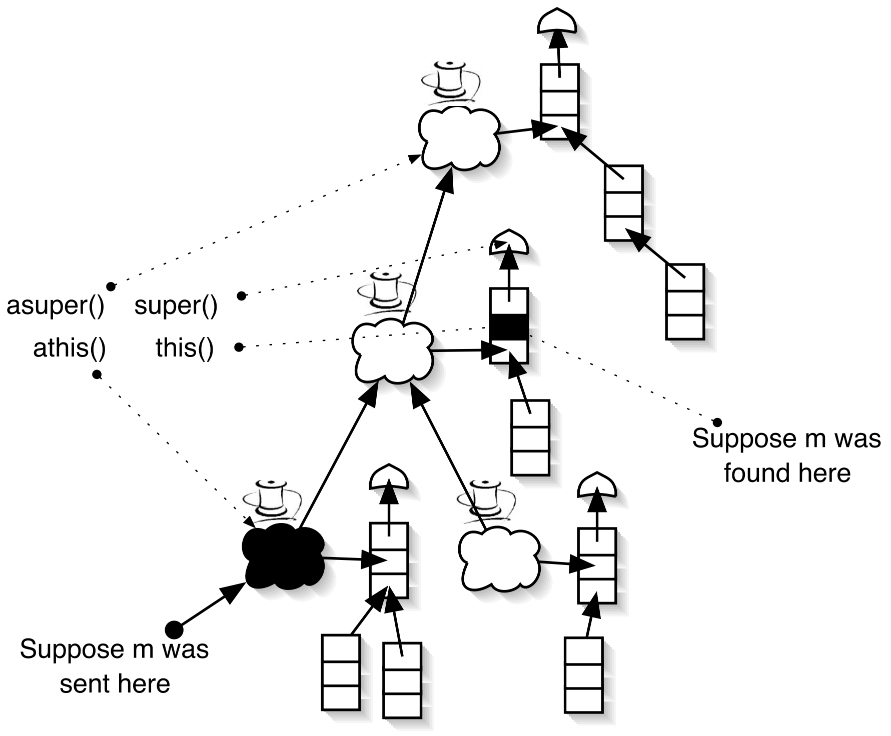


Figure 8.4: ChitChat Object Hierarchies

as this would break the scheduling semantics of active objects. Active objects would no longer be serialized as two methods might be running at the same time which might cause race conditions. But the second option also yields some unexpected results. The problem is that, upon sending a message to a passive object, the sender of m expects to be dealing with a passive object and thus to cause synchronized message passing. However, instead the thread of the sender will end up in the queue of the active o.

This analysis forces us to postulate that

passive objects should never have active parents

8.5.4 Method Lookup and Method Execution Context

The typical structure of a ChitChat object is depicted in figure 8.4. The active parts are descendants from each other (possible across the network). Passive objects can also form a hierarchy but never across network boundaries.

Figure 8.4 also shows what happens when sending a message to an active object (indicated by the solid black object). The message is searched for along the inheritance hierarchy (possibly across the network). In every active object,

kind of receiver:	$A.m()$ (A is active) $A = a^1_{f_1}...a^N_{f_N}$	$P.m()$ (P is passive) $P = [f_1 f_1...]$
m found in:	passive frame f_i in a^i	passive frame f_i
m runs:	after scheduling in $Queue_i$	immediately (maybe serialized!)
m returns:	a promise yielding a value	a value
`this()`	f_i	f_i
`super()`	$root$	f_{i+1}
`athis()`	a^i	does not change
`asuper()`	a^{i+1}	does not change

Table 8.1: Message Passing and the Meaning of Context Functions

the corresponding passive object is searched for. This happens "in one shot", i.e., the lookup process itself does not generate any work in the queues. However, when found, the activation that corresponds to the message is enqueued in the right queue. In the figure, the method is found in the passive part of the second active object (note the solid black frame). The figure also shows the meaning of the runtime context functions `athis()`, `asuper()`, `this()` and `super()`. The first one refers to the active object that actually received the message. `asuper()` returns the active object that is the parent object of the object in which the code is running. `this()` is the passive object in which the method was found. `super()` is the passive object that is the parent object of `this()`. The philosophy behind this scheme is that `this()` always refers to the machine on which the code is running while `athis()` is the machine on which the receiver resides. `super()` is the parent object of the `this()` object that also resides on the local machine. `asuper()` has to be thought of as "the parent machine" of the machine on which the currently running method resides.

As explained, the philosophy of the model is that active objects should be used as the unit of distribution. Internally, active objects can manage very complex passive object hierarchies which are discouraged from being "published" outside the active object that "owns" them. The major reason is that passive objects are always copied when they cross network borders, *and*, that passive objects are not serialized by default which means that they can give rise to race conditions unless one explicitly serializes them. When executing synchronous messages sent to "local" passive objects, the meaning of `athis()` and `asuper()` do not change. Hence even though one can have several local messages being sent, `athis()` always refers to "the receiving machine" and `asuper()` to the parent machine of the machine on which the current code is running. The meaning for `athis()` and `asuper()` is summarized in table 8.1.

8.5.5 Wrap Up: ChitChat's Semantic Rules

Let us now summarize ChitChat's (active and passive) object structure, its message passing semantics and its synchronization rules. For deeper detail, we refer to appendix B.

The ChitChat Object Structure in a Nutshell

The internal structure of objects obeys the following laws:

1. There are two kinds of objects: active objects and passive objects. Active objects consist of a passive part, a message queue and a thread that runs infinitely.

2. Both active and passive objects can be constituents of an inheritance hierarchy. But the parent object of a passive object is always a passive object and the parent object of an active object is always an active object. Mixed hierarchies cannot exist.

3. References to passive objects never cross network boundaries. If a passive object has to cross a network boundary, a deep copy is passed on.

4. Parent references can cross network boundaries which will give rise to a network-based delegation. But by rule 3, only active objects can inherit from each other over a network.

5. Upon creation, the passive part of an active object is immediately serialized such that other references to that passive part can never entail intra-object concurrency if they would be running inside the object at the same time a message to the active one arrives.

Internal Object Scoping, Continued

Now we can finalize the internal object scoping rules started in section 8.4.2. As explained, there is a subtle difference between objects and frames. Only constants are looked up along the passive delegation chain as these can never be mutated and hence cannot give rise to race conditions. All the other names are solely looked up in the frame in which the currently running method resides. Because, as thoroughly explained in chapter 5, constants are public and mutable slots are local to an object, this means that there is a big difference between `this().m()`, `athis().m()`, `m()` and `.m()`. The messages sent to `this()` and `athis()` are late-bound sent to the public parts indicated by the pointers in figure 8.4. `m()` is only looked for in the frame in which the currently running method was found and `.m()` is looked for in the passive object that is the parent of the frame containing the currently running method.

Intra-Object Synchronization

Finally, synchronisation is summarized as follows:

1. Messages sent to passive objects are always sent synchronously. Passive objects *can* be manually serialized which will cause other senders to wait.

2. Messages sent to active objects are always sent asynchronously. Asynchronous objects process one message at a time, which renders them de facto serialized.

3. Asynchronous messages always immediately return with a promise. The promise will be fulfilled by the receiver.

4. Delegation along a chain of passive objects always proceeds synchronously.

5. Delegation along a chain of active objects proceeds asynchronously. The method is searched for "in one shot" but the activation is scheduled in the queue of object that beholds the method.

6. `super(...)` and `this(...)` are handled synchronously.

7. `asuper(...)` and `athis(...)` are handled asynchronously and return a promise.

8.6 Distribution

Having presented ChitChat's concurrency model, let us now turn our attention to distribution. Remember from section 7.2.1 that the ChitChat concurrency model was explicitly designed to map well onto its distribution semantics. Hence, it should not be much of a surprise that explaining the distribution semantics is pretty straightforward after having analyzed its concurrency model.

8.6.1 'First' Object Referencing: Channels

In section 7.4.1 we explained that any distributed programming language has to offer a way for objects to get an "initial" network reference to an object residing on a different machine. We have reviewed the two basic options, namely the centralized name server approach and the broadcasting approach. The hardware configurations we targeted in chapter 1 rule out centralized name servers. We have therefore provided ChitChat with a very simple service discovery mechanism that would fit a broadcasting implementation much better. The idea is that active objects can register *themselves* to a named channel by a call of the form `register("any Name")`. This will register the current active object to that channel which is meant to be broadcasted continuously. Any interested party can ask for the active objects that are currently registered on that channel by invoking `members("any Name")`. At any time this call yields a ChitChat table containing a network reference to the active objects that are currently registered on that channel. Objects can unregister from a channel by calling `unregister("any Name")`. Of course this will not change any references already established to that object from within other machines.

As Miller puts it in his work on the E programming language [MMF01], "connectivity begets connectivity". This is also the case in ChitChat. Once one has an initial reference to an object, one can send it messages and pass along other (active) objects that will henceforth be known by the receiver as well. The results returned by executing the associated method will be a newly known acquaintance to the sender.

8.6.2 The Power of Attributes, Revisited

In section 8.5 (and notably section 8.4.4) we have been explaining and using active and passive objects depending on our needs (of course obeying the rules outlined in section 8.5.5). Remember from section 5.8.5 that Pic%'s objects were created by successive applications of views, mixins and cloning methods which are installed by "prefixing" their name by "modifiers" such as `view`, `cloning` and `mixin`. In section 8.5 we already hinted at the existence of similar modifiers `aview` and `amixin` to create active objects. Since we know from section 8.5.5 that mixed object hierarchies are impossible, some hygiene is required in combining these different kinds of modifiers. For cloning methods, nothing changes with respect to Pic%. We have been as liberal as possible in the possible combinations of `view`'s, `mixin`'s, `aview`'s and `amixin`'s excluding only those cases that would violate the object structuring rules outlined in section 8.5.5. The most problematic case we had to exclude is the invocation of a "passive view method" on an active object. Although the other combinations was given a meaningful semantics, the resulting object structures sometimes yield intricate semantics. An important technical detail when actually working with views, mixins, active views and active mixins is the meaning of the scope functions `this()`, `athis()`, `super()` and `asuper()` *while* these (active) views and (active) mixins are running. We refer to appendix B for the details.

8.6.3 Distributed Object Creation

We already mentioned several times that ChitChat's concurrency model was deliberately designed with distribution in mind. For example, as explained in section 8.5.2 we have derived restrictions on the way object hierarchies can be built under the assumption that different active descendants of an active object can reside on different machines. This is accomplished by creating active objects. We postulate:

Active view methods are executed on the machine of the sender of the message that triggered the active view method

This is ChitChat's unique way to introduce distribution. Using the channel technique outlined above, a first (remote) object reference has to be obtained. From that point onward, messages can be sent to that remote object. If these messages are implemented by "ordinary" methods we get back references to objects that reside on the machine of the implementor[4]. But if that message happens to be implemented by an active view method, then an active object

[4]An exception to this rule occurs when the method returns passive objects. These are *always* copied across the network as explained in section 8.6.4.

is spawned on the machine of the sender which has the receiver of the method as active parent. From that point onwards, a new active object exists on the machine of the sender that has the receiver of the message as a *remote* active parent.

At this point it is instructive to refer back to the notion of first class methods and active closures explained in section 8.3.2. As was explained, an active closure emerged from selecting a first class method in an object without invoking it. It consists of the method code, the receiver in which it was searched for and the active object in which it was actually found. This active closure represents a function but because of the different types of methods ChitChat features, invoking the function can have different semantics. If the function emerged from selecting an ordinary method, invoking it will schedule the corresponding activation in the queue of the object that implements the function. If the function emerged from selecting an active view method, its invocation will result in a distributed object being created on the machine of invocation.

8.6.4 Arguments and Return Values

In ChitChat, the way arguments and return values are sent back and forth is practically completely the consequence of rules on distributing active and passive objects outlined in section 8.5.2. The semantics is summarized as follows:

1. Passive objects used as argument or return value are passed **by copy**. This is consistent with the philosophy of the model given in section 8.5.1: active objects should be the main source for distribution and passive objects should remain encapsulated inside active ones. The copy is a transitive closure of all passive objects for otherwise, the rule would be broken. Active objects residing in these passive ones will be used by reference.

2. Active objects are always passed **by reference** and are never copied. In chapter 9 we show that active objects can be moved around by ChitChat's mobility features. This enables remote methods to temporarily draw the arguments to the other side of the network.

Although deep copying passive objects on every network crossing might seem queer with respect to, we are not the first ones to come up with this semantics. Eiffel// [EAC98], albeit not distributedly, also uses this semantics when passing on passive objects between concurrent processes. Experiments with ChitChat have shown that the fact that we *do* allow passive objects to be shared by more than one active one was actually a design mistake because race conditions sneak in too easily. Adopting the Eiffel// is part of our short term future work.

8.6.5 An Experiment: The Chat

Let us now turn our attention to an experiment we conducted to illustrate ChitChat's distribution facilities. The idea is that of a centralized chat server that can dynamically spawn chat clients. The chat server is to be thought of as a distributed white board that is shared between the chatters that write on it. In contrast to conventional object-oriented implementations, our ChitChat implementation really reflects this sharing. In the code below we distinguish an active object constructor `chatServer` that, when called, creates a new chat server on the machine that launched that call. Notice in the very last line of the constructor that, upon being created, the chat immediately registers itself on a provided channel so that listeners can "import" a reference to the chat server by listening to the broadcast. Once the chat server is up and running, remote clients can send it the message `registerClient`. This is an active view method that will, when being sent from a remote client, spawn a remote active object that automatically delegates all messages to the server. There is only one such message implemented in the server. It is the `sendMsg`. When a user interface sends this message to a chat client, it gets delegated — over the network – and will get scheduled in the server's queue. Once the server is ready to process its next message, it distributes the message received to all its clients asynchronously. Notice that this is possible because the view method that spawns clients registers them in a table by immediately putting `athis()` into a centralized table which its accesses by a `asuper(...)` send. This table is consistent at any time because of extreme encapsulation: no one is able to spawn descendants except for the chat itself.

```
{ aview.chatServer(channel, maxClients) :: {
    clients[maxClients] : void;
    occupancy: 0;
    aview.registerClient(nam) :: {
       receiveMsg(from,msg) :: display(from,": ",msg,eoln);
       asuper(
         if(occupancy=maxClients,
            error("Sorry, channel is full"),
            clients[occupancy := occupancy+1] := athis()) ) };
    sendMsg(msg) :: {
       from: athis(nam);
       for(i:1, i <= occupancy, i:=i+1,
           clients[i].receiveMsg(from, msg));
       "sent"  };
    register(channel) };

progServer: chatServer("prog", 10);
aChatter: progServer.registerClient("Working on My PhD");
aChatter.sendMsg("Hello prog") }
```

At the site of a client, the following code is used. The required channel is read and it is sent the message `registerClient` which will create a chat client on the sender site. It can be sent a message `sendMsg` which will give rise to the message dissemination machinery described above.

```
 { progServer: members("prog")[1];
```

```
    aChatter: progServer.registerClient("Sleepless in Seattle");
    aChatter.sendMsg("Hello world") }
```

8.6.6 Another Example: Remote Remote Controls

One of the applications of this technique is what we call *Remote Remote Controls*. The idea is that of a PDA that sends an initial message to, e.g., a household device. The result of the message is a local object on the PDA that implements the complete control software for the device. For example, upon entering a meeting room, one might ask a video projector that is attached to the ceiling for a remote remote control such that it can be controlled from the PDA. Of course, when two such remote remote controls exist, they share the actual state of the device (e.g. the video projector can not be on and off at the same time, and it can only have one brightness level). The shared state sits with the active object on the video projector that is the remote parent of the remote controls that runs on the PDA.

8.7 ChitChat: Evaluation and Epilog

Based on the ChitChat design and the experiments presented in this chapter we can make a number of considerations about ChitChat and further situate it in the realm of distributed programming languages.

8.7.1 Networked Method Lookup Foolish?

A common reaction of most people that are first confronted with the ChitChat distribution model is "But, surely, you don't want method lookup to proceed over a network!?!". This is true for "ordinary" hierarchies such as graphical shapes or collection hierarchies. Therefore, these classic code reuse cases can still be conceived with "classical" passive local hierarchies.

Referring back to middleware solutions in which objects are remotely represented by proxies, these proxies are typically "empty" objects that have no logic in them but actually *delegate every message over the network*. Java RMI e.g. creates a special thread for every RMI-call just to set up the right machinery to get the message sent over the network. In this regard, our model is much more powerful. ChitChat's transparent network references are like proxies: they delegate *all* messages over the network. But it *is* possible to make a local version of methods by putting code in active views. Since these views remotely override methods, the ChitChat is more powerful in the sense that it can handle **smart proxies** that can reduce the network trafic by implementing some messages locally. Hence we conclude that the networked views on objects are simply a **generalisation of proxies**. A very nice side effect of this technique is that several "smart proxies" can refer to a "real object" and make sure the real object contains the one and only state of the object. Serialization of executing the proxy code inside the real object is guaranteed by the fact that calls to `asuper(...)` are serialized. Furthermore, when the real object performs self sends to `athis()`, they are intercepted by the proxy because of late binding of self in a prototype-based language.

Another argument to defend the mechanism is that, maybe, the time is ripe for this kind of machinery to enter programming languages. Only very recently, a distributed version of Self, dSelf [TK02] was proposed in which the authors defend delegation of method lookup to distributed parents. However, ChitChat is far more developed and much better understood than dSelf. The early publications on dSelf merely mention an extension of the Self method lookup over networks. They do not mention synchronisation, race conditions, deadlocks or any other phenomenon that manifests itself in the context of objects and distribution.

8.7.2 Distributed Scoping

We are not the first to come up with a rather exotic network-oriented language. The Obliq language, reviewed in section 7.5.3 features the notion of distributed lexical scope. No matter where objects reside on the network, their lexical scope is always guaranteed to be preserved even if it means that lexically scoped variables have to be looked up across network boundaries.

We argue that the ChitChat model is more powerful in two senses. On the one hand, the networked delegation scheme allows for network-oriented scope to be exploited, precisely as in Obliq. However, our model is more refined. By clever combinations of active views, it can be fine tuned which parts of the objects reside at what machines. On the second hand, as will be explained in chapter 9, our model allows active objects in an active object hierarchy to be moved across a network. This means that load balancing techniques can be used by children to drag their parent to the same machine that they reside on. This way the number of network references from a descendant to the parent can be reduced. This is impossible to achieve in Obliq because there is no way an object can get a reference to its lexical scope with the goal to drag it towards its own host.

8.7.3 Briot's Analysis

In one of the "classic concurrency papers", Briot and Yonezawa [BY87] argue that the delegation model of object-oriented programming is unreconcilable with concurrency. The basis of the argument is that, in the classic delegation-based scheme initially proposed by Liebermann [Lie86], reading and writing of (instance) variables happens by message sending. Based hereupon, the paper shows that expressions as simple as `x:=x+1` are a source of race conditions in the delegation paradigm. Our model shows that inheritance between objects and concurrency is a perfectly good marriage as long as the connection between an object and its parent is a more privileged one than the relation between an object and its "regular" acquaintances. In our model, objects are fully encapsulated entities in the inheritance chain. Scope function calls like `asuper(x:=x+1)` allow expressions occurring in descendants to be executed atomically in the context of the parent.

8.7.4 Situating the Model

Let us look at how ChitChat fits in with the landscape of existing models of distribution.

- As explained in chapter 3 we consider sharing to be an essential ingredient of distribution. The problem with existing approaches is that state sharing has to be manually implemented by "plumbing together" object references. Furthermore, keeping shared state consistent requires a lot of machinery to be implemented manually. In ChitChat this kind of structural sharing can be expressed very naturally by putting shared data in shared parents. Furthermore, the unwanted sharing relation imposed by classes as explained in chapter 3 does not manifest itself. Whereas ChitChat's (just as Pic%'s) immutable state and methods are conceptually shared by objects in order to improve reentrancy, this kind of sharing can never be detected by running code because it is immutable. So it is safe to duplicate it across a network. An exception to this is when immutable object slots refer to mutable structures (such as a ChitChat table or passive object). We consider this as a rough edge to ChitChat and plan to eliminate it by precluding passive structures to be shared between active ones. Race conditions simply sneak in too easily.

- Although the ChitChat model still has some rough edges and needs more polishing, it integrates many of the features of the languages discussed in section 7.5 without resorting to a straightforward language union:

  - ChitChat's concurrency model was explicitly based on **ABCL**. It adopts its promises system and its stateful active objects that forbid intra-object concurrency. ChitChat does not adopt ABCL's express messages because they breach the latter. Neither does it adopt the plethora of inter-object synchronization (past, future, now): every message sent asynchronously yields a promise. In contrast with ABCL, ChitChat promises are transparent placeholders for their value (yet to be computed). As such they are single values whereas ABCL promises actually represent queues that can be filled up by the receiver of a message and be claimed multiple times by the sender of that message. In contrast to ABCL, ChitChat features delegation with late binding of self between active objects. The resulting parent sharing structures are explicitly extolled to implement state sharing between concurrently running processes.
  - As explained in section 7.6 ChitChat and **Obliq** share the same vision on encapsulation. Both are prototype-based languages that, in the context of the amended Treaty of Orlando we proposed in section 4.6.5, align their templates model and their object model, but

keep their object model strictly separated from their self representation model. Obliq objects can be declared `protected` such that they allow some operators to be applied to themselves and no longer by external message sending clients. But in Obliq this also excludes delegation-based object sharing. Agora's view method and mixin method technique allows us to build (distributed) delegation-based inheritance structures in ChitChat without giving in on encapsulation.

- The **Argus** model consistes of guardians that are published on the network wich manage a non-published local state "behind the scenes". Several handlers of a guardian play the role of entry points for external clients. In their implementation, handlers can use transactions to make sure they do not cause race conditions on the state. In the ChitChat model this way of "resource management" is accomplished by putting the shared resource in a parent object and by defining the handlers in a semantically coherent way in descendant objects. All the objects can run concurrently and race condition free access to the shared state is accomplished with the scope functions. However, this atomic access is not a full-fledged transaction system as ChitChat's super sends cannot be rolled back. The exact relationship between the two models remains a topic for further investigation.
- As we will see in chapter 9 the major heritage from **Emerald** is the mobility features. But apart from that, Emerald's distribution model of classless objects to which messages can be sent transparently (even in the context of mobility) was one of the major sources of inspiration for our work.

8.7.5 (Mutual) Extreme Encapsulation & Security

Although it has been stated several times already, we do want to draw the reader's attention one more time to the fact that the ChitChat model allows for flexible object structures to be built over a network without having to compromise on security issues. Objects react to messages and *can* generate offsprings or clones *if* they were programmed to do so. Because of Pic%'s extreme encapsulation and reflection protection there really is no way to overrule this restriction.

Apart from precluding objects to tamper with other objects using encapsulation breaching operations such as inheritance, cloning and reflection (see section 3.6.3), ChitChat fully endorses Miller's view on security in programming languages [MMF01] discussed in section 3.6.2. Miller's view on security basically boils down to the slogan "connectivity begets connectivity", i.e., objects should only be able to access references to objects that were explicitly given to them. Miller allows only three ways by which an objects can get references to other objects:

- *Connectivity by Introduction:* Either some object introduces an object to a new object as the parameter of a message.

- *Connectivity by Parenthood:* Every object has a reference to the objects it creates.
- *Connectivity by Construction:* Every created object has references that were passed along to the construction code.

Miller argues that these three ways are sufficient to exclude various security problems that usually pop up when using "text-based resource referencing". For example, instead of letting a downloaded applet access a local database by allowing it to refer the database through some system call of the form `System.getDB("myDB")`, it is the task of the receiving machine to explicitly hand over that database to the applet when the applet arrives at the host, or, by sending a message to the applet. Resources that the applet has not gotten a reference to in this way should be hidden for the applet. ChitChat fully endorses this vision. The only way a freshly spawned active object can get a reference to another object is by parameter passing or by accessing state it shares with its parent and other descendants.

A related issue is what we might call *mutual extreme encapsulation* which is quite common a situation in (distributed) object-oriented programming. The problem is that two cooperating parties need each other's internal state in order to function correctly, without wanting to publish this state to the rest of the system. Unless one resorts to a C++-like friends mechanism, the only way to do this is by providing getters and setters as in the Java beans model. But this has the drawback that everyone can access this state. In ChitChat this is easily resolved by providing those two parties with a shared parent object that encapsulates the shared state. By using the scope functions correctly the two objects can read and write each others state without having to provide accessors that can be invoked by anyone else on the network. Again, the fact that only predefined (active) view methods can generate extensions in a controlled way (due to extreme encapsulation) is essential.

8.7.6 Rough Edges

Apart from the fact that our model lacks some essential features for writing realistic distributed applications in the context of open networks (e.g. there are no provisions for machinery that goes out of earshot, there are no provisions for partial failure and even a decent exception handling mechanism is lacking) the ChitChat model also has some shortcomings that need more work in future iterations over the language:

- As explained in section 8.4.5, the interleaving of super and self references can easily lead to deadlocks when these do not reside in tail call positions. This is definitely a topic that needs more work.
- Consistently copying passive objects across network boundaries might lead to subtle errors. Indeed, for a call like `eProduct.putInBasket(myBasket)` the `myBasket` object will be copied if it is a passive object. Additions to the

basket at the location of the `eProduct` will not affect the original basket. Solutions could vary from generating warnings every time a passive object crosses a network boundary, to automatically wrapping the object as an active object that is then passed by reference.

- Finally, ChitChat's object model that is essentially based on parallel active and passive hierarchies as depicted in figure 8.4 is not always easy to understand. We hope to simplify this model in future iterations over the language.

- Related to the two previous items is the fact that race conditions can occur easily whenever a passive structure such as a table or a non-serialized passive object is shared between two or more active ones. We plan to investigate in the very near future the effect of disallowing passive structures to enter or leave active object boundaries altogether. This will probably yield clearer concurrency semantics and aligns perfectly with ChitChat's distribution semantics. As we have already indicated this is already part of ChitChat's philosophy anyway: every active object is encouraged to posses and manage "its" passive data.

8.7.7 Implementation

In section 1.2.2 of the introduction we have explicitly taken the stance that language design and language implementation are to be kept separate as long as possible. But we also stated that a minimum of realism is needed in order not to step into the trap of inherently inefficient features. The ChitChat model was therefore given a prototype implementation in the context of Van Cutsem and Mostinckx's master's thesis [VM04]. The experiments shown in this chapter were conducted using this implementation.

8.8 Conclusion

This chapter introduced the ChitChat prototype-based model of concurrency and distribution. ChitChat features both active and passive objects. The active objects are to be seen as the units of distribution and are never physically moved upon using them for referencing and parameter passing. They implement ABCL style guardians that handle asynchronous messages one by one. They can be part of a hierarchy of active objects of which different parts can reside on different machines. The method lookup of messages sent to these objects proceeds along the networked inheritance hierarchy and the corresponding method activation is scheduled in the queue belong to the active object where the message was found. Active objects are to be thought of as managers of other active objects and also passive objects. Message passing to passive objects proceeds synchronously. Passive objects never give rise to network references and are always deep copied when they cross network boundaries. Passive objects are also part of hierarchies but these hierarchies reside on a single node in the network.

We have illustrated how active object hierarchies give rise to the notion of shared active parents of active descendants. These parents can reside on the same machines as one of the descendants or reside on a separate machine. The shared parent can contain state that is conceptually shared between the different active descendants. Cleverly designed scope functions allow chunks of code to be evaluated in the context of a parent or a child in serialized ways guaranteeing the absence of race conditions caused by the concurrently running parents. One of the most important changes with respect to Pic% is to restrict the scope of variables to the object frame in which they reside. A “manual” usage of the scoping functions guarantees atomicity.

We have illustrated this model of concurrency and distribution by describing the experiments we conducted to show its applicability for standard problems such as the dining philosophers and concurrently calculated fibonacci numbers. The chat experiment has shown how ChitChat enables the construction of distributed applications in which the sharing relationships between the distributed components are effectively visible from the way the program is structured. The resulting distribution structures have the hierarchical structure of a client-server topology but the power of peer to peer configurations. The latter will become apparent in chapter 9 will show how ChitChat's mobility features allow the distributed active objects to change host.

Chapter 9

Mobility in ChitChat: “Move” Considered Harmful

This chapter extends the ChitChat model of chapter 8 with features for strong mobility. The chapter reviews different kinds and causes of mobility and gives an overview of existing language proposals for code mobility. Subsequently, a suite of desired properties for mobile programming languages is distilled by conducting a number of gedankenexperiments and by considering the restrictions of the open networks we target. This allows us to formulate one of the main theses defended by this dissertation: a “*move*” instruction in mobile languages is a harmful feature. Based on this thesis, the design of ChitChat’s strong mobility is presented and a collection of mobile programming patterns is presented.

9.1 Introduction

As was noticed by Cardelli [Car95] and many others, the context outlined in chapter 1 in which users move around with respect to each other will very likely cause the software that runs on their PAN to move as well. The futuristic scenario presented in section 1.1.1 has given many examples of mobile software.

In chapter 8 we have presented a proposal for “structured distribution” in the context of open networks in which the object is no longer uniquely defined to be the unit of distribution. Thanks to distributed delegation, structural sharing patterns in distributed systems can be expressed very naturally by putting the structurally shared state of the system in a shared parent. An example of this was the chat application presented in section 8.6.5 in which the chat clients are local views on a shared parent object that represents the shared state of the distributed system. So far, this merely seems like a sophisticated way to program flexible client-server applications. However, in this chapter we show that the model is much more powerful if we extend the language further with mobility mechanisms.

As we will see, programming language design for mobility is pretty much unexplored. Existing language proposals usually boil down to the addition of a "move" operator to a distributed language. Such an operator takes an object and a node designation and simply moves the object to this new location. One of the basic arguments we will start out from is that this "move" operator that relocates an object, as found in existing languages, is a harmful feature. We argue this in section 9.6.4 by illustrating how it leads to incomprehensible object soups and how it is a source of security problems. Based on this analysis, mobility in ChitChat will be presented by means of a special kind of attributes (like view methods or cloning methods) that engenders mobility of the objects that implement it. These *move methods* (as opposed to move *operators*) are the heart of the mobility model.

In order for the discussion to be scientifically rooted, we start by briefly reviewing the state of the art in mobile systems. After introducing some basic notions of mobility in sections 9.2, 9.3 and 9.4 and after having a look at currently existing programming languages for mobile computing in section 9.5 we present the ChitChat mobility model based on move methods. Section 9.7 explains them and analyses how they interact with other language features. Section 9.8 presents some programming patterns supported by move methods. Section 9.9 evaluates the proposal.

9.2 Mobility: Definitions and Taxonomy

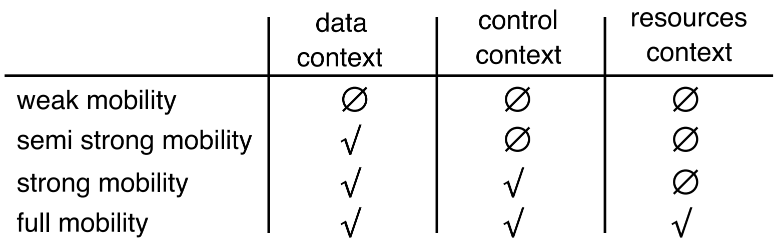
Before delving into the technicalities of middleware and languages that support mobility, we first describe what we mean by mobility and shed some light on the different kinds of mobility.

9.2.1 The Computational Context

In order to distinguish between the different kinds of mobility it is useful to consider the *computational context* of a running program. The computational context is a temporal snapshot of the knowledge a language processor has about the program it is executing. It typically consists of environments, a working memory, a runtime stack and so on. We will divide this computational context into a *data context*, a *control context* and a *resources context*. This distinction will help us to classify the different kinds of mobility in the following section.

The *data context* of a running program at a certain point in time consists of the variable bindings that are accesible by and allocated for the program at that moment in time. It is also called the state of a program.

The *control context* of the running program consists of the status of the computation. This usually takes the form of a reference to a point in the code (like a program counter or a "current expression") together with a description of those past states of the computation that are still relevant to determine the



	data context	control context	resources context
weak mobility	∅	∅	∅
semi strong mobility	√	∅	∅
strong mobility	√	√	∅
full mobility	√	√	√

Figure 9.1: Types of Mobility

future states of the computation. The latter typically takes the form of a runtime stack or a continuation.

The *resources context* of a program consist of those bindings of a program in memory that are not allocated for the program alone. This consist both of bindings in internal memory such as operating system resources (e.g. a window manager) as well as bindings that refer to external resources such as databases.

Using this terminology we can define the different types of mobility.

9.2.2 Kinds of Mobility

Depending on the source, terminology differs when it comes to characterizing the different flavors of mobility [FPV98, BN02, Tho97, CGPV96]. In this dissertation we distinguish between four types of mobility. We have summarized them in figure 9.1.

- **Weak Mobility** means that the ability is offered to allow "dead code" to travel over the network without any context information whatsoever. This can take several forms. A sending machine can send code to a receiving device (called "remote evaluation") or a receiving device might undertake the initiative and ask code to be downloaded from a sending device (called "code on demand"). In both cases, the mobility shares the characteristic that code is transmitted without any context information whatsoever. The code arrives at the destination machine and then starts running as if it never ran before.

- **Semi-Strong Mobility** is the form of mobility implemented by most middleware solutions. Semi-strong mobility allows a running process to move from one device to another one but requires the process itself to make the necessary provisions for halting and resuming the computation right before and right after the move. This means that the mobility does take the data context into account but does not consider the control context. Hence programs written using semi-strong mobility technology have to "manually copy" their control context into a data context and, upon arrival at the new site, make sure they restore the control context based on the transmitted data context.

- **Strong Mobility** occurs when a computational system has the ability to move from one device to another one taking into account both the data context and the control context. Strong mobility allows a running process to hop from one machine to another without manually halting the computation it is performing.
- **Full Mobility** arises when a strongly mobile program moves along with the complete closure of its data context including all resources that it will ever need during its execution. This means that not only the data context and the control context are taken into account, but the resources context as well.

Most people are aware of weak mobility because of its tremendous popularity in the incarnation of Java applets. With Java technology, a web browser can download a chunk of stateless code (i.e., an applet) and run it on the client that initiated the download. Apart from some static bindings like constants, the applet does not contain any state from previous runs. Full mobility and semi-strong mobility are not very well-known and in sections 9.2.3 and 9.2.4, we will make a case against them, especially in our research context. Strong mobility is not very well-known as a technology but most people associate it with *autonomous agents* that roam networks to accomplish a certain task on behalf of their owner. However, in section 9.3 we explain that strong mobility does not necessarily have to be associated with autonomous agents and that much more mundane applications of the technique are more likely to set in.

9.2.3 A Case Against Semi-Strong Mobility

Although semi-strong mobility has the advantage of being technically simple to implement, its practical applicability is problematic. Since semi-strong mobility is capable of moving programs with data context but without control context, programs have to halt their execution manually before they can be moved. This usually happens by notifying the runtime system that a move is required and consequently hand it over control of execution. The mobility machinery then serializes the "dead object" together with its code and moves it to another machine. The receiving device reconstructs the object and notifies it that the arrival has completed. It is the responsibility of the mobile program to restore the state and the computation it was performing before the move happened.

Proponents of semi-strong mobility usually revere the simplicity of the paradigm. The catch however is that the designer of the mobile software has to completely change the way he structures his code. Indeed, semi-strong mobility requires the algorithm to encode its control context explicitly into its data context such that it "remembers" what it was doing after being reactivated. This means that algorithms have to be explicitly divided along the mobility lines: they have to encode explicitly what "their future" will be after a move. This is known as continuation passing style programming (CPS) and it is generally accepted to be very hard to master fully by programmers. This argument is exactly the same as the one we held in section 7.3.5 when we discussed the actor model of concurrent programming. Both cases have a lot in common: at first sight they have the advantage of being simple. However, practicability is limited because they require CPS. CPS is usually considered hard as it requires one to turn one's algorithms "inside out".

9.2.4 A Case Against Full Mobility

As explained, full mobility means that a mobile process moves to another machine along with all the resources it needs. This means that no resources are ever rebound. The different forms of resource rebinding are thoroughly discussed in [FPV98]. They vary from actually moving them, making a copy, or just create a reference to them from within the new machine. Depending on the type of resource this has different repercussions. For example, for an integer number the network reference solution is unacceptable while moving or copying entire databases will be out of the question as well.

Full mobility was originally "invented" in the context of process migration for load balancing purposes. The idea is that a runtime system can — based on some parameters — decide to move a running process to another machine because that is beneficial to the overall computational performance or to the amount of network traffic generated.

Note that full mobility with *no* rebinding at all is probably not very useful in our context of open networks. It would mean that a device forces a computation to move over to another device to proceed in its new location without *any* interaction with computations or resources on the new host. Indeed, any form of interaction would require the computation to get an "initial reference" on the new host, which at least requires *some* rebinding. It appears that full mobility is only useful to allow "parasite computations" to consume computational power on another device than the one it started. In all other cases, *some* resource rebinding is necessary.

9.2.5 Conclusion

We conclude this section by repeating that there are four types of mobility: weak mobility, semi-strong mobility, strong mobility and full mobility and that we only deem the first and the third interesting and opportune. Semi-strong mobility is extremely hard to use by programmers because it induces CPS and full mobility is probably uninteresting in our research context because it (by definition) has no interactions with its environment.

9.3 Why Strong Mobility

Having "ruled out" semi-strong mobility and full mobility in the previous section, this leaves us with weak mobility and strong mobility.

The ChitChat model as explained in chapter 8 already has a built-in conception of weak mobility. By sending a message from one device to another device, code might be downloaded to construct an active view on an object residing on the sending device. This was illustrated in section 8.6.5 where chat clients where downloaded to the client machines by sending a message to the chat server. As we have explained in chapter 8, this model of distribution can be referred to as the *connected applets* model because it envisions the same kind of weak mobility as promoted by Java's applet technology. Note that the Java and JavaScript technology movements have made weak mobility completely accepted: many websites have applets or JavaScript programs embedded in them and even more sites contain embedded JavaScript code. We therefore consider it no longer necessary to scientifically motivate the need for weak mobility and the need for programming models and languages that technically enhance the ease with which weak mobile systems are constructed.

Strong mobility is a bit more controversial however and many computer scientists are still not convinced that it is a feature that one wants in a programming language. This is probably partially caused by the hype around e-commerce that has burnt terms like "agent" and "mobility" in the late nineties. This is why we will further motivate the need for strong mobility in this section. We identify three sources for strong mobility.

9.3.1 Intentional Strong Mobility

The first source of mobility is probably the best known one. But is is also the most controversial source of mobility and the one that gave rise to strong mobility not being generally accepted as "really needed". By "intentional" mobility, we mean that the mobility is a consequence of the fact that the program explicitly states that an object has to move around for some reason. In other words, intentional mobility means that the mobility aspect is part of the goals of the designer of the code. This kind of mobility can have several sources:

- **resource optimization**. This is the classic case for strong mobility. The point is that at some situation in the execution of a program, two or more cooperating objects are generating too much network traffic and the total traffic could be reduced by moving one or more of them. This is the classical load balancing argument for strong mobility. Notice that this does not necessarily have to lead to full mobility as explained above. It is perfectly possible and even likely that a strongly mobile object rebinds some of its resources upon arrival at its new destination.
- **software engineering**. Although it is harder to characterise, another source for strong mobility might be caused by design considerations. Designers of mobile applications might consider some objects "to logically

belong together" even though there are no quantitatively measurable reasons for allocating them to a certain machine. This might mean that these objects have to "move along" with another object even if they are currently "active" because they are on a runtime stack e.g.

- **Identity Preservation** In section 7.4 we already made a distinction between sending arguments of a remote method invocation by copy or by visit. The reason for making a difference between both transfer modes is all about preservation of identity. Whenever we have an object and we send it to another machine without making a copy, it means that we really want that object to move in such a way that all references to the object at the sending machine shall henceforth also refer to the moved object. At all times only one object is "the real" one. An immediate application of this is the iTicket in the scenario outlined in section 1.1.1. Other examples are Miller's purses containing digital money [MMF01].

Apart from these "contestable" reasons for strong mobility there are two more mundane sources for strong mobility. We discuss them in the following sections.

9.3.2 Distribution-Caused Mobility

A second cause of strong mobility is much more mundane and therefore much less controversal. As explained in chapter 7, concurrency models based on active objects are much more suitable to program open distributed systems than thread-based concurrency. We also saw in chapter 8 that one of the consequences thereof is that one naturally ends up in the situation that active objects are passed as actual arguments of a remote method invocation. In the case that one decides to actually pass the active object, we have a case of strong mobility because we are passing around a running object whose queue might be filled with requests. This clearly shows that, in a language supporting both remote method invocation and active objects, it is conceptually very easy to end up with the need for mobility. To the best of our knowledge, this was one of the main reasons for supporting mobility in Emerald [HRB+91]. As we will see in section 9.5.3 its mobility operators are most naturally used when sending along active objects as arguments to remote method invocations.

In section 9.2.1 we made a distinction between a control context, a data context and a resources context. This allows us to detect a hidden form of strong mobility that is not often discerned in the less technical literature. We have depicted it in figure 9.2. Suppose we have an object that processes a message by running a certain method m. Suppose that the object itself sends a message and passes itself as an argument. The method m' that receives the message receives an object that appears to be "dead". Suppose that m' decides to move this object to another machine. Upon returning from the move, m' gets control back. Now m' returns control back to the object by falling back into m. However, the object has changed location. This means that m has to

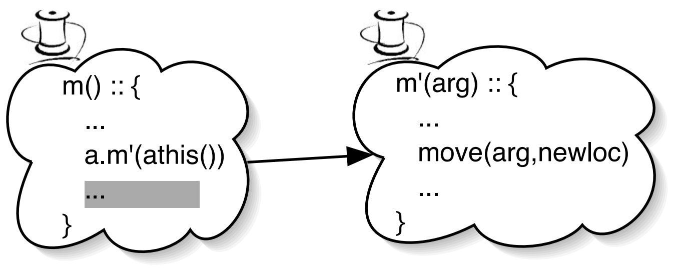


Figure 9.2: Unexpected Sources of Strong Mobility

continue the execution (i.e., the grey code in figure 9.2) at the new location which requires us to move the control state of an object that appeared to be "dead" because it was not "running" at the actual moment of the move. Again, this is a form of strong mobility that easily pops up in contexts that are *not* necessarily associated with agents that autonomously roam networks. It is a very mundane consequence of sophisticated distribution facilities.

9.3.3 Mobility from Mobile Computing

A third source of strong mobility is particularly relevant in our research context. It was already coined by Cardelli [Car95] and Fuggetta [FPV98] who make a distinction between *mobile computations* and *mobile computing.* The former is about software entities that roam networks. The latter is about people that carry along (miniaturised) computers and themselves move around. This is another way to designate the research context set out in chapter 1. Mobile computing easily generates the need for mobile computations because of two requirements Fuggetta calls *autonomy* and *disconnected operation.* Consider for example an integrated PDA/mobile-phone that is being used during a meeting as a text processor by one of the participants to take notes. As his phone rings he wants to leave the meeting temporarily to pick up the phone outside. He therefore wants to beam the text document together with the running notepad to one of his colleagues so that the meeting can continue while he is on the phone outside. The is an example to illustrate the fact that hardware mobility has a high probability of generating software mobility in the near future.

9.4 The Applicative Approach

In chapter 7 we saw that there are three approaches to add concurrency and distribution to object-orientation: the applicative approach (i.e., middleware), the reflective approach and the integrative approach (i.e., languages). In the same vein, these three approaches have emerged in the field of mobile code, and it is the applicative approach (very often in combination with the reflective approach) that seems to be the most popular again. Just as is the case with concurrency and distribution, there seems to be a tension between the applicative approach and the reflective approach. On the applicative "extreme", one starts from a regular object-oriented programming language and one constructs libraries to support mobility. Application development is then completely steered by the correct usage of these libraries because the application logic has to be moulded in order to make it fit the logic and constraints of the libraries. On the other end of the spectrum we find a pure reflective approach where one has the utopic situation of a sequential application that is changed into a mobile one by making the appropriate changes — at the meta level — to the way the object-oriented language internally works. Just as for concurrency and distribution, most middleware is situated somewhere in between these two extremes.

Especially with the advent of Java, middleware for mobility is attracting a lot of attention. It seems that Java's "compile once, run everywhere" slogan has launched the idea that code can travel over a network for good, especially because it has been combined successfully with the notion of dynamic class loading.

Unfortunately, Java is not very well suited for mobility as can be noticed from the tremendous complexity one has to deal with to accomplish a notion that is actually conceptually extremely simple — that of moving a running object from one machine to another one. In all the Java approaches we describe below, either the underlying middleware is extremely complex, or the middleware is kept simple but then the complexity is put on the shoulders of its users. That is probably why most mobile code today merely consists of simple Java-based applets. For all other forms of mobility, middleware researchers have been (and are) struggling with Java-based technology. All these attempts can actually be summarized as finding clever ways to circumvent the fact that Java's reification of running programs, i.e., the class `java.lang.Thread`, is not `Serializable` and can thus not be reified to send it over a connection.

The best example of these attempts is IBM's Aglets agent system [LO98]. This is a set of libraries with which users can develop mobile Java software entities, called "aglets". Aglets is an example of purely applicative middleware in the sense that it does not apply reflective technology to render an existing program mobile. Programmers are required to step into the Aglets logic and structure their applications as such. Part of this logic is that every aglet must be declared as a subclass of a class `Aglet` that is predefined by the middleware. Hence, most of the problems with middleware in the context of distribution also apply here. For example, the way two aglets have to communicate with each other consists of making instances of the class `Message`, manually providing it with the arguments (using a method `setArg`) and then explicitly sending the message from one aglet to another one. One of the most important drawbacks of technology such as Aglets is that it is a case of semi-strong mobility as discussed in section 9.2.2 which means that the Aglets library is structured in an event-driven way. For example, aglets have to implement the method `onArrival` which is invoked by the middleware after an aglet has moved. As explained in section 9.2.3, this requires one to write mobile algorithms in CPS because one manually has to encode and decode the control context into and out of the data context right before and right after moving an aglet.

Other middleware approaches based on Java are less extreme in their application of libraries and allow for a more transparent mobility which requires less effort from the application programmer. These approaches struggle much more explicitly with the fact that objects of the class `java.lang.Thread` are not serializable and can thus not easily be transmitted over a wire. There are two "solutions" to this problem which both seem to "reject" each other:

- The first solution consists of changing the Java Virtual Machine such that it implements the provisions necessary to implement a `go` native method on the `Thread` class. An example of this approach is JavaThreads [BHP03], D'Agents [GCK$^+$02], Sumatra [ARS97], Merpati [Sue00] and Ara [PS97]. The approach of adapting the JVM is criticized by the other camp because of compatibility reasons. The strong mobile code that relies on the extra provisions will not run on "standard" JVM's as dictated by Sun. One of the difficulties in this approach appears to be the fact that the extra serialization machinery seems to be the reason for extra inefficiencies and that non-mobile threads have to pay for this as well.

- The second approach can be classified as the transformational camp which consists of transforming (byte) code such that methods contain the appropriate instructions to copy their computational context into the data context. Examples of the transformational approach are WASP [FM99], JavaGo [SMY99] and Correlate [TRV$^+$00b]. The biggest drawback of the transformational approach appears to be the fact that it is not easy to guarantee the preservation of the semantics of programs by the transformation. Just as was the case with distribution (see section 7.4.1) code transformation is hard to combine with language features such as computational reflection.

In summary we can state that the middleware approach to mobility is — today — completely monopolized by Java-based solutions. Since contemporary Java implementations do not provide the underlying machinery needed to implement strong mobility, the class `java.lang.Thread` is not `Serializable`. Solutions vary from avoiding this class altogether (as in Aglets) to the development of sophisticated technical tricks to bypass the problem thereby trying to stick to Sun's definition of Java and JVM as much as possible.

9.5 OOP Languages for Strong Mobility

After having shed some light on the current status of middleware technology in the field of mobility, let us now (in parallel with the way we have reviewed concurrency and distribution in chapter 7) have a look at the integrative approach, i.e., how strong mobility is incorporated in todays' object-oriented programming languages. In contrast to the huge amounts of research effort spent on middleware approaches, the status of this field is pretty deplorable with a few notable exceptions such as TeleScript, Emerald, Obliq and Borg.

9.5.1 TeleScript

TeleScript is a proprietary language of General Magic which renders it quite inaccessible. To the best of our knowledge, there are no freely available implementations and the only paper describing the language is [Whi96]. The current status of the language is unclear.

TeleScript is a class-based language in which classes can be declared `abstract` or `sealed`, TeleScript's terminology for what Java calls "final". TeleScript features mixins in the style described by Bracha [BC90]. Mixins are named abstract *sub*classes that can be applied to several classes in row. This is used by TeleScript to implement some of its security: TeleScript features four built-in mixins which can be applied to classes to "modify" them: `unmoved` renders objects of the modified class unmovable, `uncopied` turns a class into a class whose instances cannot be copied (which might be necessary to implement digital money e.g.), `copyright` turns a class into one that can only be instantiated by "copyright enforcer" objects and `protected` turns a class into one whose instances cannot be modified. Apart from these class-based security techniques, TeleScript is in fact a pretty standard class-based language.

TeleScript's provisions for strong mobility are completely covered by a predefined class hierarchy that consists of three classes: an abstract superclass `Process` with two useful concrete classes called `Place` and `Agent`. The idea is that places can host agents. An agent is a process that can be moved from one place to another. To this end, the class `Agent` implements a method `go` that takes an object of type `Ticket` as a parameter. A ticket is basically a set of references to places. The ticket determines the trip of the object upon invocation of `go`. A variant of `go` is `send` which can be used to move copies of the agent to different locations. This is reminiscent to the master-slave pattern discussed in section 9.8.2.

TeleScript has a capability-based security model. Every object of type `Agent` contains a `permit` field. Its content is an object containing the capabilities of the agent. Apart from some bookkeeping issues (such as its age), an agent's permit defines a set of predefined booleans such as `canCreate`, `canGo`, `canGrant` and `canDeny` that speficy whether the agent can create new processes, can move around, can raise and lower the permission level of other processes. The effective capabilities of a process are the computed intersection of its own permits, the local permits and extra permits which the currently running process might define with additional `restrict` instructions.

From our analysis in section 3.2, the fact that TeleScript is class-based implies that it is only practicable in networks with predetermined topologies.

9.5.2 Obliq

Obliq is arguably a mobile programming language. Although Obliq was explicitly written for *"... computations that roam networks..."* [Car95], Obliq has no explicit mobility operators.

In section 7.5.3 we have reviewed the four Obliq language operators that can be applied to objects: selection/invocation, cloning, aliasing and assignment. In Obliq mobility is actually accomplished by combining its cloning operator with its aliasing operator. At the heart of this technique is the property of the cloning operator which makes sure that clones made of remote objects are actually created locally. So it is more correct to speak about *remote cloning* than just cloning in Obliq. Combining this feature with aliasing enables mobility in Obliq as follows. A self-synchronized Obliq object that wants to move:

1. sends a first class procedure with itself in the scope to an execution engine residing on the receiving host. The first class procedure should return a clone of the object in its scope (i.e., the "self"). But since the first class procedure is sent across the network, the object is turned into a reference.

2. by taking a clone of that reference on the destination machine, the Obliq cloning operator makes sure the clone is constructed locally.

3. upon returning from executing that first class procedure remotely, the object that spawned the procedure has to redirect — with one atomic instruction that is syntactic sugar for several aliasing instructions — all its fields to refer to the remote clone.

Part of the trick here is that Obliq's objects are required to move themselves because no thread should be able to tinker with the object during the execution of these steps. It therefore all has to happen atomically. Obliq enables this by declaring the object as self-synchronized so that the method that causes the copying and redirecting cannot be interrupted.

9.5.3 Emerald

We already explained Emerald in section 7.5.1 in the context of distribution. On top of what was explained there, Emerald has a number of explicit language constructions that support object mobility. They can be divided into expressions, statements and annotations.

- The expression `locate exp` evaluates the expression and returns an object of type `node` which is the location of the object denoted by that expression.

- The concept of a node is used in the following mobility statements:

  – `move e1 to e2` is an instruction that causes the object denoted by `e1` to be moved to the node denoted by `e2`. If `e2` is not a node, the node of its value is taken. Notice that in Emerald, mobility as specified by programmers is always a mere hint to the implementation. The kernel might discard it anyway.

  - `fix e1 at e2` is an expression that will cause the object denoted by `e1` to be fixed at the location denoted by `e2`. After the fix, the object can no longer be moved.
  - `unfix e1` is a statement that is exactly the opposite of `fix`.
  - `refix e1 at e2` refixes a fixed object at a new node.

- Invocations of Emerald methods allow their actual arguments to be annotated with the keywords `move` or `visit`. In this way, the actual argument will be moved to the location of the object that receives the message. Upon returning from the method call, arguments annotated by `move` will stay at the node of the receiver. Those annotated by `visit` will come back to the node of the sender of the message.

Apart from these explicit mobility mechanisms, Emerald allows objects to be attached to other objects which will cause them to move together. To this end, Emerald features a “modifier” `attached` that can be used to annotate variable declarations in the same spirit that Java allows variable declarations to be annotated by `static`. The attachment relationship induced by such declarations is transitive but not symmetric.

9.5.4 Borg

Borg [VF01] is an early extension of the original Pico language explained in chapter 5 with mobile agents in the form of autonomous objects that can roam networks. In Borg, a lot of effort was spent on implementation technology, to wit efficient object serialization operators and efficient re-routing of object references and messages sent to moved objects. To this end, Borg included a sophisticated distribution layer with hierarchically organized routing of messages between virtual machines [VF01].

Borg features the creation of autonomous agents encapsulating a Pico dictionary and a runtime stack that could be subject to serialization in order to be sent across a network. Borg makes a complete transitive closure of the dictionary making up an agent, except for the root dictionary of the system. Upon arrival, the dictionary of the agent is restored and attached to the root of the new host, and the runtime stack is installed such that the agent continues to run as if nothing had happend in the first place.

From a programming language point of view, Borg’s innovative features were mainly focused on data communication between agents. It features pretty sophisticated handshaking mechanisms that allow for “bidirectional messages” to be sent between distributed agents. Both sides can specify several actual and formal parameters at the same time. A unification system is responsible for unifying the formals with the actuals on both sides and for making the necessary bindings.

From a mobility point of view, Borgs features are pretty low level as we will argue in section 9.6.4. Borg features an operator `agentmove` that takes two arguments: a location (which is a reference to a machine) and an agent. The effect is that the specified agent is moved to the designated location. Every pattern of mobility that is slightly more complicated than this has to be manually encoded by Pico code that explicitly manages locations and agents and that decides "manually" which agents should move whereto at what time.

9.5.5 Other Mobile Languages

We have only discussed four mobile programming languages in detail. The reasons for selecting them is the fact that they are object-oriented and that they support strong mobility.

Other languages such as Java, JavaScript, Limbo and O'Caml are object-oriented languages but merely feature weak mobility in the form of applets. Facile is neither object-oriented, nor strongly mobile and Safe Tcl does not support object-oriented data abstraction. For more details on them, we refer to [Tho97]

9.6 Issues in Mobile Language Design

Let us now consider the language design alternatives we can consider to design a mobility model for ChitChat. After considering some general alternatives in sections 9.6.1, 9.6.2 and 9.6.3, we finally present one of the main theses defended in this dissertation: section 9.6.4 argues that a "move" operator that bluntly moves objects from one location to another one is a harmful feature in a mobile programming language.

9.6.1 Push, Pull and Agent Technology

Most survey literature on mobility (such as [FPV98]) distinguishes between three fundamentally different approaches of making code mobile:

- The first option is sometimes called "code on demand", or "fetch technology", or **pull technology**. The basic idea is that a chunk of code is downloaded from one machine to another one in order to be executed on the latter. The initiator is the (code running on the) machine to which the mobile code will move. The classic example of pull technology is Java applets.

- The second way to enable code mobility is called "ship technology", or "remote evaluation", or **push technology**. In a push architecture, the initiator sends a chunk of code to a machine. The receiving machine will run the code and possibly send results back to the initiator. Intelligent query languages are often named as examples of push technology. The TeleScript language discussed above with its `go` and `send` methods is also an example of push technology. Emerald's actual argument annotations are examples of push technology as well.

- Finally, the third possibility for code to move from one machine to another one is autonomicity or **agent technology**. Examples of agent technology are the Borg system described above and also Obliq. In agent technology, a running process decides autonomously to move from one machine to another without initiation from the sender or the receiver.

A remarkable fact is that pull technology is usually associated with weak mobility because it is a running process that pulls a “dead object” to another machine. Furthermore, strong code mobility is very often associated with agent technology, i.e., the idea of autonomously running objects that hop from one machine to another.

9.6.2 Classes vs. Prototypes

Although the relationship between classes, prototypes, open networks an mobility was extensively discussed in chapter 3 we want to draw the reader’s attention back to the fact that — with the exception of TeleScript — all object-oriented languages discussed in section 9.5 are actually prototype-based. In chapter 3 we extensively discussed the non-viability of class-based systems in the context of open networks. Needless to say, the problems merely get worse when strong mobility is involved [DD02]. Objects might move around and end up on machines where their class already exists with different versions for the methods and with different values for the static variables. Emerald as a prototype-based language is not totally free of problems in our context of open defined networks either. The fact that objects need a unique name within the network can easily lead to problems when unforeseen devices enter the network that contain objects with a name that already exists. Furthermore, the fact that every object in Emerald has a type yields the same problems if objects with the same type name but different signatures enter the network. Maybe the typing issues could be solved by associating the identity of the “source machine” to every type behind the scenes. The naming problem seems inherent to Emerald though. As explained in chapter 7 another argument “against” Emerald in the context of mobility for open networks is the fact that it only supports synchronous message passing in a thread-based concurrency model.

9.6.3 Security Issues: Resource Rebinding

As was already explained in section 3.6, security is a very important topic in open networks and mobility. We have followed Thorn’s analysis in that security is needed at different levels of mobile code execution. We have concentrated on programming language level security and have argued that the prototype-based languages discussed in chapter 2 suffer from serious problems when it comes to security. A language that allows programmers to write secure programs is a language that allows one to bundle resources according to capabilities and pass these capabilities on to other parts of the program. Those parts can only access the resources for which they have received the capabilities and there is no way this can be circumvented. Abstractly spoken, a capability is nothing but a designator for an entity together with a description of what can be done with that entity. In an object-oriented programming language, a capability is typically modeled by a reference to an object or by a method that resides in an object.

Security in mobile systems on the level of programming languages is all about resource rebinding. Indeed, in the case of full mobility as described in section 9.2.2, there are no security problems possible because all the resources the mobile program might ever need are transported along with the program. It is when a mobile program gets access to local file systems, windowing systems, and so on, that security problems pop up. In order to avoid them, no global runtime authority should exist to which extra capabilities can be "asked" on the basis of a name — a meaningless string. It is of utmost importance that every operation that an object-oriented application might perform on a resource is conveyed by a capability, i.e., a reference to an object or a method. In our vision of connectivity begetting connectivity this means that the Granovetter operator as discussed in section 3.6.2 is the only way capabilities can be obtained. The painfully slow convergence of the Java sandbox model towards a secure environment has demonstrated the world how difficult it is to achieve true security without sticking to an exclusive use of the Gravovetter operator. In an architecture that *does* stick to it, sandboxing a mobile program "simply" means that one does not hand it over any capabilities upon arrival of its mobile code. Furthermore, using the Granovetter operator, gradations of sandboxing are possible by providing mobile code with a limited set of capabilities. Remember that his operator can technically take many forms varying from parameters being passed during a method invocation, via references being obtained at object creation time, to variables being shared and mutually accessed by different objects. The point is to use the scope rules of a programming language to obtain a reference to a resource.

9.6.4 "Move" Considered Harmful

When considering strongly mobile programming languages in general and the particular cases discussed in section 9.5 we notice that — with the exception of Emerald — these are actually ordinary programming languages, often with sophisticated distribution facilities, to which a `move` or `go` operator was orthogonally added. This operator bypasses the language's interaction mechanisms (such as message passing, function calling and process spawning) and bluntly moves a running process or object to a specified destination machine. It is one of the explicit theses defended by our research that adding such an operator is not sufficient to master the complexity of mobile systems, and that more higher order mobility features are required in order for mobility technology to be lifted from a purely academic trinket to engineering levels. In section 9.7 we present ChitChat's mobility model as a first proposal for such a feature.

In order to motivate ChitChat's mobility model thoroughly, we first present an analysis that rules out — by elimination — a number of language design options using logical argumentation as well as by conducting a number of gedankenexperiments. As we will see, all these arguments describe symptoms of the same fundamental problem: **move considered harmful**.

A Software Engineering Argument against "Move"

In many articles on mobility it is explicitly stated that mobile code paradigms are programming paradigms in which the concept of location is explicitly incorporated. These locations are often modeled as an additional data type or as instances of some special class such as `Place` in TeleScript [GM95]. The programming language then features a `move` or `go` operator which consumes two arguments — an object and a location — and which moves the object to the new location while taking care of automatic rebinding of local to remote references, automatically managing the runtime stack and so on. Although this is quite a technical achievement, it is too basic an operation to be useful for true mobile software engineering.

In order to motivate this further, consider a distributed program that consists of several objects running on different machines in a network. Suppose all of them run idiosyncratic code and suppose they somehow refer to each other and communicate by sending around messages. Now imagine the complexity that would arrise if all our current modern object-oriented programming techniques such as regular control flow instructions ("while", "for", ...), late bound message passing, double dispatch, meta programming and exception handling would be further complicated by a `move` instruction that explicitly moves objects to first class locations. We claim that with only a few lines of code that use one or more of these features, networked object-soups can be created the structure of which cannot be predicted or understood by human readers of the code. It is one of our explicit research assumptions that *we need abstraction mechanisms that enable human readers to "predict" the locus of objects.*

We would like to draw a parallel with the `goto` statement from early imperative programming languages. Surely, by moving from goto-programming towards structured programming, the power of imperative programming did not decrease. In the same way, we claim there is a need for **structured mobility**, i.e., language mechanisms that can help human readers of code determine the loci of objects by reading the code. Although structured programming does not allow one to express less powerful programs than goto-programming, it has rendered many programs that would have been unreadable with goto-technology easily readable. In order to further fortify our argument, consider the table in figure 9.3. It compares the current state of affairs in mobile programming languages with the evolution of `goto`. The table shows how current mobile programming languages in which `move` instructions use a label to denote the destination where objects have *to go*, correspond to imperative languages that allow labels in their `goto` instructions. Surely the latter was an improvement with respect to `goto` instructions that hardcode line numbers. However, most computer scientists will consider them as equally low level. The top left entry of the table shows the corresponding language feature for mobility. With a little

move	goto
move(obj,134.184.43.120)	goto 140
move(doc,RndSelect(t , p))	goto 140*x
move(doc, Printer)	goto computeSalaries

Figure 9.3: Move Considered Harmful

generalisation we can thus state that today's mobility abstractions are hardly above the IP-level.

Relative Locations

Most literature on mobility literally states that mobile programming is all about programming with explicit locations in mind [FPV98, Tho97]. We only partially agree with that. Surely, mobile programs are written with the explicit assumption in mind that parts of the program will reside on and move between different locations. However, this does *not* imply that mobile programming languages should explicitly model the concept of "a location". Apart from the fact that such locations easily give rise to the need to design a "move" operator — which brings us in the situation described in the previous section — we consider it a matter of bad language engineering to model locations as absolute locations in a logical or hardware network topology. As witnessed by Emerald's `attach` declaration, designers of mobile software are actually much more interested in **relative locations**. One wants objects to reside at or move to locations that host other objects. Notice that one has to interpret the words "object" in the previous sentence in a broad sense: a runtime system is an object, a database is an object, hardware is represented as objects and so on. In this sense, mobile software is not about moving objects to certain hardcoded locations. It is all about moving objects to locations of other objects. Another witness of this is the way Emerald programmers specify that parameters of message expressions have to be passed by reference, *by move* or *by visit*. Again, this is a manifestation of the fact that a programmer wishes to express that some objects belong together in some situations. Hence, mobile programming languages should focus much more on relative locations instead of absolute locations.

Respecting Granovetter vs. Mobility

An explicit "move" instruction in a prototype-based language is actually an operator that gets defined on the object model, in the sense of section 3.5. We have seen in section 3.6.3 that each and every one of those operators is actually a way to bypass the Granovetter operator which we declared to be our main yardstick to measure the "connectivity begets connectivity" principle. Especially in the context of open networks, this is really important. Since moving an object will be combined with at least *some* resource rebinding (we ruled out full mobility in section 9.2.4), an explicit "move" instruction means that one can bluntly move an object to a certain device without any approval of the device which can imply the device's resources being unauthorizedly bound to the object. Indeed, applying the Granovetter operator would mean that we *introduce* the moved object to the new device by means of a message. Clearly such a message passing way for engendering mobility (which we will discuss below) is not what the "move" operators discussed until know implement. Hence we consider a "move" operator as a breach of encapsulation because it is a way to cause connectivity without an application of a Granovetter operator. This can be dangerous for both the receiving host (one might move a hostile object) as well as for the object that is being moved (one might move an object to a hostile device).

Object Swarms

Another drawback of an explicit "move" instruction is that it renders the implementation of what we will call *object swarms* difficult. Object swarms are groups of object that have to move together for some reason. In the case of a "move" operator, one bluntly moves an object without the active cooperation of the object. If the object is required to "stick together" with other objects, it will manually have to ascertain the fact that they have drifted apart and take the necessary actions to move those objects as well.

A solution to this might be to enrich the programming language with predicates or attribute modifiers like Emerald's `attach` feature. This might do the job but can get pretty cumbersome to use when the swarms are not known upfront and are determined dynamically because such "togetherness proclamations" have to be established at the time of writing the code or at least before any "move" instruction is executed. Although not impossible to use, this is probably cumbersome and error-prone.

We conclude that an explicit "move" instruction that bluntly moves objects to a new device hampers programmers in expressing the fact that objects are to be grouped together on the same machine.

Vigna's and Thorn's Analyses

A particular variant of "move" is encountered in languages (such as Obliq) that adhere to the autonomous agent vision (as discussed in section 9.2.2) in which a running agent decides to move itself to another device. In [Vig04], Vigna presents a devastating analysis by listing no less than ten reasons why agent technology is bound to fail. Some of these reasons are organizational ("Agents are difficult to design" or "Agents lack a ubiquitous infrastructure"), managerial ("Mobile agents lack a shared ontology"), technological ("Mobile agents do not perform well") or cultural ("Mobile agents are difficult to test and debug") in nature and can thus be circumvented in the future by a change in culture and education, by the invention of abstract design methodologies and by technology getting mature. However, other reasons are quite technical and will probably indeed hamper agents — as we know them today – from being used in everyday programming. One of the most fundamental criticisms Vigna describes is that "Mobile agents are suspiciously similar to worms". His analysis is based on the fact that agents can autonomously move and trigger the transfer to a remote host where they resume execution. This is strongly related to our analysis of respecting Granovetter presented above. It can be brought back to the presence of a "move" operator in the mobile language that bluntly moves an agent (in casu itself) to a new location.

Thorn [Tho97] makes the same analysis when discussing the mobility features of TeleScript. He states that "Telescript agents have their own initiative to travel and are thus more powerful than Java applets, but in a sense, also more dangerous: they can be hard or impossible to control once launched.". From our discussion on TeleScript presented in section 9.5.1 we know that Thorn is talking about the `go` instruction here.

Vigna concludes that "We advocate that other (simpler) forms of mobility, like remote evaluation or code on demand, can provide support for the user's requirement in terms of service.... without raising many of the issues that have to be addressed when using mobile agents". In section 9.3 our analysis has pointed out that this is not correct: even "plain" distribution with active objects being passed around as parameters rapidly requires strong mobility. The point of Vigna's analysis is the autonomicity of agents due to the "move" instruction that bluntly move themselves to a new location.

Conclusion

We conclude by repeating that we have found no less than five good reasons to abandon an explicit "move" instruction from a mobile programming language. They all boil down to the fact that it is low level, does not allow programmers to structure the mobility aspect of their code and is a potential danger for security.

Formulated in the terminology of section 9.6.1 our rejection of "move" seems to rule out both the push model as well as the agent model of mobility. Indeed, the push model is a straightforward application of a "move" operator. Using the "move" operator, one pushes code to another machine without the active participation of the receiver. In the same way, agent technology — in which a running object declares to move itself to another device — can be regarded as a variant of a "move" operator in which the object being moved is "self". Although this eliminates some of the problems discussed above (i.e., at least the moved object itself is involved), most of them remain.

9.7 ChitChat Mobility

Based on the language design restrictions presented in the previous section, we can now present ChitChat's mobility model in the right context. Before presenting the model itself, let us first shed some light on the parameters in language design that we have at our disposal.

9.7.1 Design Space Parameters

In moving an object from one location to another there are actually three parties involved:

- The **sender.** This is the device or program that actually sends the object to another location. Although the initiative of the move might lie with another party it is the sender that performs the actual move. It can also "decide" how much to move of the object graph.
- The **receiver.** This is the device or process that receives the moved object. Following the philosophy of the Granovetter operator, the receiver will have to provide the moved object with a number of references such that the moved object can benefit from resource rebinding.
- The **moved object** itself. This not only involves a designated object. In cooperation with the sender, it implicitly also comprises "how much" that has to be moved, i.e., "how deep" the object graph has to be moved along. Furthermore, in cooperation with the receiver, the moved object will have to make sure some of its resources are rebound upon arrival at the location of the receiver.

Using this terminology, our analysis presented in the previous section forces us to postulate that

the receiver should play an active role in the move process

because otherwise the security problems concerning resource rebinding discussed in section 9.6.4 pop up. Likewise

the moved object should play an active role in the move process

because if this is not the case we are dealing with sheer push-technology as advocated by languages with a "move" instruction.

Needless to say, aside from these restrictions, we want our mobility features to endorse our extreme encapsulation principle presented in chapter 3. The following two sections present an initial suite of mobility language features that satisfy all these restrictions.

9.7.2 Move Methods

The idea of move methods is that they are — just like mixin methods, view methods, cloning methods, active view methods and active mixin methods — a new type of method that can be offered by the public interface of ChitChat active objects. Using the syntactic system explained in section 5.8.5, these methods

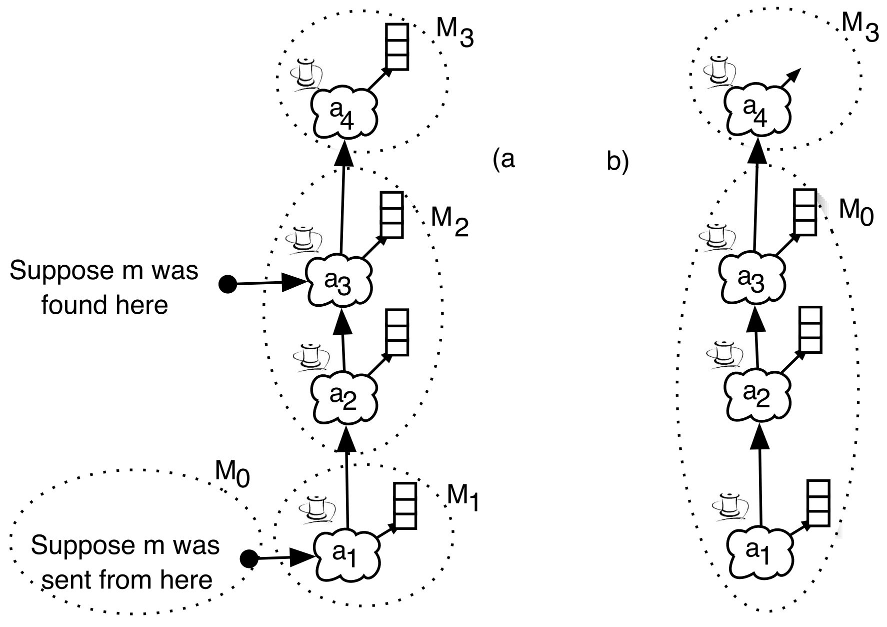


Figure 9.4: Move Methods in Action

are recognised by the fact that their name is prefixed by `move`[1]. Apart from the prefix `move`, programmers are free to choose the name of a move method. They can provide different move methods in an object each with their own functionality, they can freely decide which and how many parameters a move method requires and they can freely specify the body that implements the functionality of move methods in order to yield the effect they have in mind. Because ChitChat allows only active objects to be referenced over the network, move methods only make sense for active objects.

Invocation Semantics

The semantics of a move method is depicted in figure 9.4. Remember that the ChitChat distribution model is conceived around chains of active objects which can possibly reside on different machines and which are tied together by parent-of links. The chain is the result of successive applications of active view methods. Every active object can have internal references to both active and passive objects. The figure shows such a configuration on the left hand side where four machines M_0, M_1, M_2 and M_3 are involved. On these machines a distributed active object a_1 is created. a_1 is an object whose parent object a_2 resides on M_2. The parent of a_2 resides on M_2 as well. The parent of a_3 is a_4

[1]Remember that this is actually but the name of a higher order function called `move`, which the Pic% interpreter will use to "lift" the method to its new type before it is actually installed.

which resides on M_3. Now suppose a (move) message m is sent to a_1 from within machine M_0. Networked method lookup finds the method in object a_3 which resides on another machine than the machine of the sender. The semantics of a move method dictates that all active objects along the delegation chain, starting from the receiver of the message (in our case a_1 residing on M_1) up until the active object that contains the method (which is a_3 residing on M_3 in our case) will be moved to the machine of the sender of the message M_0. In case all these objects already reside on that machine, nothing is moved. The semantics further dictates that the body of the move method is run on the destination machine (i.e., after the move) but *only if* there actually was a move. Hence, the body of a move method will not be executed in case the sender of the message, the receiver and all active objects on the method lookup path until the implementor, reside on the same machine. Hence, the body of the move method is the action to be taken by the active object after it has moved to a new location. Of course, given our vision on distribution transparency as reflected by the constructs presented in chapter 8, moving objects is transparent from a technical point of view: the active object is moved and all other active objects that are registered to refer to the one that is moved are notified to change their references.

Marshalling Semantics

Referring back to the internal structure of objects as explained in chapter 8 we have to define what exactly is moved and what is not. The semantics of moving an active object consists of moving the active object (as an entity) itself, the passive part that internally defines the active object and the queue of the active object. The dictionaries that make up the passive part as explained in chapter 5 can contain "passive values" (such as Pico tables, passive objects and basic values such as methods, numbers and texts) as well as (references to) active objects. The semantics is that the passive values are moved along with the active object that is moving. Active objects are not moved along. Upon arrival at the new destination, they are replaced by network references to the original active objects. Hence the question of "what is moved along and what is not" is answered with the same semantics as the one that was used with argument passing and return value returning explained in chapter 8. Remember that this semantics was mainly driven by the fact that we did not want to create "passive network references", i.e., references to objects that would give rise to synchronized remote method invocations.

Just like ordinary methods, an activation of a move method is scheduled in the queue of the receiver of that move method. Invocation of a move method is an atomic action, so there can be no methods running between the start and the finish of the move. This guarantees atomicity.

The fact that moving is "shallow" results in references to active objects being changed into remote references. However, this is where the body of the move method comes into play. Upon arrival, after all the pointer plumbing has been managed by the interpreter, the body of the move method is executed on the new machine. By invoking move methods in its turn, the body of the move method can make sure that acquaintances are moved as well. And in the same way, by performing super sends (using the dot notation) the moving semantics can be propagated to the parent because — just as with cloning methods — the parent of an object in which a move method or cloning method is executing will be replaced by the result of a super send performed in that method. Notice that no manual pointer assignments are necessary. Simply sending a message will replace the network reference by the moved object both in the case of acquaintances as well as for the parent.

The remainder of the chapter further explains move methods by means of examples. For the precise technical details, we refer to appendix B.

Moving DAGs and Cycles

The fact that move methods are only executed if something actually moved is of vital importance when moving directed acyclic graphs (DAGs) and cyclic structures. The following code excerpt illustrates this:

```
aview.join(name)::{
  move.mv()::{
    display("join ", name," has arrived",eoln) }}

aview.pipe(name,o)::{
  move.mv()::{
    display("pipe ", name," has arrived",eoln);
    o.mv() } }

aview.fork(name,o1,o2)::{
  move.mv()::{
    display(name," has arrived",eoln);
    o1.mv();
    o2.mv()}}

e:join("ending")
p1:pipe("p1",e)
p2:pipe("p2",e)
f:fork("begin",p1,p2)
```

The code shows three types of objects that allow us to construct a small sewerage system: a join, a pipe and a fork. A DAG is constructed by forking two references to the join, via two different pipes. By sending the message `f.mv()` from within a different machine, the move mechanism gets in action. First the fork is moved to the site of the sender and then the body of the move method is executed on that site because there was actually a move. This causes the message `o1.mv()` to be sent which will cause the first pipe to move and launch the body of its move method as well. Likewise, `o2.mv()` is sent which has the same effect on the second pipe. Because both pipes were actually moved, their move method runs as well. This will cause the message to be forward to the join which will move the join in the same way. However, only one of these messages will actually give rise to a move (depending on the relative speed of the pipes). If the second one is processed the join will already reside on the new location which will cause the message to be discarded. In this way the move method in the join will run exactly once.

The fact that a move method is only executed after an actual move has happened is also crucial for moving cyclic object graphs. Indeed, suppose a cyclic graph of four objects is sent a message that is implemented by a move method. And suppose the move method propagates the message through the graph by invoking move methods as well. By the time the message is propagated to an object that has already been visited, the next to last node forming a loop will have moved. Therefore the corresponding move method will be run on the new location. If the body of that method sends a move method in its turn to the node in the chain that was already visited, this message will have no effect because the node was already moved to the new location when visiting it for the first time.

Parameter Passing

As explained in section 8.7, ChitChat endorses the view on security that references to objects should only be obtained through applications of the Granovetter operator. This also holds for mobile objects that arrive at a new location. We explicitly reject forms of dynamic scoping where objects can refer to context parameters at the new location that are not obtained through some form of parameter passing, or string-based mechanisms in which a reference to an object is obtained by "asking" a central authority (such as an operating system) for the reference on the basis of the name of the required object.

This is exactly what parameters in move methods are good for. By calling a move method with references to local resources, a move method can drag an object to the site of the sender and install references to the local resources at the right spots in the moved object. Since move method typically call other move methods, the parameters can be passed down the call chain easily.

As an example, suppose a lecturer enters a meeting room that has a video projector hanging from the ceiling. Upon request, the controller for the projector gets moved to the lecturer's PDA. Because different PDA's have different screen resolutions, dimensions, and color pallets, this information has to be handed over to the controller object upon arrival at the PDA. It can be compared to Apple's Gestalt-manager, a part of the operating system that allows applications to "query" for the gestalt of the system on which they are running. Here is how this could be achieved with a move method.

```
aview.Controller():: {
   ...
   move.toPDA(aScreenGestalt)::{
      ...
     }}
```

Using this configuration, it is completely up to the gestalt manager of the local PDA to decide which resources it provides to the controller it receives by sending it `toPDA`. There is no other way for the controller to access the local resources of the PDA on which it is running. This is in sharp contrast with the resource binding technique used in Java for example: upon arrival of an applet, the arguments are textually handed over to the applet straight from the HTML source. All security issues in mapping these strings onto meaningful object references is to be handled by an interaction between the virtual machine (implementing a sandbox model) and the applet logic itself.

As a final remark, the parameters of a move method can also be used to determine the semantics of the move algorithm. For example, flags could be passed around to steer the move algorithm in how deep the object graph has to be moved. This is a generalisation of the way objects are attached to each other in Emerald.

9.7.3 Design Space Parameters, a Second Time Around

On the basis of the analysis of section 9.6.4, in section 9.7.1 we deduced a number of restrictions on the design of mobility features. Now that we have presented the ChitChat mobility model, let us run through them again[2].

- We stated that the **sender** of the moved object should play an active role. In the move method mechanism this sender is the receiver of the message that corresponds to the move method. Basically, the sender of the object has three parameters at its disposal to determine whether something will be moved and if so, how much shall be moved. First, it can decide whether to implement such a request by a move method at all. In any other case there simply is no move. Second, the receiving object determines how much of that receiver that will be moved by implementing the move method in the appropriate active object in the delegation hierarchy. Third, using super sends in move methods, move methods can further implement fine tuned moving algorithms.

- We stated that the **receiver** of the moved object should play an active role. In our scheme this is the code that triggered the move method by sending it a message in the first place. In the object-oriented paradigm, one cannot get any closer to playing an active role than being the sender of messages.

- We stated that, at all times the capability-based object referencing should be preserved by only handing over object references through successive applications of the **Granovetter** operator. This is clearly the case. The

[2]Notice that in this discussion the terms "sender" and "receiver" are overloaded. On one level there are the sender and the receiver of mobile objects and code (i.e., machines). On another level there are the sender and receiver of messages between objects. It should be clear from the context which meaning is meant.

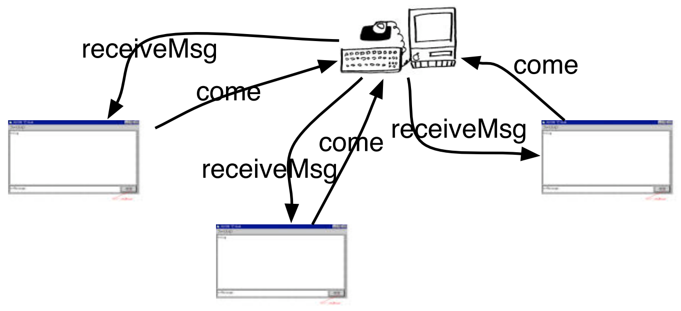


Figure 9.5: The Mobile Chat Server

only way an object can have access to local resources is by handing over a reference to those resources as arguments of the invoked move method.

- We stated that the **object that is subject to moving** should play an active role as well in order for it to determine how much of its graph should be moved and how its resource references will have to be bound at the new location. In our case, this active role is fulfilled by the algorithm implemented by the move method.

.

This analysis clearly shows that move methods are indeed a language feature that respects the requirements outlined in section 9.7.1. Move methods form a powerful general purpose mobility mechanism that does not suffer from the problems presented in section 9.6.4. Section 9.8 will further illustrate their use by deriving a number of higher order abstractions ("patterns") that use move methods, and appendix B gives a semi-formal semantics. Let us now turn our attention to an experiment that exemplifies the use of move methods.

9.7.4 Experiment: The Mobile Chat Server

This section reports on an experiment that we conducted in our prototype implementation of ChitChat. The experiment is actually a second version of the chat application presented in section 8.6.5. Remember that this experiment presented a distributed chat server on which chat clients represent local views. A chat client is spawned by sending a message to the chat server. Since the message is implemented by an active view method, the chat client is created on the machine of the sender of the message. Chat clients and the chat server have controlled access to each other's private variables by means of the `asuper` and `athis` scope functions which evaluate their argument expression atomically in the context of the parent object or the descendant object. The configuration is shown in figure 9.5.

The experiment presented below is a version of the distributed chat server that uses a move method `come` that can be invoked to pull the chat server to another machine. This move method is used by the chat clients to drag the chat server to their location from the moment they detect that they have become the biggest network traffic generator. The algorithm to determine this is quite simplistic: the chat server remembers at all times the number of messages sent by the biggest prater and every client remembers the number of messages it has generated. Once a message is sent to the chat server, the server updates this number by calling `athis(count:=count+1)`. Furthermore, the chat server contains `max`, the number of messages generated by the biggest prater. If `sendMsg` detects that this number is smaller than the number of messages generated by `athis`, then `athis` has to be dragged towards the machine of the client. This is accomplished by evaluating `asuper().come()` in the context of `athis`.

```
aview.chatServer(channel, maxClients) :: {
  clients[maxClients] : void;
  occupancy: 0;
  max: 0;

  move.come()::
    display("arrived", eoln);

  sendMsg(msg) :: {
    from: athis(nam);
    lcount: athis(count:=count+1);
    if (lcount > max, {
      athis(asuper().come());  max:=lcount });
    for(i:1, i <= occupancy, i:=i+1,
      clients[i].receiveMsg(from, msg));
    "message sent" };

  aview.registerClient(nam) :: {
    count: 0;

    receiveMsg(from,msg) ::
      display(from,": ",msg,eoln);

    asuper( if(occupancy=maxClients,
              error("Sorry, channel is full"),
              clients[occupancy:=occupancy+1]:=athis())) };

  register(channel) }
```

This code shows quite a number of pretty complicated interactions in only twenty two lines of code. We believe it somehow “shows” the expressivity of the ChitChat paradigm. Notice that this code is also totally secure. The only interaction it has with the machine on which it runs is an invocation of the `display` function. This could be avoided as well by handing over a “user interface” to the chat client as a parameter of the active view method that creates it.

9.7.5 A Variant: Visit Methods

In the introduction of the dissertation we heralded our analysis of "move" being considered harmful and we explained to give a first technical proposal to overcome the problems "move" poses. In order to stress the fact that move methods are but one possible solution, ChitChat also features a second mobility language feature called *visit* methods. Visit methods are methods which, when they are processed from an object's message queue, will move the receiver (more precisely all the active objects along the delegation chain from the receiver up to and including the object in which the visit method resides) to the location of the sender, run the method on that location and will subsequently move the object back to the location where it resided at the moment the visit method was invoked. Apart from the fact that all objects that were replaced return to their original location after executing the visit method, the semantics of move methods and visit methods is the same. In the following section we will illustrate a possible use of visit methods.

9.8 Mobile Programming Patterns

In the same spirit of the presentation of chapter 6 this section presents some programming patterns and idioms to illustrate the expressivity of the ChitChat model. Although the patterns presented in this section have been shown to run in our prototype-implementation of ChitChat, from an engineering point of view they might be less stable than the ones outlined in chapter 6 simply because ChitChat is more recent than Pic%. More "real life" experimentation with some patterns is needed to see how they behave in practice. Again, we discuss the implementation of a number of existing patterns occurring in the literature because they are particularly elegantly solved using ChitChat's features, as well as a number of techniques that are specific to (a combination of) ChitChat's features. Some patterns in the literature are explicitly rejected because of their inherent adherence to push technology or agent technology. An example is Aridor's forwarding pattern [AL98] that receives an agent and "manually" forwards it to a new destination machine.

9.8.1 Itinerary

An important mobility pattern frequently occurring in the literature [TOH99, AL98] is called the itinerary pattern. The idea of the itinerary pattern is that a mobile object is given an object that contains a list of destinations and routing among those destinations. By subsequent applications of "move" the object finishes the trip prescribed by the itinerary. An example of the itinerary pattern is an active document that has to be piloted through an administration. Different authorities have to put their "stamp" on the document before it can be forwarded to the next booth. The document might be encoded as a mobile object and the path of authorities it has to follow is encoded in the itinerary object.

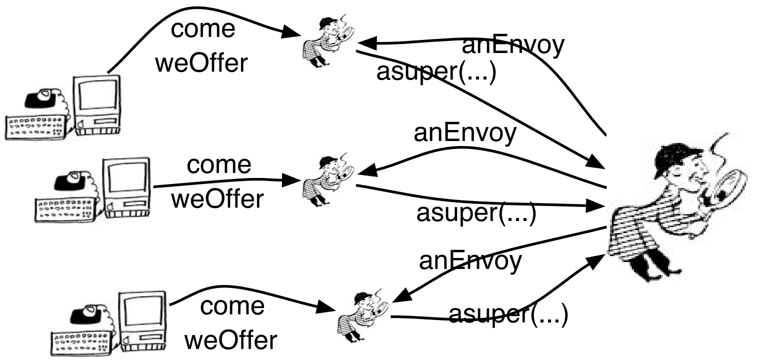


Figure 9.6: The Master-Slave Pattern

As we argued in section 9.7, ChitChat is a pull model and because the itinerary pattern seems inherently agent-oriented the only way to implement the pattern in ChitChat is by means of an abstraction that is closely related to the application of double dispatch in the famous visitor pattern. The idea is that an "agent" keeps a list of remote references. One by one, it sends `accept` to the references handing over a reference to `athis()` as an actual parameter. The remote reference then calls a move method on its argument which will pull the agent to the destination.

Following the arguments of section 9.7.1 we think it is not difficult to defend our implementation of the itinerary pattern in ChitChat. Both the moving agent as well as the destination objects have to "agree" (by implementing the right machinery) to implement the pattern. We feel this is an improvement for stability and security in the context of open networks because no agent is ever able to enter a device unless it is explicitly invited to. Nevertheless, our solution shows that — even in a pull architecture — the famous itinerary pattern is relatively easy to implement.

9.8.2 Master-Slave

Another pattern frequently referred to in the literature is the master-slave pattern [AL98]. The idea of the master-slave pattern is that there is an object that is charged with a certain task. The pattern prescribes that this object (the "master") can spawn a number of mobile envoys which can — in parallel — accomplish the task partially at another location.

The example we present here was also the one used in [DD03a] and [DCD03]. The idea of the example is the conception of a negotiation "agent" that is to roam a collection of remote hotels in order to find the best price. It is structurally depicted in figure 9.6. For the sake of the example, hotels are active objects that are registered on the `WestBestern` channel. Every hotel has a name and a price. It implements a `visit` method[3] that accepts a negotiation agent. It pulls the agent towards the hotel and starts a negotiation. This implementation looks as follows:

```
{  aview.Hotel(name,price)::{
      visit(agent)::{
        agent.come();
        agent.weOffer(name,price)  };
   register("WestBestern")};

   h1:Hotel("Sleep E",50);
   h2:Hotel("Sleepless In Seat-L",60) }
```

The following code shows how hotels are sent an agent. A hotel search agent is a centralized active object that has a `bestHotel` and a `bestPrice` state. Every slave is an `envoy` which is a view on the master. The slave implements a move method `come` such that all hotels can pull the slave towards their machine. Nevertheless, the slaves are all connected implicitly by means of the shared parent. Once the hotel starts a negotiation with the slave, the slave will (atomically) change the parent state it shares with the other states in case this hotel is cheaper. The code below shows this. All hotels of the channel `WestBestern` are referred to and sent an envoy one by one.

```
aview.hotelSearchAgent():: {
  bestHotel:void;
  bestPrice:10000;
  result()::display("Best Offer at",bestHotel,bestPrice);
  aview.anEnvoy()::{
    move.come()::void;
    weOffer(name,price)::{
      asuper(if(bestPrice>athis(price),
               { bestHotel:= athis(name);
                 bestPrice:= athis(price)}))} }}

{ hotelSearcher:hotelSearchAgent();
  hotels:members("WestBestern");
  for(i:1,i<=size(hotels),i:=i+1,
    hotels[i].visit(hotelSearcher.anEnvoy())) }
```

The classical implementation of the pattern is quite clumsy [AL98]. For example, the master has to create the agent, manually dispatch it to a destination, wait until all agents have accomplished their mission and then call a method `getResult` on every slave. Merging the results has to be done while or after collecting the results from the slaves this way.

[3]Not to be confused with visit-methods as a language feature.

9.8.3 Fixing Objects (a.k.a. "Stationary")

Most mobile programming languages discussed in section 9.5 have a way to fix an object at a certain host such that it can no longer be moved. In TeleScript this has to be declared statically by applying the `unmoved` mixin to the class of the object. In Obliq it is accomplished by simply not implementing moving machinery that remotely copies an object. Emerald probably has the most powerful scheme because of its `fix...at...`, `unfix` and `refix` operators that can be applied to objects. In ChitChat objects can be fixed by applying a "destructive" mixin to them after they have been moved. The mixin overrides the move methods with ordinary methods that do nothing at all. This implementation of the stationary pattern is shown in the following code excerpt.

```
aview.moveable()::{
  move.comeHere()::{true};
  amixin.fixIt()::{
    comeHere()::{false} } }
```

The object constructor `moveable` contains a move method `comeHere` that can be used to move the object around. However, sending `fixIt` to the object "inserts" an ordinary method that will override the move method. Because this is a mixin and not a view, all references to the object will have changed such that it can never be moved again. Since ChitChat currently does not allow objects to get rid of some of its constituents, this operation cannot be undone.

9.8.4 Local Algorithm Execution

Having presented an implementation of existing patterns in ChitChat, let us now turn our attention to a number of programming techniques that are enabled because of specific combinations of ChitChat's features. The first technique combines visit methods with Pico's special parameter passing semantics. It allows one to temporarily pull an object to a machine to execute an algorithm locally.

Imagine an administrative system of a governmental ministry that manages files that are stored in a centralized database which represents the archive of the ministry. Now and then a department of the ministry needs a file to process it. After the department has used the file, it is sent back to the central database so that other departments can use it. Suppose that files have a single identity and that only one department can work with a file at a time.

Using our abstractions, the database could be represented as a collection of active objects that each represent a file. A file has a name and a table of references representing the pages of the file. The mobility logic described above can be easily encoded as a visit method that is combined with Pico's call-by-expression:

```
aview.DBFile(fileName,fileEntries)::{
  visit.runIt(algorithm(name,entries))::
```

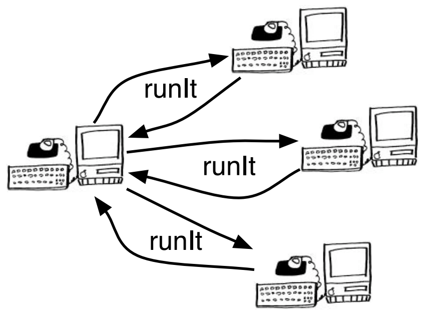


Figure 9.7: Star-Shaped Pattern

```
    algorithm(fileName, fileEntries)
  }
```

The above expression shows how a file `f` can be sent `f.runIt(...)` handing it an algorithm depending on `name` and `entries`. The file object will be transfered to the machine of the sender of `runIt` and then the message will be processed locally (thereby invoking the `algorithm`), after which the file object goes back to wherever it came from (i.e., the database in our case). Whether the entries referred to by the file will move along with the file depends on whether they are active or passive objects. As explained in section 9.7.2, passive objects always move along with the object that contains them. Active objects are replaced by remote references.

Notice that because ChitChat does not feature intra-object concurrency, the file will only be at one machine at a time and only one call to `runIt` at the time will be processed. The pattern discussed here resembles the star-shaped pattern in [TOH99] which is depicted in figure 9.7. The difference is that ChitChat's approach is much more flexible: because of parameter passing, the algorithm that will be processed locally does not have to be implemented as part of the agent, but can instead be dynamically "injected" into the agent by the local host using Pico's special parameter passing technique discussed in section 5.5.3. Because of ChitChat's extreme encapsulation, the agent is in full control about how and when it will execute the algorithm passed as a parameter and which local state variables it will pass on to it.

9.8.5 Swarms

In this section we show the power of ChitChat's features by giving examples of how to conceive object swarms. We call *a swarm* a group of objects that move together. As explained in section 9.7.2, "hardcoded" swarms are extremely naturally programmed with move methods: it suffices that the body of a move method sends "move messages" to its acquaintances in order for the acquaintances to follow wherever the object executing the move method is moving to.

There are nevertheless two interesting patterns that implement swarms by using ChitChat's language features in non-trivial ways. One of them is based on aggregation. Another one is based on delegation.

An Aggregated Swarm

In section 6.9 we explained how first class methods can be used as a technique to support the construction of higher order connectors. The fact that methods are first class allows one to pass them on to a third party that can henceforth call them without knowing their name upfront. This allows us to write higher order composition technology that invokes certain methods on objects without having to hard code the names of the messages in the composing entity, and without having to resort to meta programming.

This technique can also be used to group objects together in a higher order construction called "swarm". The goal of the swarm is to make sure that objects — that are unrelated in the sense that they neither refer to each other nor delegate to each other — move systematically across the network.

The following code excerpt shows how the swarm is implemented. `Swarm` takes a table of first class methods. Calling it creates a new active object that understands `doMove` with any number of arguments. All that it does is, upon arriving at a new location, to call all the first class methods. If these methods are move methods, this causes the objects they encapsulate to follow the swarm automatically. In analogy with what we said in section 6.9, this is actually a higher order mobility construct. It can be used with any number of — totally unrelated — objects that contain move methods with any number of arguments.

```
aview.Swarm@methods::{
  move.doMove@args :: {
   i:0;
   while(i<size(methods),
     methods[i:=i+1]@args) } }
```

The following code excerpt illustrates how the swarm higher order construction can be used. It shows the creation of three types of objects, bees, birds and fish. All three of them implement a one argument move method that is called `swarm`, `fly` and `swim` respectively.

```
aview.Bee(country)::{
  display("Bee created in ",country);
  move.swarm(c)::{
    display("Bee arrives in ",country:=c)}}

aview.Bird(country)::{
  display("Bird is born in ",country);
  move.fly(c)::{
    display("Bird just flew to ",country:=c)}}
```

```
aview.Fish(country)::{
  display("A fish sees the territorial waters of ",country);
  move.swim(c)::{
    display("Fish reaches the coast of ",country:=c)}}
```

We can illustrate how the code works by evaluating

```
{ b1:Bee("Belgium"); b2:Bird("Belgium"); f:Fish("Belgium") ;
  s:Swarm(b1.swarm,b2.fly,f.swim);
  s.register("aZoo")  }
```

on one machine and then calling `members("aZoo")[1].doMove("Holland")` from another machine. As a result the entire swarm will be moved to the new machine. Upon arrival the move method's body is executed which will invoke the move methods on all the component objects. The only thing that constraints this technique is that the first class methods grouped by the swarm have to accept an identical number of arguments. Apart from this, the objects in the swarm can be totally unrelated.

A Delegating Swarm

Another example of a swarm can be programmed by connecting objects along a parent-of link. The basic idea consists of a move method that will move an object in such a way that the machinery of the move method will "drag along" all the descendants of that object as well. In order to illustrate the basic mechanism, consider the following code excerpt.

```
aview.parent()::{
  move.comeHere()::{void};
  retract()::athis(). follow();
  aview.descendant()::{
   move. follow()::{void}} }
```

The code shows views that create a parent object and a descendant that inherits from that parent. With the move method `follow` in the descendant, the descendant can be moved to another machine than the one on which the parent is residing. However by sending `retract` to the descendant, method lookup will cause the parent to send `follow` to its descendant. The result is that the descendant will be returned to the parent location. On its own, we might call this technique a *retraction pattern* because it allows a parent to retract remote descendants to its own location.

The retraction pattern can be used to implement a swarm if it is combined with the MVC-technique presented in section 6.8 in which the view methods are used to make a parent keep a reference to all the descendants it generates. In section 6.8 we have called the collection of all such references `us`.

As an example of such a swarm implementation that uses delegation, consider a timetable system for the Brussels public transportation system which can be conceived as a graph of nodes some of which are underground stations, busstops, regional express net train stations and tramway stops. Many junction nodes are two or more of these at the same time. Suppose we want to design a system that allows every node to contain its time table such that users can enter the station with their PDA, connect to the object that is registered on the channel[4] `"MIVB Schedule"` send it the message `schedule().show()`. The result is a digital schedule of that node on their PDA.

Every node can be one or more types. Moreover, the types of node share information such as the name of the node, a map, a collection of amenities (e.g. is there an elevator for disabled people?) and so on. Every schedule is in a view on the object being the identity of the node. The structure of the code looks as follows[5].

```
aview.Node(name, amenities, map)::{
    us:void;

    aview.AsBusStop(schedule,line,isNightBus)::{
      ... functionality of a busstop ...
      asuper(us:=[athis(),us]) };

    aview.AsTramWayStop(schedule,line)::{
      ... functionality of a tramway ...
      asuper(us:=[athis(),us]) };

    aview.AsMetroStation(frequency,trunk)::{
      ... functionality of an underground station ...
      asuper(us:=[athis(),us]) };

    aview.AsGENTrain(railwayCompanyRef)::{
      ... functionality of a commutor train station ...
      asuper(us:=[athis(),us]) };

    show()::{
     ... UI code to set up the schedule ... }

    clone.copy()::
      ... copy for individualized behaviour ...

    move.down()::{
      fst:us;
      while(not(is_void(fst)),
        { fst[1].down();
          fst:=fst[2] })};
```

[4]MIVB (Maatschappij voor Intercommunaal Vervoer Brussel) is the name of the public exploitation company of the public transportation system.

[5]GEN means “Regional Express Net” in Dutch; local commuter trains.

```
    aview.schedule()::{
      s:asuper().copy().down();
      show()::s.show() };
  }
```

By sending `schedule` to an object reference corresponding to a `Node`, a local view is created that contains a local copy of the node schedule. This is accomplished by referring to the remote schedule by `asuper()`, sending `copy()` to this remote reference and subsequently pulling the copy to the local machine by sending it `down()`. The copy mechanism is used to make the example more realistic: one cannot just pull the schedule of the node to one's PDA because that would preclude other passengers from using the same schedule. Therefore a copy of the object is "downloaded". The `down` move method shows the implementation of the delegation-based swarm. Because of ChitChat's object extension mechanisms, every active view that was created on a node, is registered with the node by adding it to the `us` list. The `down` move method traverses this list and calls a move method on the views as well. This makes sure that all the views travel along with their parent node. This is what we mean by a delegation-based swarm.

9.9 Evaluation and Epilog

Having explained the design of ChitChat's mobility mechanism and having presented some patterns and techniques to try to validate its expressiveness somehow, let us now take a step back and contemplate the proposal.

9.9.1 Rough Edges

Although we believe that the ChitChat model is more than an initial step in the direction towards "structured mobility", there are some rough edges left that need more work:

- One of the problems is related to the network-reference semantics explained in the previous chapter. We stated that every time passive objects cross network boundaries (i.e., when using them as arguments or as results of methods) they get copied. As explained in this chapter, this semantics is also used for defining mobility. This means that, when two active objects share a passive object (e.g. they refer to the same table on one machine) and one of them is pulled towards another machine, then the passive object gets copied. In the previous chapter we explained that it is therefore not a good idea to share passive structures between active objects. The philosophy of the model is to think in terms of active objects that own private passive data structures. Still, this is not enforced by the language and it allows for subtle race condition bugs to sneak in easily.

- As explained in chapter 8, an ordinary view that resides inside an active object will generate (passive) objects that have the behaviour of the active object as a parent. Therefore, when this behaviour is moved, all possible (passive) extensions of this behaviour have to be moved along. This can be problematic if these extensions have left "their" active object and are referred to from within other active objects. The current move semantics has to make sure an error is generated when trying to move an active object that has local passive extensions that might be referred to from within other active objects. As already explained at the end of chapter 8 we plan to change the semantics such that passive objects can never leave "their" active object as in Eiffel. This way such local extensions cannot leave the active object and will therefore alwasy be consistently moved along when moving the active object.

- A related problem is that mobility can break sharing of constants between clones. Remember from chapter 5 that cloning semantics prescribes that only the mutable part of an object is cloned. The immutable dictionaries are shared between the object and the clones that it generates. This means that immutable slots can contain *mutable* data that is shared between an object and its clones (e.g. a table or another object). However, upon moving one of the clones, the immutable data will be copied across the network if it concerns passive data such as a table or a passive object. This breaks the sharing relation between the object and its clone. A possible answer is that ChitChat immutable slots are public slots and that the philosophy of the model is to align public data with active objects which are never copied but passed by reference. As such, a public slot that contains an active object causes no harm as the active slot is moved by reference such that is stays shared between the object and its moved clone. But we do not consider this as a satisfactory answer.

- ChitChat move methods are executed *after* the receiving objects are relocated from the device on which they reside to the device on which the sender of the "move message" resides. However, while experimenting with our prototype-implementation we have often felt the need to do something *before* the move is actually accomplished. A natural reaction might be to implement an ordinary method that runs whatever has to be done before the move, and then invoking the move method by a self-send. However, this does not work because, at the time of the self send, the sender of the "move message" and the receiver are on the same machine. Hence, nothing will move and the move method will not be executed. A trick to bypass this shortcoming is to make a local view that contains a method to do whatever is needed before the move and then call the move method with a super send. This technique was used in the public transportation example. The local `schedule` view is used to first copy the super and then send it the "move message". Although it works, we have to admit that is is pretty cumbersome.

- Move methods are methods that change the location of an object. This location can be considered as part of the state of the object and therefore, move methods easily give rise to race conditions on that state. This is a complex way to say that two parties might be "fighting" for the same object by both sending a "move message" to the same object. Depending on the relative speed of the connections, the first or the second object will win the race. This object will have lost the fight because the last executed move method will cause the object to be pulled towards the sender of that message.

  At the time of writing, we have no simple solution for this problem. One answer might be to say that ChitChat is not designed for programming mobile agents that end up in "hostile" situations where objects are "fighting" for the agent to be at their location. Hence, it is the task of the programmer to make sure objects in a system never have to answer two "move messages" like this in row. However, even in our limited experiments we already ended up in situations where the final location of an object depends on whichever move method lost the race and was executed last. A solution probably consist of designing higher order abstractions that allow a "batch" of messages to be sent between two objects while they reside on the same machine. This could be accomplished by pairing a move method with a "trampoline method" that sends the object back to where it came from due to the initial move method. Other move methods are never executed before the trampoline method of the last executed move method is executed as well. We consider this as future work. Notice that this is a form of behavioural synchronisation. Just like a buffer's job consist of putting "get" messages on hold if the buffer is empty, an object has to put certain move methods on hold until the state of the object is ready to process the next move.

9.9.2 Security and Extreme Encapsulation

Now that we have presented the entire ChitChat model, it is instructive to reconsider our analysis of chapter 3 where we concluded that the powerful prototype-based programming languages presented in chapter 2 have inherent security problems because of their object model being too rich with respect to the operators defined on it. We showed this being problematic for distribution in open networks, but especially in the context of mobility.

In this and the previous chapters, we have shown that it *is* possible to design a full-fledged prototype-based programming language that features cloning, object extension and even prototype-based distribution and strong mobility without sacrificing the fundamental encapsulation guarantee offered by a late-bound message-passing operator. In ChitChat *every* object operation is subject to late binding polymorphism and it is completely up to the receiving object to define what to do with a message that is sent to it. Depending on the type of attribute operations as different as method invocation, cloning, moving or connected applet creation are accomplished. Furthermore, because of the absence of encapsulation breaching operations, only applications of the Granovetter operator allow objects to obtain references to other objects. Although these language characteristics do not imply that all software written in ChitChat is by definition secure, we *can* conclude that the language at least *allows* for secure software to be written. Once well-encapsulated mobile and/or distributed objects are designed, implemented, tested and released on a network, there really is no way — at the language level at least — to bypass their encapsulation.

9.9.3 Us

A remarkable observation made by the programming experience gained so far in ChitChat is that many designs require the notion of `us`; i.e., a collection containing all descendants of an object. The patterns of section 9.8 have also given a number of examples that heavily depend on this idiom. Thanks to extreme encapsulation, it is easy for an object to keep a reference to all the descendants it has spawned. We think it would therefore be useful to build this into the language. `us` could even be a generalization of "this", a value that constantly refers to all descendants of an object. Notice that until now we have no mechanism to remove a descendant from the `us` list and this is also very hard to implement because that list is conceived using regular object references such that a garbage collector can never collect a descendant if it is no longer referred from within other objects. And conversely, the garbage collector can therefore not notify the `us` list that one of the objects it contains has become obsolete. Building the concept into the language would alleviate this problem.

9.9.4 Mobility Models

In section 9.6.1 we have distinguished between pull technology, push technology and agent technology. ChitChat is pretty unique in this respect because it is — as far as we know — the first language that combines strong mobility with pull technology. All other language proposals reviewed in section 9.5 either use push technology (e.g. Emerald and TeleScript) or agent technolgy (e.g. Borg and Obliq). Until now, the pull model was restricted to applet systems that only use weak forms of mobility. In section 9.7.1, we explained that ChitChat's pull model is a direct consequence of the argument against a "move" operator because both the object being moved as well as the receiving location need to play an active role in the move process. ChitChat demonstrates that Vigna's critique on mobile agents and his conclusion that code on demand is a better scheme (see [Vig04]) does not necessarily exclude strong mobility.

Moreover, we have shown in our conception of the itinerary pattern and the master-slave pattern that it is quite easy to implement agent-like and push-like interactions using a combination of move methods and the general object-oriented double dispatch technique: one simply demands a host to pull itself towards it.

9.9.5 Implementation

The ChitChat model was given a prototype implementation in Java. We have used a simulation architecture in which several "distributed interpreters" with completely disjoint runtime structures run in different windows. Internally the interpreters are implemented as if they are really running on different machinery. The only complexity we bypassed was the underlying network technology. Our interpreters marshall their data contexts and runtime context using standard Java serialization. The communication between two interpreters is achieved by handing over a binary stream from one interpreter to another. The receiving interpreter unmarshalls the stream back into an object graph.

Efficiency was not our primary concern during the implementation because we basically had no idea what to implement. During our research, the implementation was mainly used as an experimentarium to distill the language features presented in the dissertation. But now that we have an initial understanding of these features we plan to implement an actually distributed version in the context of the PhD research of new students.

In order to correctly implement the serialisation of both the data context as well as the runtime context, it was impossible to implement the interpreter in a standard recursive way. The reason is that Java does not feature "current continuations" such that it is impossible for an interpreter to grab the runtime context and transmit it in order to continue the execution on the other site. We therefore implemented the entire interpreter in explicit continuation style. Continuations are "frames" that contain slots for actual parameters and a backward link to propagate results back to their calling continuations. Roughly spoken, at any moment in time during the execution of a ChitChat interpreter, a linked list of continuations defines the runtime context of the interpreter. Strong mobility is then implemented by marshalling and transmitting this list of continuations.

Remote references were implemented by giving every interpreter a bidirectional table that maps remote references to actual objects (for incoming references) and to machines (for outgoing references). This implementation of networked pointers is overly simplistic especially in the context of open networks in which hardware can unexpectedly disappear. We plan to replace this implementation by a conception of "ruberband pointers" that "keep on referring to other devices even though they are temporarily out of earshot". This research is currently being conducted [DV04].

In brief, the ChitChat model presented in this dissertation was given a prototype-implementation in which the technical network component was ignored. The logical network layer was taken into account however.

9.10 Conclusion

In this chapter we presented ChitChat's mobility model. We have given a basic overview of mobility models, we presented several technical reasons for motivating strong mobility — even without resorting to visionary scenarios of agents that autonomously hop between machines — and we have looked at the basic techniques currently in use to implement mobile software. We have argued that these techniques are practically all based on Java technology and that Java technology currently is highly unsuited for the purpose of writing strongly mobile software.

We have subsequently looked at other programming languages (to wit Emerald, Borg, Obliq and TeleScript) that have built-in features for strong mobility Although the technical conception of mobility in these languages clearly outranks the cumbersome Java-based architectures, we have argued that they still fall short on high level mobility constructions. With the exception of Emerald's `attach` declaration and Obliq's remote cloning, all mobility is entirely based on an explicit `move` instruction that bluntly moves an object from one machine to another. We have presented an extensive argument against this conception of mobility. Based on an analysis of the mobility process from a software engineering point of view and from a security point of view, we have distilled a number of restrictions for a mobility mechanism that was the basis for ChitChat's move methods and visit methods.

We have tried to validate the model by conducting a number of experiments in ChitChat that demonstrate its expressivity. This comprises an adaptation of chat server of chapter 8 with mobility, the elegant expression of a number of existing mobility patterns (to wit itinerary, master-slave and stationary), and the distilling of a number of programming techniques that elegantly combine ChitChat's specific features (to wit move methods, visit methods, Pico's functional parameter passing and first class methods) into powerful higher order mobility abstractions such as local algorithm execution and swarms.

Chapter 10

Conclusions, Related and Future Work

The final chapter of the dissertation takes a step back and contemplates the results. We summarize the work presented in the dissertation thereby referring back to the "promises" made in the introduction. We give some pointers to related work that is currently in progress and we speculate on how the research can be continued by listing topics for future work, some of which is already underway.

10.1 Summary and Contributions

The contributions presented in the course of this dissertation can be described from two different points of view. The technical contributions are summarized in section 10.1.2.

10.1.1 Contributions to the OO Field

From a more research field-driven point of view, we have conducted an exercise in language design that unifies knowledge from three fairly independent schools of object-oriented language design. Indeed, ChitChat can be seen as a cross-fertilization of the following three language design schools.

1. ChitChat's concurrency model was heavily influenced by the one from ABCL, as far as we know the most powerful attempt in combining the flexibility of the actor paradigm with the stateful ideas behind object-orientation. The result is a stateful object-oriented language that features both synchronous and asynchronous message passing. In the case of asynchronous message passing a system with transparent futures takes care of synchronization between the sender and the receiver of the message whenever this is necessary. Therefore, to the best of our knowledge, we have adopted **the state of the art in object-oriented concurrency**.

2. We have adopted numerous ideas of delegation-based prototype-based languages. Whereas previous languages were also often prototype-based (like Obliq) or at least object-based (e.g. Emerald), to the best of our knowledge, no language combines concurrency with delegation. Prototype-based languages that have been combined with concurrency and distribution feature no delegation with parent sharing. We have shown that by cleverly designing the scope rules between parents and descendants, parent sharing can serve as a powerful paradigm to handle shared state between concurrent and distributed objects. By incorporating features like views, mixins and first class methods in the context of extreme encapsulation, we have therefore also made progress in **the state of the art in prototype-based language design**.

3. Finally, we have adopted features of existing distributed and mobile languages, and in particular distribution transparency in the sense that no explicit machinery has to be provided by programmers to send messages to objects residing on other machines. Furthermore, we have argued how a low level "move" instruction is a source of problems in structuring mobile software, both from a software engineering point of view as well as from a security point of view. The model we proposed is based on move methods and visit methods and seems to impose much more structure on a mobile application. We consider it as a first proposal towards languages that feature *structured mobility* in analogy with structured programming that succeeded goto-programming. We therefore claim to have made progress in **the state of the art in mobile programming language design**.

It is the combination of these feature that renders ChitChat a unique exercise in language design. Although the model still has some rough edges left, we think these are not fundamental to the combination of features we incorporated. It is our firm belief they are merely technical in nature and can be solved by short-term research and polishing. Solving them is a matter of transpiration and not of inspiration. We will get back to this in section 10.3.

10.1.2 Technical Contributions

Aside from these research field-oriented contemplations, the technical contributions ChitChat makes are:

An intersection between classes and prototypes. We have shown in chapter 3 that classical prototype-based programming languages suffer from a number of inherent security problems because of the way their manifold of operators requires their object model to be much more complex than the simple record-based message passing centred object model of class-based languages. We have shown in chapters 4 and 5 how these problems can be avoided — thereby sticking to our **extreme encapsulation** principle — without ending up with a class-based language or without ending up with a prototype-based language that excludes powerful features such as dynamic object extension and parent sharing. The insight is to allow an object to contain different kinds of methods which — upon invoking — have different effects depending on their type. The technique originates in Agora. It was refined in Pic%.

Extension from the outside. Although the formal semantics of this paradigm and its theoretical situation in the design space of object-oriented programming languages is original research, the language itself is not. It was already presented in the PhD thesis of Steyaert [Ste94]. Nevertheless we have contributed a lot to the understanding of the paradigm. One of the critiques one often gets when presenting this paradigm is that it is counter intuitive to object-oriented programming because it requires all potential extensions of objects to reside in those objects. In section 4.6.2 and 5.9.3 we have shown how this can be circumvented by reflection techniques or by clever language design resulting in extension from the outside. At the heart of the solution however stays message passing. An object that should not be extensible simply does not implement those messages and will never be subject to extension.

An intersection of lambda-based languages and object-based languages. In the chapter on Pic% we have seen how the notions of first class methods and functions can be aligned. By storing methods statelessly in a dictionary (as opposed to closures) we get the efficiency of object-oriented reentrancy. The key however to first class methods consists of creating a closure upon referencing the method in any way. At that point the frame in which the method resides is taken as its lexical scope. The original receiver from which the method was selected is also wrapped in the closure such that later references to `athis()` in the code of the method are indeed redirected to the original receiver. In chapter 6 we have shown how this allows for a very elegant implementation of object-oriented patterns where other languages require block constructs or clumsy constructions such as Java's anonymous classes.

A concurrency model that reconciles active objects with delegation. Whereas this was considered impossible up until now, in chapter 8 we have shown how to unify these two language characteristics. The key to the unification was to preclude "internal delegation" in active objects: whereas external messages are normally looked for along the delegation chain, methods can only "see" the context of the object in which the method resides. This way active objects can be linked to form delegation chains without causing race conditions. However, two special scope functions `asuper(...)` and `athis(...)` allow parent objects and descendants to access each other's state in a controlled way. Especially `asuper(...)` is useful: it allows for an atomic execution of code of the descendant in the context of the parent. This way, parents of active objects can store shared state. `asuper(...)` is used by descendants to modify this state in an atomic way such that no race conditions occur.

A distribution model that aligns proxies with delegation-based descendants. By generalizing the proxy concept used by distribution models towards full-fledged descendants in the delegation-based sense, we actually generalized the notion of a proxy. In our model, a proxy is nothing but a delegation-based descendant that appears to reside on another machine than the original object. Because of the fact that two objects can share the same parent, we can handle multiple proxies for the same object in such a way that the state of the object stays consistent for all proxies. The distribution model is a generalisation of the concurrency model: the active objects that share the same active parent are allowed to reside on different machines. The result is a distributed object model in which shared parents are used to store structurally shared state of the distributed system.

Move in mobile language was argued to be a harmful language feature. This was analysed both from a software engineering perspective of which the basic argument was that a "move" operator results in incomprehensible object soups, as well as from a security point of view where the argument was that agents bluntly put on another machine are harmful for security unless one resorts to full-mobility. We have presented an initial proposal for structured mobile programming — move methods and visit methods — in which all parties — the sender, the receiver and the object being moved — are taken into consideration in a move, and which respects the Granovetter principle that connectivity begets connectivity, also in the context of mobility. The model was given a prototype-implementation and was experimented with. This has yielded some initial, though powerful programming idioms such as higher order mobility.

Higher order mobility patterns such as the implementation of object swarms in section 9.8.5 were introduced. Just as is the case with component wiring in chapter 5, first class functions seem to combine very well with move methods and visit methods. Because of the fact that they can be treated as any other value, they are subject to composition.

10.1.3 The Fundamental Problems, Continued

The above considerations can also be reformulated in the terminology used in chapter 1. There we stated that it was not our goal to solve all problems related to language design for AmI but that we *did* want to make a contribution by attempting to solve four fundamental problems. Let us reconsider them one by one:

The Ambient Object Paradigm Problem.

The first problem consists of the fact that neither existing class-based, nor existing prototype-based programming languages are a viable option to design AmI applications in a reliable and consistent fashion. Class-based languages have some inherent paradigmatic problems in open networks and (interesting) prototype-based languages suffer from tremendous security problems. Rather than sticking to prototype-based languages that exclude the powerful features of prototype-based programming (as many distributed and mobile languages currently do), we have shown that this apparent stalemate can be resolved quite elegantly. We formulated the extreme encapsulation principle as a yardstick and have shown that languages in which objects can offer special kinds of attributes open up vast possibilities for object extension, cloning, complex inter-object scoping mechanism, distribution, concurrency and mobility in several flavours. The point is that objects only respond to messages that *can* be implemented by special kinds of methods that act in special ways upon invocation. The result is a fully empowered prototype-based language that offers all the security needed for open networks.

The Concurrent Parent Sharing Problem.

A second problem was the tension between prototype-based object-oriented state sharing on the one hand, and the avoidance of shared state for concurrency reasons on the other hand. We have designed an object-oriented language that seems to turn this reasoning upside down: ChitChat's concurrency model encourages state sharing between different objects in the form of a parent object that is shared between child objects. Since objects are (thanks to extreme encapsulation) fully in charge over the descendants they generate, this allows for a very controlled way of sharing state between two active objects. Moreover, some cleverly designed scoping mechanisms allowing the atomic execution of descendant code in the context of the shared parent seems a promising abstraction to avoid race conditions between two active objects.

The Distributed Sharing Problem.

We have argued in the introduction that writing distributed systems is all about writing code that shares structures on geographically dispersed machinery. We have also argued that current day distributed programming languages only allow dispersed code to share (object) references. This is because these languages take the object to be the unit of distribution. ChitChat's distribution model is an experiment that sets aside this premise and explicitly allows distributed objects to share their common structures in a shared parent object. The shared parent thus forms a structural sharing mechanism while standard networked pointers are a mere referential sharing mechanism. The special scoping functions discussed above allow code from one node to execute in the context of the shared parent which contains the structures shared by all the components of the distributed system.

Move Considered Harmful.

We have argued that a programming language that has the dynamic features to write programs comfortably that are resilient enough to survive in open networks, is bound to be a programming language with strong mobility features. We have argued that contemporary mobility proposals in our context will lead to insecure software that consists of unmanageable distributed object soups. We have argued that the culprit of this is the "flat" move instruction that forms the basis for every strongly mobile programming language and that in future languages a form of structured mobility will be needed. We have conducted an analysis in which we have unraveled the different parties involved in moving a running object and have formulated an initial proposal — move methods and visit methods — that tries to structure mobility somehow.

10.2 Related Work

Although some of the language features presented in the dissertation might seem quite unorthodox and exotic, it is our impression that the time is probably ripe for such kind of features to start influencing our languages. Proof of this is the exotic networked lexical scoping scheme already featured by Obliq in 1995, but also a number of more recent attempts to reconcile the dynamics of classless object-oriented programming and distribution in the context of open networks.

10.2.1 dSelf

Not so long ago, a distributed version of Self was presented by Robert Tolksdorf and Kai Knubben [TK02]. Unfortunately, the authors only present the idea of different Self interpreters communicating with each other. Distributed method lookup is presented as well. However, as far as can be deduced from the paper, all communication is purely synchronized as no concurrency model is mentioned. Furthermore the authors do not investigate the implications of distributed inheritance at all. They *do* mention the fact that a (old) Self feature, namely local methods, can be used to "shadow" a remote method locally. This roughly corresponds to ChitChat's method overriding in proxies. dSelf does not mention any implications this scheme has on the meaning of the `self` pseudo variable, an essential ingredient of delegation. Furthermore dSelf does not feature mobility. We consider dSelf more important as a language that helps motivating our work than as a serious attempt in distributed programming language design.

10.2.2 IO

IO is an open source project led by Steve Dekorte [Dek04]. Although, to the best of our knowledge, nothing has been officially published about the language, the project seems to progress. IO is a prototype-based language with a syntax very much like Pic%'s. It features a comb-line inheritance scheme that is directly based on NewtonScript. IO features an ABCL-like concurrency model as well. Using special annotations, messages to objects can be sent synchronously, "purely" asynchronously or asynchronously with transparant futures as in ChitChat. Like Pico and Pic%, IO features a reflection model based on modifiable trees instead of bytecodes.

Although IO needs much more work, the website lists a number of speculative features the designers want to see added in the near future. They list such things as *"multi-state (multiple independent VMs can run in the same application)"*. Although it is not further specified what this means precisely, we believe it endorses ChitChat's vision of structural sharing of objects between different machines.

10.2.3 E

A language that only recently came to our attention but deserves some more consideration is the E language designed by Mark Miller [MMF01]. The similarities, particularly in vision, between E and ChitChat are striking. However, while ChitChat is the result of object-oriented language designers applying their ideas in the context of (insecure) open networks, E is the result of security specialists trying to get the best out of the object-oriented paradigm.

Both E and Pico (and thus also Pic% and ChitChat) have readability and palatability for people acquainted with C-like syntax as explicit goals. Therefore a lot of effort was spent on the design of a conventional syntax in E as well.

Both languages endorse the extreme encapsulation principle. In E this principle is covered by the motto “do not add security — remove insecurity”. The result is a language in which objects are considered as fully encapsulated dictionaries mapping names to attributes. Inheritance is — like in Pic% — accomplished by a combination of lexical scoping, methods and reified environments. Shared parents in E are used to communicate secret data between objects. An illustrative example of this in E is the creation of purse objects that carry around digital money. In order for money to have a value all purses should share the same value system (otherwise purse objects can be created or copied without a central authority, rendering it worthless) called a mint. In E the purses have the mint as a shared parent.

For concurrency and distribution, E features an asynchronous message passing operator (known as the “eventually” operator in E) that returns promises, just like in ABCL or ChitChat. However, E’s promises are not transparant and a “when *promise* do *action*” construct is needed to specify what will happen upon fulfilling a promise. This can actually be considered a bit closer to the original actor paradigm than ChitChat’s transparant promises because the “when do” construct resembles a continuation very much. However, E allows one to wrap this construct in an object that intercepts all messages until the promise is fulfilled.

Another resemblance between ChitChat and E that is very fundamental is that both languages are experiments which leave the “standard” track that objects are to be the unit of distribution. Although their mechanisms to do so differ strongly, the goals are the same: to make distributed objects share part of their internal structure with other objects. This is in sharp contrast to other proposals where distributed entities only contain references to a shared party in a way such that they both have to communicate through message passing with that party. In ChitChat this structural sharing over a network is accomplished by promoting objects to be shared parents of distributed objects. In E, shared objects can form an “Unum” so that they share a single replicated state that is kept consistent over the network.

10.3 Shortcomings and Future Work

Research is never finished and neither is the one presented in this dissertation. In the very last section of the dissertation, we list a number of topics for further investigation and for perfectioning the ChitChat model. Section 10.3.2 presents some pretty open-ended topics that will require considerable redesigning of the model. Before doing so, we first enumerate a number of shortcomings and design mistakes in the model as presented in this dissertation.

10.3.1 Shortcomings and Design Mistakes

A number of **language features that are considered indispensible** for any modern programming language still have to be **added** to the model, and their interaction with the existing features will need some investigation. E.g. in the model presented so far, no attention was given to features such as introspection, elementary meta-programming and standard C++-like exception handling. Their interactions have to be investigated with the rest of the model.

As explained in section 8.4.5, the interleaving of super and self references using the `asuper(...)` and `athis(...)` scope functions pretty easily leads to **deadlock** especially for calls to the scope functions that are not in a tail position. This definitely needs further attention in order to make the model practically applicable.

It was explained both in chapter 8 and in chapter 9 that **copying passive object structures** every time they cross network boundaries is **problematic**. The problem manifests itself when passive object structures are used as arguments or results in combination with remote method invocations. This was also deemed problematic in chapter 9 in the case that two active objects sharing a passive one and that one of those active objects is moved across the network. In that case, the passive structures are duplicated because of the copy semantics. A possible solution to this problem is to preclude passive data from leaving the active object in which it was created, neither as an argument of a message sent nor as the result of answering a message sent. This way, all active objects would contain "their" passive data structures and sharing passive data structures between active objects always yields an error.

As explained at the end of chapter 8, ChitChat **object structures** arising from combining active and passive hierarchies as depicted in figure 8.4 are **not always easy to understand**. Proof of this is the complexity of the runtime context functions as defined in table 8.1. We hope to simplify this model in future iterations over the language. The route to a solution will probably consist of completely uncoupling the creation of active objects and passive objects and consider them as two totally different objects that should not be confused in successive view and active view applications. The current complexity actually stems from the fact that we originally started our research with the goal of mixing active and passive objects freely, both in containment relationships as well as in delegation relationships.

Finally, something that we experience as a shortcoming is the **lack of experience in deeply exploring the program design space** that is delimited by the interaction of ChitChat's features. Although we already found some elegant applications of the presented features and some nice patterns arising from combining first class methods and move (or visit) methods (e.g. the swarm pattern and the component wiring pattern) and combining ChitChat's exotic parameter passing mechanism with visit methods (e.g. the local algorithm execution pattern), we feel that there is more potential to the model and that was not systematically studied. Questions that fully explore the design space such as "what can be accomplished by invoking a cloning method by a self-send from within a visit method" have been left unanswered. We definitely feel the need to write bigger programs in ChitChat in order for more patterns and useful combinations to arise.

10.3.2 Future Work

The previous sections have presented some shortcomings of the model presented in this dissertation as well as some "local" extensions needed by the model, that will probably not interfere too much with the existing features. This section presents some more challenging issues in order to yield a production language for AmI.

As was already mentioned several times, **partial failure** — the idea that part of a system goes down while the rest keeps on performing useful computations — is one of the issues that distinguishes distributed systems from "ordinary" concurrent ones. Although the mobility model of ChitChat offers *some* help for anticipated partial failures, the real crux is unanticipated partial failure. This is already highly problematic in "ordinary distributed systems" but gets especially daunting in our research context of open distribution. The notion of partial failure is related to the notion of **disconnected operation**, i.e., the ability of a part of a distributed system to continue functioning even though the connections it needs (and more importantly, the parties on the other side of those connections) are temporarily broken. In open networks this will be the rule rather than the exception as devices can enter and leave the processor cloud at unforeseen times. We currently envision two possible research tracks to tackle these problems. The first one is currently being investigated by other members of our lab [DV04] [DDV03]. It basically consists of implementing **"rubberband" pointers** over a network using connectionless actors at lower levels. The actors are equipped with ingoing and outgoing mailboxes and (re)joining a network gives rise to messages being sent to those mailboxes which allows the system to (re)establish connections. This "connectionless system level technology" will be used to support the connection-oriented features presented in this dissertation. Another track that we plan to look at is the field of **reversible computations.** Examples of the latter currently exist in monadic form in functional programming, a track we already explored shortly in [De 97].

If we consider prototype-based programming as combining objects by means of "has-a" links (i.e., objects containing other objects) and "is-a" links (i.e., objects delegating to other objects) then ChitChat is an experiment that tries to learn from the consequences of distributing the is-a link over different nodes in a network. An interesting research track that is orthogonal to the research presented here is to distribute the "has-a" link as well. This way different objects could share the same instance variables, much in the spirit of class (a.k.a. static) variables, but made explicit instead of being hidden by language machinery. This could give rise to **distributed cloning families**; objects that are idiosyncratic entities but which share static variables over a network. Replication machinery is responsible for keeping those static variables consistent and meta programming could be adopted to adjust the replication machinery to the needs of particular applications.

The idea of a processor cloud can also be adopted to the level of the programming language in the incarnation of what we have called **multivalues** in [DDD03a]. Multivalues *are* many values and performing operations on a multivalue asynchronously performs the operation on every value in the multivalue. The result of performing an operation on a multivalue is a multivalue of results. Multivalues are close to the array programming features recently added to Smalltalk [MD03]. Adding such an algebra to a programming language could prove extremely useful to model sensors whose values constantly change due to mobile hardware. First class methods could be used to be attached to a "smell variable" whose multivalue changes constantly. The first class method is invoked on every value produced by that variable.

Finally, the move and visit methods proposal is just the tip of the iceberg. We have clearly formulated our vision that bluntly moving objects is harmful but we have also clearly stated that move methods and visit methods are but a first technical proposal in the direction of **structured mobility**. On the one hand we need much more experience to understand fully the implications of this kind of mechanisms. On the other hand, the mechanism itself needs more work. For example, at the end of chapter 9 we explained that race conditions might occur when two move methods execute simultaneously leaving some objects of the delegation chain on one machine and others on another. This might indicate that structured mobility will require more inter-machine constructions in the spirit of ChitChat's shared parent techniques. We therefore have the impression of having only scratched the surface of an exciting new field of research.

Appendix A

Pic% Semantics

This is not a formal definition of Pic%, but rather a highly structured informal definition. For reasons of readability we assume a meta formalism with mutable state instead of passing on explicit state parameters with every function as would be the case in a full-fledged denotational description. Many equations are written with superscript C. This is a context parameter that refers to scope pointers at all times. We decided not to pass this as an explicit argument purely for esthetic reasons. It would — in our opinion — have cluttered up the equations and made them less readable.

Concerning Pic%, we limit ourselves to the core language and do *not* give semantics for:

- super sends.
- multidimensional tables.
- quoting and meta provisions.
- operator definitions and usage.
- multiple and variable argument functions and the apply operator.
- first-class continuations
- apart from `this()` and `super()`, no natives are treated

Super sends are easily added by passing along an extra argument in the context. This is not an essential feature but merely makes the denotations harder to read, especially the DO^C equations given below. The semantics of first-class continuations is far beyond the goal of our dissertation. It makes the denotations needlessly unreadable and can be found back in nearly every introductory text on denotational semantics.

A.1 Values and Abstract Grammar

The following shows tagged elements of the abstract grammar and tagged values. We use subscript notation for selecting subexpressions. For example, the components of a $frm \in \mathbf{FRM}$ are selected as frm_{cst}, frm_{var} and frm_{nxt}. The names of the "fields" that we use in the subscript can be derived from the second column.

Name	**Tagged Value**	**Set**
Basic Values		$\mathbf{SLF}$
Number	$nbr(n), n \in \mathbb{N}$	$\mathbf{NBR}$
Fraction	$frc(f), f \in \mathbb{R}$	$\mathbf{FRC}$
Text	$txt(s), s \in String$	$\mathbf{TXT}$
Table	$tab(a), a \in Array$	$\mathbf{TAB}$
Void	voi	$\mathbf{VOI}$
Expressions		$\mathbf{EXP}$
Reference	$ref(nam)$	$\mathbf{REF} \approx ref(\mathbf{NAM})$
Tabulation	$tbl(exp, idx)$	$\mathbf{TBL} \approx tbl(\mathbf{EXP}, \mathbf{EXP})$
Application	$apl(exp, arg)$	$\mathbf{APL} \approx apl(\mathbf{EXP}, \mathbf{EXP})$
Message	$msg(exp, inv)$	$\mathbf{MSG} \approx msg(\mathbf{EXP}, \mathbf{INV})$
Definition	$def(inv, exp)$	$\mathbf{DEF} \approx def(\mathbf{INV}, \mathbf{EXP})$
Declaration	$dcl(inv, exp)$	$\mathbf{DCL} \approx dcl(\mathbf{INV}, \mathbf{EXP})$
Assignment	$ass(inv, exp)$	$\mathbf{ASS} \approx ass(\mathbf{INV}, \mathbf{EXP})$
Attributes		$\mathbf{ATT}$
Function	$fun(nam, arg, bdy)$	$\mathbf{FUN} \approx fun(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
View	$viw(nam, arg, bdy)$	$\mathbf{VIW} \approx viw(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
Cloning	$cln(nam, arg, bdy)$	$\mathbf{CLN} \approx cln(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
Native	$nat(idx)$	$\mathbf{NAT}$
Objects		
Closure	$clo(att, cur, ths)$	$\mathbf{CLO} \approx clo(\mathbf{ATT}, \mathbf{FRM}, \mathbf{FRM})$
Binding	$bnd(nam, val, nxt)$	$\mathbf{BND} \approx bnd(\mathbf{NAM}, \mathbf{VAL}, \mathbf{BND})$
Frame	$frm(cst, var, nxt)$	$\mathbf{FRM} \approx frm(\mathbf{BND}, \mathbf{BND}, \mathbf{FRM})$
Context	$ctx(cur, ths)$	$\mathbf{CTX} \approx ctx(\mathbf{FRM}, \mathbf{FRM})$

$\mathbf{INV}$	$\equiv$	$\mathbf{REF} \cup \mathbf{TBL} \cup \mathbf{APL}$	(invocations)
$\mathbf{EXP}$	$\equiv$	$\mathbf{INV} \cup \mathbf{MSG} \cup \mathbf{DEF} \cup \mathbf{DCL} \cup \mathbf{ASS}$	(expression)
$\mathbf{ATT}$	$\equiv$	$\mathbf{FUN} \cup \mathbf{VIW} \cup \mathbf{CLN} \cup \mathbf{NAT}$	(attributes)
$\mathbf{SLF}$	$\equiv$	$\mathbf{NBR} \cup \mathbf{FRC} \cup \mathbf{TXT} \cup \mathbf{VOI} \cup \mathbf{TAB}$	(self evaluating)
$\mathbf{VAL}$	$\equiv$	$\mathbf{SLF} \cup \mathbf{CLO} \cup \mathbf{FRM}$	(values)
$\mathbf{AGR}$	$\equiv$	$\mathbf{SLF} \cup \mathbf{EXP} \cup \mathbf{ATT} \cup \mathbf{VAL}$	(abstract grammar)

A.2 Dictionary Structures

$newFrm : \mathbf{FRM} \rightarrow \mathbf{FRM}$
$newFrm(frm) \equiv frm(voi, voi, frm)$

$addVar : \mathbf{FRM} \times \mathbf{NAM} \times \mathbf{VAL} \rightarrow \mathbf{VAL}$
$addVar(frm, nam, val)$
$$\equiv \begin{cases} error & \text{if } getAny(frm, nam) \neq voi \\ \{frm_{var} := bnd(nam, val, frm_{var}); val\} & \text{otherwise} \end{cases}$$

$addCst : \mathbf{FRM} \times \mathbf{NAM} \times \mathbf{VAL} \rightarrow \mathbf{VAL}$
$addCst(frm, nam, val)$
$$\equiv \begin{cases} error & \text{if } getAny(frm, nam) \neq voi \\ \{frm_{cst} := bnd(nam, val, frm_{cst}); val\} & \text{otherwise} \end{cases}$$

$setBnd : \mathbf{BND} \times \mathbf{NAM} \times \mathbf{VAL} \rightarrow \mathbf{VAL}$
$$setBnd(bnd, nam, val) \equiv \begin{cases} error & \text{if } bnd = voi \\ \{bnd_{val} := val; val\} & \text{if } bnd_{nam} = nam \\ \quad setBnd(bnd_{nxt}, nam, val) & \text{otherwise} \end{cases}$$

$setVar : \mathbf{FRM} \times \mathbf{NAM} \times \mathbf{VAL} \rightarrow \mathbf{VAL}$
$setVar(voi, nam, val) \equiv error$
$setVar(frm, nam, val) \equiv \mathbf{let}\ v = setBnd(frm_{var}, nam, val)\ \mathbf{in}$
$$\begin{cases} setVar(frm_{nxt}, nam) & \text{if } v = error \\ v & \text{otherwise} \end{cases}$$

$getBnd : \mathbf{BND} \times \mathbf{NAM} \rightarrow \mathbf{VAL}$
$getBnd(voi, nam) \equiv voi$
$$getBnd(bnd, nam) \equiv \begin{cases} bnd_{val} & \text{if } bnd_{nam} = nam \\ getBnd(bnd_{nxt}, nam) & \text{otherwise} \end{cases}$$

$getCst : \mathbf{FRM} \times \mathbf{NAM} \times \mathbf{FRM} \rightarrow \mathbf{VAL}$
$getCst(voi, nam, slf) \equiv voi$
$getCst(frm, nam, slf) \equiv \mathbf{let}\ v = getBnd(frm_{cst}, nam)\ \mathbf{in}$
$$\begin{cases} close(v, frm, slf) & \text{if } v \neq voi \\ getCst(frm_{nxt}, nam, slf) & \text{if } bnd_{nam} = nam \end{cases}$$

$getAny : \mathbf{FRM} \times \mathbf{NAM} \rightarrow \mathbf{VAL}$
$getAny(voi, nam) \equiv voi$
$getAny(frm, nam) \equiv \mathbf{let}\ v = getBnd(frm_{cst}, nam)\ \mathbf{in}$
$$\begin{cases} v & \text{if } v \neq voi \\ \mathbf{let}\ v = getBnd(frm_{var}, nam)\ \mathbf{in} & \\ \left\{ \begin{array}{ll} v & \text{if } v \neq voi \\ getAny(frm_{nxt}, nam) & \text{otherwise} \end{array} \right\} & \text{otherwise} \end{cases}$$

A.2.1 Tables

$$\begin{aligned} makTab^C(nbr, exp) &\equiv tab([EVAL^C(exp^1), \ldots, EVAL^C(exp^{nbr})]) \\ getTab(tab, nbr) &\equiv tab[nbr] \\ setTab(tab, nbr, val) &\equiv \{tab[nbr] := val; tab\} \end{aligned}$$

A.2.2 Closure Creation

$close : \mathbf{AGR} \times \mathbf{FRM} \times \mathbf{FRM} \rightarrow \mathbf{AGR}$

$$close(v, cur, slf) \equiv \begin{cases} clo(v, cur, slf) & \text{if } v \in \mathbf{FUN} \cup \mathbf{VIW} \cup \mathbf{CLN} \cup \mathbf{NAT} \\ v & \text{otherwise} \end{cases}$$

A.2.3 Argument Binding

$bind^C : \mathbf{FRM} \times \mathbf{INV} \times \mathbf{EXP} \rightarrow \mathbf{VAL}$

$bind^C(frm, nam, exp)$

$\quad \equiv addVar(newFrm(frm), nam, EVAL^C(exp))$

$bind^C(frm, nam(nam'), exp)$

$\quad \equiv addVar(newFrm(frm), nam, close(fun(nam, nam', exp), C_{cur}, C_{ths}))$

A.2.4 Cloning

The cloning operator clones a list of frames from $from$ to to, to not included. The idea is that the original and the clone share the same parent.

$clone : \mathbf{FRM} \times \mathbf{FRM} \rightarrow \mathbf{FRM}$

$clone(from, from) \equiv from$

$clone(from, to) \equiv frm(from_{cst}, copy(from_{var}), clone(from_{nxt}, to))$

$copy : \mathbf{BND} \rightarrow \mathbf{BND}$

$copy(voi) \equiv voi$

$copy(bnd(nam, val, bnd)) \equiv bnd(nam, val, copy(bnd))$

A.3 Evaluation Rules

$EVAL^C : \mathbf{EXP} \rightarrow VAL$

$EVAL^C(slf) \equiv slf$

$EVAL^C(def(ref(nam), exp)) \equiv addVar(C_{cur}, nam, EVAL^C(exp))$

$EVAL^C(def(tbl(nam, idx), exp)) \equiv addVar(C_{cur}, nam, makTab^C(EVAL^C(idx), exp))$
$EVAL^C(def(apl(nam, arg), exp)) \equiv close(addVar(C_{cur}, nam, fun(nam, arg, exp)), C_{ths})$

$EVAL^C(dcl(ref(nam), exp)) \equiv addCst(C_{cur}, nam, EVAL^C(exp))$
$EVAL^C(dcl(tbl(nam, idx), exp)) \equiv addCst(C_{cur}, nam, makTab^C(EVAL^C(idx), exp))$
$EVAL^C(dcl(apl(nam, arg), exp)) \equiv close(addCst(C_{cur}, nam, fun(nam, arg, exp)), C_{ths})$

$EVAL^C(set(ref(nam), exp)) \equiv setVar(C_{cur}, nam, EVAL^C(exp))$
$EVAL^C(set(tbl(nam, idx), exp)) \equiv setTab(getAny(C_{cur}, nam), EVAL^C(idx), EVAL^C(exp))$
$EVAL^C(set(apl(nam, arg), exp)) \equiv close(setVar(C_{cur}, nam, fun(nam, arg, exp)), C_{ths})$

$EVAL^C(ref(nam)) \equiv close(getAny(C_{cur}, nam), C_{cur}, C_{ths})$
$EVAL^C(apl(exp, arg)) \equiv DO^C(EVAL^C(exp), arg)$
$EVAL^C(tbl(exp, idx)) \equiv getTab(EVAL^C(exp), EVAL^C(idx))$

$EVAL^C(msg(exp, inv)) \equiv SEND^C(EVAL^C(exp), inv)$

A.4 Do (= apply) and Send

Application takes a closure and an argument expression. Depending on the type of closure (ordinary method, view method,...) the appropriate action is taken.

$DO^C : \mathbf{CLO} \times \mathbf{EXP} \rightarrow \mathbf{VAL}$

$$DO^C(clo(fun(nam, arg, bdy), cur, ths), exp) \equiv EVAL^{C'}(bdy) \textbf{ where } \left\{ \begin{array}{l} C'_{cur} \equiv bind^C(cur, arg, exp) \\ C'_{ths} \equiv ths \end{array} \right.$$

$$DO^C(clo(viw(nam, arg, bdy), cur, ths), exp) \equiv EVAL^{C'}(bdy) \textbf{ where } \left\{ \begin{array}{l} C'_{cur} \equiv bind^C(cur, arg, exp) \\ C'_{ths} \equiv newFrm(ths) \end{array} \right.$$

$$DO^C(clo(cln(nam, arg, bdy), cur, ths), exp) \equiv EVAL^{C'}(bdy) \textbf{ where}$$
$$\textbf{let} \left(\begin{array}{l} lst = clone(ths, cur_{nxt}) \\ cur' = last(lst) \\ ths' = first(lst) \end{array} \right) \textbf{ in} \left\{ \begin{array}{l} C'_{cur} \equiv bind^C(cur', arg, exp) \\ C'_{ths} \equiv ths' \end{array} \right.$$

$DO^C(clo(nat(\mathbf{super}, \epsilon), cur, ths), exp) \equiv cur_{nxt}$
$DO^C(clo(nat(\mathbf{this}, \epsilon), cur, ths), exp) \equiv ths$

Here is the message passing operator. *getCst* is the delegation operator. It looks in the receiver and takes a third parameter which is "the current self" (in the first call of *getCst* this is the receiver itself of course).

$SEND^C : \mathbf{FRM} \times \mathbf{INV} \rightarrow \mathbf{VAL}$

$SEND^C(obj, ref(nam)) \equiv getCst(obj, nam, obj)$
$SEND^C(obj, fun(nam, arg)) \equiv DO^C(getCst(obj, nam, obj), arg)$
$SEND^C(obj, tbl(nam, exp)) \equiv getTab(getCst(obj, nam, obj), EVAL^C(exp))$

Appendix B

ChitChat Semantics

This appendix presents the semantics of ChitChat. This is not a formal description but rather a highly structured semi-formal treatment that can be used as a guide to implement the language. The meta language we use assumes many constructions that are not accepted by "real" denotational semantics. For example, the concurrency part is covered by a construction $\infty(exp)$ which creates a new thread that executes exp eternally. Synchronization between such "meta threads" is denoted by $\ll exp \gg$. Whenever such an expression occurs on the right hand side of a semantic equation it means that that equation acquires a lock on exp and therefore has to wait if another semantic equation is dealing with the value of exp at that time. Moreover, just as in the semantics of appendix A, we assume the existence of a meta language with assignment.

The semantics of ChitChat is based on the semantics of Pic%. One of the big differences is the distinction between active and passive objects $\mathbf{OBJ} \equiv \mathbf{ACT} \cup \mathbf{PAS}$ and the distribution giving rise to local active objects and remote active objects $\mathbf{ACT} \equiv \mathbf{LAO} \cup \mathbf{RAO}$.

In the semantics we use Greek letters to indicate locations on the network and we subscript data values and functions to indicate clearly at which node the data resides and and which node the function is executing. So $lao_{\pi}(...,...,...,...)$ is a local active object on node π. Futhermore, $getCst_{\pi}(...,...,...)$ is the implementation of $getCst$ on machine π that will be running.

For the sake of simplicity we have chosen to model move methods but leave out visit methods. Furthermore, the same restrictions for ChitChat hold as those for the semantics of appendix A: we model no mixins, no first class continuations, etc. Also call with current promise, the basis of all behavioural synchronization is not treated. Giving a credible semantics to these language features would require us to look at the denotational semantics of concurrent programming languages. This is a hard topic that cannot be relegated to an appendix. Combined with the mobility features, it is probably worth a dissertation by itself.

B.1 Abstract Grammar and Values

Name	**Tagged Value**	**Set**
Basic Values		**SLF**
...	...	...
Promise	pro_π	**PRO**
Expressions		**EXP**
Reference	$ref_\pi(nam)$	$\mathbf{REF} \approx ref(\mathbf{NAM})$
Tabulation	$tbl_\pi(exp, idx)$	$\mathbf{TBL} \approx tbl(\mathbf{EXP}, \mathbf{EXP})$
Application	$apl_\pi(exp, arg)$	$\mathbf{APL} \approx apl(\mathbf{EXP}, \mathbf{EXP})$
Message	$msg_\pi(exp, inv)$	$\mathbf{MSG} \approx msg(\mathbf{EXP}, \mathbf{INV})$
Definition	$def_\pi(inv, exp)$	$\mathbf{DEF} \approx def(\mathbf{INV}, \mathbf{EXP})$
Declaration	$dcl_\pi(inv, exp)$	$\mathbf{DCL} \approx dcl(\mathbf{INV}, \mathbf{EXP})$
Assignment	$ass_\pi(inv, exp)$	$\mathbf{ASS} \approx ass(\mathbf{INV}, \mathbf{EXP})$
Attributes		**ATT**
Function	$fun_\pi(nam, arg, bdy)$	$\mathbf{FUN} \approx fun(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
View	$viw_\pi(nam, arg, bdy)$	$\mathbf{VIW} \approx viw(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
Cloning	$cln_\pi(nam, arg, bdy)$	$\mathbf{CLN} \approx cln(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
Active View	$avw_\pi(nam, arg, bdy)$	$\mathbf{AVW} \approx avw(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
Move Method	$mov_\pi(nam, arg, bdy)$	$\mathbf{MOV} \approx mov(\mathbf{NAM}, \mathbf{EXP}, \mathbf{EXP})$
Native	$nat_\pi(idx)$	**NAT**
Objects		
Pas. Closure	$pcl_\pi(att, cur, ths)$	$\mathbf{PCL} \approx pcl(\mathbf{ATT}, \mathbf{FRM}, \mathbf{PAS})$
Act.. Closure	$acl_\pi(att, cur, ths)$	$\mathbf{ACL} \approx acl(\mathbf{ATT}, \mathbf{ACT}, \mathbf{ACT})$
Binding	$bnd_\pi(nam, val, nxt)$	$\mathbf{BND} \approx bnd(\mathbf{NAM}, \mathbf{VAL}, \mathbf{BND})$
Frame	$frm_\pi(cst, var, nxt)$	$\mathbf{FRM} \approx frm(\mathbf{BND}, \mathbf{BND}, \mathbf{NXT})$
Pas. Obj.	$pas_\pi(cst, var, nxt)$	$\mathbf{PAS} \approx pas(\mathbf{BND}, \mathbf{BND}, \mathbf{NXT})$
Local Act. Obj.	$lao_\pi(pas, que, thr, nxt)$	$\mathbf{LAO} \approx lao(\mathbf{PAS}, \mathbf{que}, \mathbf{thr}, \mathbf{ACT})$
Rem. Act. Obj.	$rao_\pi(idt^{\pi'})$	**RAO**

Remote actual objects are represented as a "wrapper" around an identity that refers to an object on another machine: $rao(idt^{\pi'})$.

CTX	$\equiv$	$ctx(\mathbf{FRM}, \mathbf{PAS}, \mathbf{ACT}, \mathbf{ACT})$	(contexts)
INV	$\equiv$	$\mathbf{REF} \cup \mathbf{TBL} \cup \mathbf{APL}$	(invocations)
EXP	$\equiv$	$\mathbf{INV} \cup \mathbf{MSG} \cup \mathbf{DEF} \cup \mathbf{DCL} \cup \mathbf{ASS}$	(expression)
ATT	$\equiv$	$\mathbf{FUN} \cup \mathbf{VIW} \cup \mathbf{CLN} \cup \mathbf{AVW} \cup \mathbf{MOV} \cup \mathbf{NAT}$	(attributes)
ACT	$\equiv$	$\mathbf{LAO} \cup \mathbf{RAO}$	(active object)
OBJ	$\equiv$	$\mathbf{ACT} \cup \mathbf{PAS}$	(objects)
NXT	$\equiv$	$\mathbf{FRM} \cup \mathbf{PAS}$	
CLO	$\equiv$	$\mathbf{ACL} \cup \mathbf{PCL}$	(closures)
SLF	$\equiv$	$\mathbf{NBR} \cup \mathbf{FRC} \cup \mathbf{TXT} \cup \mathbf{VOI} \cup \mathbf{PRO} \cup \mathbf{TAB}$	(self evaluating)
VAL	$\equiv$	$\mathbf{SLF} \cup \mathbf{CLO} \cup \mathbf{OBJ}$	(values)
AGR	$\equiv$	$\mathbf{VAL} \cup \mathbf{INV} \cup \mathbf{EXP} \cup \mathbf{TAB} \cup$ $\mathbf{ATT} \cup \mathbf{DCT} \cup \mathbf{BND} \cup \mathbf{CTX}$	(abs. gram.)

B.2 Local and Remote Active Objects

An active object at location π has its own passive object *pas*, a queue *que*, an infinite thread *thr* and a next pointer *nxt*. The queue contains 6-tuples consisting of the active current in which the message was found *acu*, the active this to which the message was sent *ats*, the sender of the message *sdr*, the frame holding formals and actuals *frm*, the attribute (i.e., method) that was found *att* and the promise that has to be fulfilled when the attribute is evaluated *pro*. The thread *thr* infinitely serves requests from the queue and fulfills the associated promises by running the attribute appropriately. During evaluation of the method, the passive part is locked so that there is no intra-object concurrency in case another party refers to that part (maybe a local extension).

$$lao_\pi(pas, que = (acu, ats, sdr, frm, att, pro) \oplus que', thr), nxt) \textbf{ where}$$
$$thr \quad = \infty \begin{pmatrix} 1. & C = ctx(frm, \ll pas \gg, acu, ats) \\ 2. & val = RUN_\pi^C(att, sdr) \\ 3. & fulfill(pro, val) \\ 4. & que := que' \end{pmatrix}$$

We assume that $newAct_\pi(pas, nxt)$ creates a new active object with an empty queue and thread. Optionally an existing queue can be passed (notably after moving an existing object) such that $newAct_\pi(pas, que, nxt)$ creates a new active object with an existing queue.

If lao_π is a local active object on machine π, then

$$idt^\pi \quad = \quad getIdt_\pi(lao_\pi)$$

is *the* identity for the object. On different occasions and different times, always the same identity for the object is returned, even though the object might have travelled across network boundaries.

$$\begin{array}{rcl} resolve_\pi(idt^\pi) & = & lao_\pi \\ resolve_\pi(idt^{\pi'}) & = & rao_\pi(idt^{\pi'}) \end{array}$$

The location of an object is absolute:

$$\begin{array}{rcl} loc(lao_\pi) & = & \pi \\ loc(roa_\pi(idt^\delta) & = & \delta \end{array}$$

ENQ_π^C puts a request in the queue of a designated local or remote active object. The first parameter is the object. An active this pointer *ats* is provided, a sender *sdr*, a frame frm containing the formals-actuals bindings and the attribute (i.e., a method) *att* to be evaluated. The frame already contains the evaluated bindings (in the case of normal parameters, and otherwise a reference to the correct dictionary). This is because the queue proceeds asynchronously and the parameters might get used when the dictionary valid at message sending time might already have changed. If the queue is on a remote machine then a network referencing of all the constituents is made to that machine using $\sim_\pi^\delta$. The subscript of this operator is the destination machine. (i.e., from δ to π in the case of $\sim_\pi^\delta$). This $\sim_\pi^\delta$ copies passive parts but uses references for active parts.

$ENQ_\pi : \mathbf{ACT}_\pi \times \mathbf{ACT}_\pi \times \mathbf{ACT}_\pi \times \mathbf{FRM}_\pi \times \mathbf{ATT}_\pi \rightarrow \mathbf{PRO}_\pi$
$ENQ_\pi(lao_\pi(pas, que, thr, nxt), ats, sdr, frm, att)$
$$\equiv \begin{pmatrix} 1. & pro = newPromise() \\ 2. & que := que \oplus (lao_\pi, ats, sdr, frm, att, pro) \\ 3. & pro \end{pmatrix}$$
$ENQ_\pi(rao_\pi(idt^{\pi'}), ats, sdr, frm, att)$
$\equiv\sim_\pi^{\pi'} (ENQ_{\pi'}(resolve_{\pi'}(idt^{\pi'}), \sim_{\pi'}^{\pi} (ats), \sim_{\pi'}^{\pi} (sdr), \sim_{\pi'}^{\pi} (frm), \sim_{\pi'}^{\pi} (bdy)))$

We assume a meta language in which an operator $BECOME$ on active objects exists. Unfortunately this function is not mathematically specifiable without drastically changing the model to manually pass around states. Become replaces all references to one object by another object.

$$BECOME_\pi \quad : \quad \mathbf{ACT}_\pi \times \mathbf{ACT}_\pi \rightarrow \mathbf{ACT}_\pi$$

B.3 Tables

$$\begin{aligned} makTab_\pi^C(nbr, exp) &\equiv tab_\pi[EVAL_\pi^C(exp^1), \ldots, EVAL_\pi^C(exp^{nbr})] \\ getTab_\pi(tab, nbr) &\equiv tab[nbr] \\ setTab_\pi(tab, nbr, val) &\equiv \{tab[nbr] := val; tab\} \end{aligned}$$

B.4 Cloning and Remote Cloning

Just as for Pic%, $clone_\pi$ makes a clone of its first argument which is a list of objects. The second argument is the end of that list. This will not be cloned such that the original list and the clone share the same parent.

$copy_\pi : \mathbf{BND}_\pi \rightarrow \mathbf{BND}_\pi$
$copy_\pi(voi) \equiv voi$
$copy_\pi(bnd_\pi(nam, val, nxt)) \equiv bnd_\pi(nam, val, copy_\pi(nxt))$

$clone_{\pi} : \mathbf{FRM}_{\pi} \times \mathbf{FRM}_{\pi} \rightarrow \mathbf{FRM}_{\pi}$
$\;clone_{\pi}(from, from) \equiv from$
$\;clone_{\pi}(from, to) \equiv frm_{\pi}(from_{cst}, copy_{\pi}(\ll from_{var} \gg), clone_{\pi}(from_{nxt}, to))$
$clone_{\pi} : \mathbf{ACT}_{\pi} \times \mathbf{ACT}_{\delta} \rightarrow \mathbf{ACT}_{\delta}$
$\;clone_{\pi}(from, from) \equiv from$
$\;clone_{\pi}(lao_{\pi}(pas, que, thr, nxt), to) \equiv newAct_{\pi}(clone_{\pi}(pas), clone_{\pi}(nxt, to))$
$\;clone_{\pi}(rao_{\pi}(idt^{\pi'})) \equiv resolve_{\pi}(getIdx_{\pi'}(clone_{\pi'}(resolve_{\pi'}(idt^{\pi'}))))$
$\;copy_{\pi}(voi) \equiv voi$

B.5 Network Referring

This is what happens when values from node π are by reference passed through a wire to node δ. It basically means that active objects are transfered as a remote reference. All the rest is copied. It is assumed that this copy-process is sensitive to cycles in the sense that it correctly copies cyclic graphs.

$\sim^{\pi}_{\delta}: \mathbf{AGR}_{\pi} \rightarrow \mathbf{AGR}_{\delta}$
$\;\sim^{\pi}_{\delta}(slf_{\pi}) \equiv slf_{\delta}$
$\;\sim^{\pi}_{\delta}(ref_{\pi}(nam)) \equiv ref_{\delta}(nam)$
$\;\sim^{\pi}_{\delta}(tbl_{\pi}(exp, idx)) \equiv tbl_{\delta}(\sim^{\pi}_{\delta}(exp), \sim^{\pi}_{\delta}(idx))$
$\;\sim^{\pi}_{\delta}(apl_{\pi}(exp, arg)) \equiv apl_{\delta}(\sim^{\pi}_{\delta}(exp), \sim^{\pi}_{\delta}(arg))$
$\;\sim^{\pi}_{\delta}(msg_{\pi}(exp, inv)) \equiv msg^{\pi}_{\delta}(\sim_{\delta}(exp), \sim^{\pi}_{\delta}(inv))$
$\;\sim^{\pi}_{\delta}(def_{\pi}(inv, exp)) \equiv def_{\delta}(\sim^{\pi}_{\delta}(inv), \sim^{\pi}_{\delta}(exp))$
$\;\sim^{\pi}_{\delta}(dcl_{\pi}(inv, exp)) \equiv dcl_{\delta}(\sim^{\pi}_{\delta}(inv), \sim^{\pi}_{\delta}(exp))$
$\;\sim^{\pi}_{\delta}(ass_{\pi}(inv, exp)) \equiv ass_{\delta}(\sim^{\pi}_{\delta}(inv), \sim^{\pi}_{\delta}(exp))$
$\;\sim^{\pi}_{\delta}(fun_{\pi}(nam, arg, arg)) \equiv fun_{\delta}(nam, \sim^{\pi}_{\delta}(arg), \quad \sim^{\pi}_{\delta}(arg))$
$\;\sim^{\pi}_{\delta}(viw_{\pi}(nam, arg, arg)) \equiv viw_{\delta}(nam, \sim^{\pi}_{\delta}(arg), \quad \sim^{\pi}_{\delta}(arg))$
$\;\sim^{\pi}_{\delta}(cln_{\pi}(nam, arg, arg)) \equiv cln_{\delta}(nam, \sim^{\pi}_{\delta}(arg), \quad \sim^{\pi}_{\delta}(arg))$
$\;\sim^{\pi}_{\delta}(avw_{\pi}(nam, arg, arg)) \equiv avw_{\delta}(nam, \sim^{\pi}_{\delta}(arg), \quad \sim^{\pi}_{\delta}(arg))$
$\;\sim^{\pi}_{\delta}(mov_{\pi}(nam, arg, arg)) \equiv mov_{\delta}(nam, \sim^{\pi}_{\delta}(arg), \quad \sim^{\pi}_{\delta}(arg))$
$\;\sim^{\pi}_{\delta}(nat_{\pi}(idx)) \equiv nat_{\delta}(idx)$
$\;\sim^{\pi}_{\delta}(pcl_{\pi}(att, frm, pas)) \equiv pcl_{\delta}(\sim^{\pi}_{\delta}(att), \sim^{\pi}_{\delta}(frm), \quad \sim^{\pi}_{\delta}(pas))$
$\;\sim^{\pi}_{\delta}(acl_{\pi}(att, aob_1, aob_2)) \equiv acl_{\delta}(\sim^{\pi}_{\delta}(att), \quad \sim^{\pi}_{\delta}(aob_1), \sim^{\pi}_{\delta}(aob_2))$
$\;\sim^{\pi}_{\delta}(bnd_{\pi}(nam, val, nxt)) \equiv bnd_{\delta}(\sim^{\pi}_{\delta}(nam), \quad \sim^{\pi}_{\delta}(val), \sim^{\pi}_{\delta}(nxt))$
$\;\sim^{\pi}_{\delta}(frm_{\pi}(bnd, bnd, frm)) \equiv frm_{\delta}(\sim^{\pi}_{\delta}(bnd), \sim^{\pi}_{\delta}(bnd), \sim^{\pi}_{\delta}(frm))$
$\;\sim^{\pi}_{\delta}((acu, ats, sdr, frm, att, pro) \oplus que) \equiv (\sim^{\pi}_{\delta}(acu), \sim^{\pi}_{\delta}(ats),$
$\qquad \sim^{\pi}_{\delta}(sdr), \sim^{\pi}_{\delta}(frm), \sim^{\pi}_{\delta}(att), \sim^{\pi}_{\delta}(pro))$
$\oplus \sim^{\pi}_{\delta}(que)$
$\;\sim^{\pi}_{\delta}(pas_{\pi}(bnd, bnd, frm)) \equiv \ll pas \gg_{\delta} (\sim^{\pi}_{\delta}(bnd), \quad \sim^{\pi}_{\delta}(bnd), \sim^{\pi}_{\delta}(frm))$
$\;\sim^{\pi}_{\delta}(lao_{\pi}(pas, que, thr, aob)) \equiv resolve_{\delta}(getIdx_{\pi}(lao_{\pi}(pas, que, thr, aob)))$
$\;\sim^{\pi}_{\delta}(rao_{\pi}(idt^{\gamma})) \equiv resolve_{\delta}(idt^{\gamma})$

B.6 Move Semantics

Here is how an object chain starting at machine π and terminating at machine π' is moved to machine δ, the last object not included (this is the super of a move method typically).

$$
\begin{array}{lll}
\approx_\delta^\pi & : & \mathbf{AOB}_\pi \times \mathbf{AOB}_{\pi'} \to \mathbf{AOB}_\delta \\
& & \approx_\pi^\pi (from, from) \\
& & \quad \equiv from \\
& & \approx_\pi^\pi (lao_\pi(pas, que, thr, nxt), to) \\
& & \quad \equiv lao_\pi(pas, que, thr, \approx_\pi^\pi (nxt, to)) \\
& & \approx_\delta^\pi (rao_\pi(idt^\gamma) \\
& & \quad \equiv \approx_\delta^\gamma (resolve_\gamma(idt^\gamma)) \\
& & \approx_\delta^\pi (lao_\pi(pas, que, thr, nxt), to) \\
& & \quad \equiv \left(\begin{array}{ll}
1. & obj_\delta \equiv newAct_\delta(\sim_\delta^\pi (pas), \sim_\delta^\pi (que), \approx_\delta^\pi (nxt, to)) \\
2. & idt^\pi = getIdt_\pi(lao_\pi(pas, que, thr, nxt))) \\
3. & idt^\delta = getIdt_\delta(obj_\delta) \\
4. & \forall \rho \;\mid\; idt^\pi \in resolve_\rho : \\
 & \quad resolve_\rho(idt^\pi).BECOME_\rho(rao_\rho(idt^\delta)) \\
5. & lao_\pi(pas, que, thr, nxt).BECOME_\pi(rao_\pi(idt^\delta))
\end{array} \right)
\end{array}
$$

B.7 Dictionary Structures

$newFrm_\pi : \mathbf{FRM}_\pi \to \mathbf{FRM}_\pi$
$newFrm_\pi(frm) \equiv frm_\pi(voi, voi, frm)$

$addVar_\pi : \mathbf{FRM}_\pi \times \mathbf{NAM} \times \mathbf{VAL}_\pi \to \mathbf{VAL}_\pi$
$addVar_\pi(frm, nam, val)$

$$
\equiv \begin{cases} error & \text{if } getAny_\pi(frm, nam) \neq voi \\ \{frm_{var} := bnd_\pi(nam, val, frm_{var}); val\} & \text{otherwise} \end{cases}
$$

$addCst_\pi : \mathbf{FRM}_\pi \times \mathbf{NAM} \times \mathbf{VAL}_\pi \to \mathbf{VAL}_\pi$
$addCst_\pi(frm, nam, val)$

$$
\equiv \begin{cases} error & \text{if } getAny_\pi(frm, nam) \neq voi \\ \{frm_{cst} := bnd_\pi(nam, val, frm_{cst}); val\} & \text{otherwise} \end{cases}
$$

$setBnd_\pi : \mathbf{BND}_\pi \times \mathbf{NAM} \times \mathbf{VAL}_\pi \to \mathbf{VAL}_\pi$

$$
setBnd_\pi(bnd, nam, val) \equiv \begin{cases} error & \text{if } bnd = voi \\ \{bnd_{val} := val; val\} & \text{if } bnd_{var} = nam \\ \quad setBnd_\pi(bnd_{nxt}, nam, val) & \text{otherwise} \end{cases}
$$

$setVar_\pi : \mathbf{FRM}_\pi \times \mathbf{NAM} \times \mathbf{VAL}_\pi \rightarrow \mathbf{VAL}_\pi$
$setVar_\pi(voi, nam, val) \equiv error$
$setVar_\pi(frm, nam, val) \equiv \textbf{let } v = setBnd_\pi(frm_{var}, nam, val) \textbf{ in}$
$$\begin{cases} setVar_\pi(frm_{nxt}, nam) & \text{if } v = error \\ v & \text{otherwise} \end{cases}$$

There is a difference between frames and objects in ChitChat. Frames can be attached to objects because of the scope rules. Therefore frames delegate the lookup of both constants and variables. Objects never delegate variable access to their super-objects automatically. Hence, objects don't delegate variables.

$getBnd_\pi : \mathbf{BND}_\pi \times \mathbf{NAM} \rightarrow \mathbf{VAL}_\pi$
$getBnd_\pi(voi, nam) \equiv voi$
$$getBnd_\pi(bnd, nam) \equiv \begin{cases} bnd_{val} & \text{if } bnd_{var} = nam \\ getBnd_\pi(bnd_{nxt}, nam) & \text{otherwise} \end{cases}$$

$getAny_\pi : \mathbf{FRM}_\pi \times \mathbf{NAM} \rightarrow \mathbf{VAL}_\pi$
$getAny_\pi(voi, nam) \equiv voi$
$getAny_\pi(frm, nam) \equiv \textbf{let } v = getBnd_\pi(frm_{cst}, nam) \textbf{ in}$
$$\begin{cases} v & \text{if } v \neq voi \\ \textbf{let } v = getBnd_\pi(frm_{var}, nam) \textbf{ in} & \\ \left\{ \begin{array}{ll} v & \text{if } v \neq voi \\ getAny_\pi(frm_{nxt}, nam) & \text{otherwise} \end{array} \right\} & \text{otherwise} \end{cases}$$

$getAny_\pi : \mathbf{PAS}_\pi \times \mathbf{NAM} \rightarrow \mathbf{VAL}_\pi$
$getAny_\pi(voi, nam) \equiv voi$
$getAny_\pi(frm, nam) \equiv \textbf{let } v = getBnd_\pi(frm_{cst}, nam) \textbf{ in}$
$$\begin{cases} v & \text{if } v \neq voi \\ getBnd_\pi(frm_{var}, nam) & \text{otherwise} \end{cases}$$

$getCst_\pi : (\mathbf{FRM}_\pi \cup \mathbf{PAS}_\pi) \times \mathbf{NAM} \times \mathbf{PAS}_\pi \rightarrow \mathbf{VAL}_\pi$
$getCst_\pi(voi, nam, slf)$
$\equiv voi$
$getCst_\pi(frm, nam, slf)$
$\equiv \textbf{let } v = getBnd_\pi(frm_{cst}, nam) \textbf{ in}$
$$\begin{cases} closeP_\pi(v, frm, slf) & \text{if } v \neq voi \\ getCst_\pi(frm_{nxt}, nam, slf) & \text{if } bnd_{var} = nam \end{cases}$$

$getCst_\pi : \mathbf{ACT}_\pi \times \mathbf{NAM} \times \mathbf{ACT}_\pi \rightarrow \mathbf{VAL}_\pi$
$getCst_\pi(voi, nam, slf)$
$\equiv error$
$getCst_\pi(lao_\pi(pob, que, thr, nxt), nam, slf)$
$\equiv \textbf{let } (v, cur, ths) = getCst_\pi(pob, nam, pob) \textbf{ in}$
$$\begin{cases} closeA_\pi(v, lao_\pi(pob, que, thr, nxt), slf) & \text{if } v \neq voi \\ getCst_\pi(nxt, nam, slf) & \text{otherwise} \end{cases}$$
$getCst_\pi(rao_\pi(idt^{\pi'}), nam, slf)$

$\equiv\sim_{\pi}^{\pi'}(getCst_{\pi'}(resolve_{\pi'}(idt^{\pi'})), nam, \sim_{\pi'}^{\pi}(slf)))$

B.8 Evaluation Rules

$EVAL_{\pi}^{C} : \mathbf{EXP}_{\pi} \rightarrow \mathbf{VAL}_{\pi}$
$EVAL_{\pi}^{C}(slf)$
$\equiv slf$

$EVAL_{\pi}^{C}(def(ref(nam), exp))$
$\equiv addVar_{\pi}(C_{cur}, nam, EVAL_{\pi}^{C}(exp))$
$EVAL_{\pi}^{C}(def(tbl(nam, idx), exp))$
$\equiv addVar_{\pi}(C_{cur}(nam, makTab_{\pi}^{C}(EVAL_{\pi}^{C}(idx), exp)))$
$EVAL_{\pi}^{C}(def(apl(nam, arg), exp))$
$\equiv closeP_{\pi}(addVar_{\pi}(C_{cur}, nam, fun(nam, arg, exp)), C_{cur}, C_{ths})$

$EVAL_{\pi}^{C}(dcl(ref(nam), exp))$
$\equiv addCst_{\pi}(C_{cur}, nam, EVAL_{\pi}^{C}(exp))$
$EVAL_{\pi}^{C}(dcl(tbl(nam, idx), exp))$
$\equiv addCst_{\pi}(C_{cur}(nam, makTab_{\pi}^{C}(EVAL_{\pi}^{C}(idx), exp)))$
$EVAL_{\pi}^{C}(dcl(apl(nam, arg), exp))$
$\equiv closeP_{\pi}(addCst_{\pi}(C_{cur}, nam, fun(nam, arg, exp))C_{cur}, C_{ths})$

$EVAL_{\pi}^{C}(set(ref(nam), exp))$
$\equiv setVar_{\pi}(C_{cur}, nam, EVAL_{\pi}^{C}(exp))$
$EVAL_{\pi}^{C}(set(tbl(nam, idx), exp))$
$\equiv setTab_{\pi}(getAny_{\pi}(C_{cur}, nam), EVAL_{\pi}^{C}(idx), EVAL_{\pi}^{C}(exp))$
$EVAL_{\pi}^{C}(set(apl(nam, arg), exp))$
$\equiv closeP_{\pi}(setVar_{\pi}(C_{cur}, nam, fun(nam, arg, exp))C_{cur}, C_{ths})$

$EVAL_{\pi}^{C}(ref(nam)$
$\equiv closeP_{\pi}(getAny_{\pi}(C_{cur}, nam), C_{cur}, C_{ths})$
$EVAL_{\pi}^{C}(apl(exp, arg))$
$\equiv DO_{\pi}^{C}(EVAL_{\pi}^{C}(exp), arg)$
$EVAL_{\pi}^{C}(tbl(exp, idx))$
$\equiv getTab_{\pi}(EVAL_{\pi}^{C}(exp), EVAL_{\pi}^{C}(idx))$

$EVAL_{\pi}^{C}(msg(exp, inv))$
$\equiv SEND_{\pi}^{C}(EVAL_{\pi}^{C}(exp), inv)$

B.9 Closures and Bindings

$bind^C_\pi : \mathbf{FRM}_\pi \times \mathbf{INV}_\pi \times \mathbf{EXP}_\pi \rightarrow \mathbf{VAL}_\pi$

$$\begin{aligned}
&bind^C_\pi(frm, nam, exp)\\
&\quad \equiv addVar_\pi(newFrm_\pi(frm), nam, EVAL^C_\pi(exp))\\
&bind^C_\pi(frm, nam(nam'), exp)\\
&\quad \equiv addVar_\pi(newFrm_\pi(frm), nam, pcl_\pi(fun(nam, nam', exp), C_{cur}, C_{ths}))
\end{aligned}$$

There are two types of closures: active closures and passive closures.

$closeP_\pi : \mathbf{AGR}_\pi \times \mathbf{FRM}_\pi \times \mathbf{PAS}_\pi \rightarrow \mathbf{AGR}_\pi$

$$closeP_\pi(v, cur, slf) \equiv \begin{cases} pcl_\pi(v, cur, slf) & \text{if } v \in \mathbf{ATT}_\pi \\ v & \text{otherwise} \end{cases}$$

$closeA_\pi : \mathbf{AGR}_\pi \times \mathbf{ACT}_\pi \times \mathbf{ACT}_\pi \rightarrow \mathbf{AGR}_\pi$

$$closeA_\pi(v, cur, slf) \equiv \begin{cases} acl_\pi(v, cur, slf) & \text{if } v \in \mathbf{ATT}_\pi \\ v & \text{otherwise} \end{cases}$$

B.10 Apply and Send

B.10.1 Message Sending

$SEND^C_\pi : \mathbf{OBJ}_\pi \times \mathbf{INV}_\pi \rightarrow \mathbf{VAL}_\pi$

$$\begin{aligned}
&SEND^C_\pi(obj, ref(nam)) \equiv getCst^C_\pi(obj, nam, obj)\\
&SEND^C_\pi(obj, fun(nam, arg)) \equiv DO^C_\pi(getCst^C_\pi(obj, nam, obj), arg)\\
&SEND^C_\pi(obj, tbl(nam, exp)) \equiv getTab_\pi(getCst^C_\pi(obj, nam), EVAL^C_\pi(exp))
\end{aligned}$$

B.10.2 Passive Closure Invocation

$DO^C_\pi : \mathbf{CLO} \times \mathbf{EXP} \rightarrow \mathbf{VAL}$

$$\begin{aligned}
&DO^C_\pi(pcl_\pi(fun_\pi(nam, arg, bdy), cur, ths), exp)\\
&\quad \equiv EVAL^{C'}_\pi(bdy) \textbf{ where} \begin{cases} C'_{cur} = bind^C_\pi(\ll cur \gg, arg, exp) \\ C'_{ths} = ths \\ C'_{acu} = C_{acu} \\ C'_{ats} = C_{ats} \end{cases}\\
&DO^C_\pi(pcl_\pi(viw_\pi(nam, arg, bdy), cur, ths), exp)\\
&\quad \equiv \begin{pmatrix} 1 & EVAL^{C'}_\pi(bdy) \\ 2 & C'_{cur} \end{pmatrix} \textbf{ where} \begin{cases} C'_{cur} = bind^C_\pi(\ll cur \gg, arg, exp) \\ C'_{ths} = ths \\ C'_{acu} = C_{acu} \\ C'_{ats} = C_{ats} \end{cases}
\end{aligned}$$

$DO^C_\pi(pcl_\pi(cln_\pi(nam, arg, bdy), cur, ths), exp)$

$$\equiv EVAL^{C'}_\pi(bdy) \textbf{ where let } \begin{pmatrix} lst = clone_\pi(ths, cur_{nxt}) \\ cur' = last(lst) \\ ths' = first(lst) \end{pmatrix}$$

$$\textbf{in} \begin{cases} C'_{cur} = bind^C_\pi(\ll cur' \gg, arg, exp) \\ C'_{ths} = ths' \\ C'_{acu} = C_{acu} \\ C'_{ats} = C_{ats} \end{cases}$$

$DO^C_\pi(pcl_\pi(avw_\pi(nam, arg, bdy), cur, ths), exp)$

$$\equiv EVAL^{C'}_\pi(bdy) \textbf{ where } \begin{cases} C'_{cur} = bind^C_\pi(\ll cur \gg, arg, exp) \\ C'_{ths} = C'_{cur} \\ C'_{acu} = newAct_\pi(C'_{ths}, C_{ats}) \\ C'_{ats} = C'_{acu} \end{cases}$$

$DO^C_\pi(pcl_\pi(mov_\pi(nam, arg, bdy), cur, ths), exp)$
$\equiv error$

B.10.3 Active Closure Invocation (Pre-Queue)

$DO^C_\pi : \mathbf{CLO} \times \mathbf{EXP} \to \mathbf{VAL}$

$DO^C_\pi(acl_\pi(fun_\pi(nam, arg, bdy), acu, ats), exp)$

$$\equiv pro \textbf{ where} \begin{cases} frm = bind^C_\pi(\epsilon, arg, exp) \\ pro = ENQ_\pi(acu, ats, C_{acu}, frm, fun_\pi(nam, arg, bdy)) \end{cases}$$

$DO^C_\pi(acl_\pi(cln_\pi(nam, arg, bdy), acu, ats), exp)$

$$\equiv pro \textbf{ where} \begin{cases} frm = bind^C_\pi(\epsilon, arg, exp) \\ pro = ENQ_\pi(acu, ats, C_{acu}, frm, cln_\pi(nam, arg, bdy)) \end{cases}$$

$DO^C_\pi(acl_\pi(viw_\pi(nam, arg, bdy), acu, ats), exp)$
$\equiv error$

$DO^C_\pi(acl_\pi(avw_\pi(nam, arg, bdy), acu, ats), exp)$

$$\equiv EVAL^{C'}_\pi(bdy) \textbf{ where } \begin{cases} C'_{cur} = bind^C_\pi(\epsilon, arg, exp) \\ C'_{ths} = C'_{cur} \\ C'_{acu} = newAct_\pi(C'_{ths}, C_{ats}) \\ C'_{ats} = C'_{acu} \end{cases}$$

$DO^C_\pi(acl_\pi(mov_\pi(nam, arg, bdy), acu, ats), exp)$

$$\equiv pro \begin{cases} frm = bind^C_\pi(\epsilon, arg, exp) \\ pro = ENQ_\pi(acu, ats, C_{acu}, frm, mov_\pi(nam, arg, bdy)) \end{cases}$$

B.10.4 Active Method Running (Post-Queue-Processing)

For some attributes such as active views, the interesting part happens before or without scheduling a request in the queue of the corresponding active receiver. However, for ordinary methods, cloning methods and move methods, the logic of the method is executed after serving the method from the queue. This will invoke RUN_π^C.

RUN_π^C : **ATT** × **ACT** → **VAL**

$$RUN_\pi^C(fun_\pi(nam, arg, bdy), sdr) \equiv \begin{pmatrix} 1. & C_{cur_{nxt}} := C_{acu_{pas}} \\ 2. & EVAL_\pi^C(bdy) \end{pmatrix}$$

$$RUN_\pi^C(cln_\pi(nam, arg, bdy), sdr) \equiv \begin{pmatrix} 1. & lst = copy_{loc(C_{ats})}(C_{ats}, C_{acu_{nxt}}) \\ 2. & C'_{acu} = last(lst) \\ 3. & C'_{ats} = first(lst) \\ 4. & C_{cur_{nxt}} := C'_{acu_{pas}} \\ 5. & C'_{cur} = clone_\pi(C_{cur}, voi) \\ 6. & C'_{ths} = clone_\pi(C_{ths}, voi) \\ 7. & EVAL_\pi^{C'}(bdy) \end{pmatrix}$$

$$RUN_\pi^C(mov_\pi(nam, arg, bdy), sdr) \equiv \begin{pmatrix} 1. & \Delta = allLocationsFromTo(C_{ats}, C_{acu}) \\ 2. & \delta = loc(sdr) \\ 3 & \begin{cases} voi & \text{if } \Delta = \{\delta\} \\ \begin{pmatrix} 1. & lst = \approx_{loc(sdr)}^{loc(C_{ats})} (\ll (C_{ats}, C_{acu_{nxt}}) \gg) \\ 2. & C'_{acu} = last(lst) \\ 3. & C'_{ats} = first(lst) \\ 4. & C_{cur_{nxt}} := C'_{acu_{pas}} \\ 5. & C'_{cur} = C_{cur} \\ 6. & C'_{ths} = C_{ths} \\ 7. & EVAL_\delta^{C'}(bdy) \end{pmatrix} & \text{otherwise} \end{cases} \end{pmatrix}$$

$$RUN_\pi^C(nat_\pi(\mathbf{asuper}, arg), sdr) \equiv EVAL_\pi^C(arg)$$

$$RUN_\pi^C(nat_\pi(\mathbf{aths}, arg), sdr) \equiv EVAL_\pi^C(arg)$$

B.10.5 natives Invocation

DO_π^C : **CLO** × **EXP** → **VAL**

$$DO_\pi^C(acl_\pi(nat_\pi(\mathbf{asuper}, \epsilon), acu, ats), exp) \equiv acu_{nxt}$$

$DO^C_\pi(acl_\pi(nat_\pi(\mathbf{asuper}, arg), acu, ats), exp)$
$\quad \equiv ENQ_\pi(acu_{nxt}, ats, ats, acu_{nxt_{pas}}, nat_\pi(\mathbf{asuper}, exp)$
$DO^C_\pi(acl_\pi(nat_\pi(\mathbf{super}, \epsilon), acu, ats), exp)$
$\quad \equiv acu_{pas_{nextobj}}$
$DO^C_\pi(acl_\pi(nat_\pi(\mathbf{this}, \epsilon), acu, ats), exp)$
$\quad \equiv acu_{pas}$
$DO^C_\pi(acl_\pi(nat_\pi(\mathbf{athis}, \epsilon), acu, ats), exp)$
$\quad \equiv ats$
$DO^C_\pi(acl_\pi(nat_\pi(\mathbf{athis}, arg), acu, ats), exp)$
$\quad \equiv ENQ_\pi(ats, ats, acu, nat_\pi(\mathbf{asuper}, exp)$

Index

Bibliography

[ACFG01] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 108–124. ACM Press, 2001.

[Agh86] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[AL98] Y. Aridor and D. B. Lange. Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents (Agents'98) in Minneapolis/St. Paul*, 1998.

[ARS97] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[AS85] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BGL98] JP. Briot, R. Guerraoui, and KP. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[BHJL86] A. Black, N. Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86. ACM Press, 1986.

[BHP03] S. Bouchenak, D. Hagimont, and N. De Palma. Efficient Java thread serialization. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 35–39. Computer Science Press, Inc., 2003.

[BL92] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer languages*, 1992.

[Bla85] A.P. Black. The Eden programming language. Technical Report 85-09-01, Dept. of Computer Science, University of Washington, 1985.

[Bla94] G. Blashek. *Object-Oriented Programming with Prototypes.* Springer-Verlag, 1994.

[Bla04] A. P. Black. Post-Javaism. *IEEE Internet Computing*, pages 93–96, 2004.

[BN02] L. Bettini and R. De Nicola. Translating strong mobility into weak mobility. In *Proceedings of the 5th International Conference on Mobile Agents*, pages 182–197. Springer-Verlag, 2002.

[BY87] J.P. Briot and A. Yonezawa. Inheritance and Synchronization in Concurrent OOP. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the ECOOP '87 European Conference on Object-oriented Programming*, pages 32–40, Paris, France, 1987. Springer Verlag.

[Car95] L. Cardelli. Obliq: A language with distributed scope. In *Proc. of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297, 1995.

[CDDS94] W. Codenie, K. De Hondt, T. D'Hondt, and P. Steyaert. Agora: Message passing as a foundation for exploring OO language concepts. *SIGPLAN Notices*, 29(12):48–57, 1994.

[CGPV96] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. A Characterization of Mobility and State Distribution in Mobile Code Languages. In M. Mühlaüser, editor, *Proceedings of the First Workshop on Mobile Object Systems, Special Issues in Object-Oriented Programming: Workshop Reader of the 10^{th} European Conference on Object-Oriented Programming (ECOOP '96)*, pages 309–318, Linz, Austria, July 1996. dpunkt.

[Cla88] K. L. Clark. PARLOG and its applications. *IEEE Trans. Softw. Eng.*, 14(12):1792–1804, 1988.

[Coo89] W. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 433–443. ACM Press, 1989.

[CR93] D. Caromel and M. Rebuffel. Object based concurrency: Ten language features to achieve reuse. In R. Ege, M. Singh, and B. Meyer, editors, *Proceedings of TOOLS-USA'93, Santa Barbara, (CA), USA*, pages 205–214. Prentice-Hall, Englewood Cliffs (NJ), USA, 1993.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160. ACM Press, 1989.

[DCD03] J. Dedecker, T. Cleenewerck, and W. De Meuter. Distributed object inheritance to structure distributed applications. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, 2003.

[DD02] J. Dedecker and W. De Meuter. Using the prototype-based programming paradigm for structuring mobile applications. OOPSLA02 Workshop on Agent-based Methodologies, 2002.

[DD03a] J. Dedecker and W. De Meuter. Communication abstractions through new language concepts. In *Workshop on Communication Abstractions for Distributed Systems, Ecoop 2003*, 2003.

[DD03b] T. D'Hondt and W. De Meuter. On first-class methods and dynamic scope. *Proceedings of LMO*, pages 137–149, 2003.

[DDD03a] W. De Meuter, J. Dedecker, and T. D'Hondt. Wild abstraction ideas for highly dynamic software. In *Workshop on Object-oriented Language Engineering for the Post-Java Era, Ecoop 2003*, 2003.

[DDD03b] W. De Meuter, T. D'Hondt, and J. Dedecker. Intersecting classes and prototypes. In *Proceedings of PSI-Conference, Novosibirsk, Russia*. Springer-Verlag, 2003.

[DDD04] W. De Meuter, T. D'Hondt, and J. Dedecker. Pico: Scheme for mere mortals. *Proceedings of the first international Lisp Workshop*, 2004.

[DDGD01] W. De Meuter, M. D'Hondt, S. Goderis, and T. D'Hondt. Reasoning with design knowledge for interactively supporting framework reuse. In *Proceedings of the SCASE'01 (Soft Computing applied to Software Engineering)*, 2001.

[DDV03] J. Dedecker, W. De Meuter, and W. Van Belle. Actors for pervasive computing. In *OOPSLA Workshop on Reference Architectures and Patterns for Pervasive Computing*, 2003.

[DDW99] M. D'Hondt, W. De Meuter, and R. Wuyts. Using reflective programming to describe domain knowledge as an aspect. In *Proceedings of GCSE '99*, 1999.

[De 97] W. De Meuter. Monads as a theoretical foundation for AOP. Proceedings of the International Workshop on Aspect-Oriented Programming at ECOOP, 1997. 25, 1997.

[De 98a] W. De Meuter. Agora: The story of the simplest mop in the world – or – the Scheme of object orientation. In A. Taivalsaari J. Noble, I. Moore, editor, *Prototype-based Programming*. Springer-Verlag, 1998.

[De 98b] W. De Meuter. Agora98 language manual. http://prog.vub.ac.be/research/agora/, 1998.

[Dek04] S. Dekorte. Io, a small programming language. http://www.iolanguage.com/, 2004.

[DMC92] C. Dony, J. Malenfant, and P. Cointe. Prototype-based languages: from a new taxonomy to constructive proposals and their validation. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 201–217. ACM Press, 1992.

[DMS96] W. De Meuter, T. Mens, and P. Steyaert. Agora: reintroducing safety in prototype-based languages. ECOOP96 Workshop on Prototype-Based Languages, 1996.

[DV04] J. Dedecker and W. Van Belle. Actors for mobile ad-hoc networks. In L.T. Yang, M. Guo, G.R. Gao, and N.K. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*. Springer, 2004. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

[EAC98] S.O. Ehmety, I. Attali, and D. Caromel. About the automatic continuations in the Eiffel// model. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 219–225, 1998.

[FM99] S. Fünfrocken and F. Mattern. Mobile agents as an architectural concept for internet-based distributed applications - the wasp project approach. In Steinmetz, editor, *Proc. KiVS'99*, pages 32–43, Springer-Verlag, 1999.

[Fow97] M. Fowler. Dealing with roles. In *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs*, pages 65–71, 1997.

[FPV98] A. Fuggetta, GP. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[FSB$^+$99] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Jo&#227;o Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. Perdis: Design, implementation, and use of a persistent distributed store. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 427–452. Springer-Verlag, 1999.

[Gab] R. P. Gabriel. Worse is better paper series. http://www.dreamsongs.com/WorseIsBetter.html.

[Gab94] R. P. Gabriel. Lisp: Good News, Bad News, How to Win Big. http://www.ai.mit.edu/articles/good-news/good-news.html, January 1994.

[GBD02] S. Goderis, J. Brichau, and W. De Meuter. A case in multiparadigm programming: User interfaces by means of declarative meta programming. ECOOP2002 Workshop on Multiparadigm Programming with Object-Oriented Languages, 2002.

[GBO$^+$98] T. R.G. Green, A. Borning, T. O'Shea, M. Minoughan, and R. B. Smith. The stripetalk papers: Understandability as a language design issue in object-oriented programming systems. In A. Taivalsaari J. Noble, I. Moore, editor, *Prototype-based Programming.* Springer-Verlag, 1998.

[GCK$^+$02] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus. D'Agents: Applications and performance of a mobile-agent system. *Software— Practice and Experience*, 32(6):543–573, May 2002.

[GD02] S. Goderis and W. De Meuter. Generating user interfaces by means of declarative meta-programming. ECOOP02 Workshop on Generative Programming, 2002.

[GF99] R. Guerraoui and M. E. Fayad. OO distributed programming is *Not* distributed OO programming. *Communications of the ACM*, 42(4):101–104, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GM95] Inc. General Magic. Telescript language reference, 1995.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[GR01] M. J. Guzdial and K. M. Rose. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall PTR, 2001.

[Hal85] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[Hen91] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, page 548568. Springer, 1991.

[Hoa73] C. A. R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford University, 1973.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HRB$^+$91] N. C. Hutchinson, R. K. Raj, A. P. Black, H. M. Levy, and E. Jul. The Emerald programming language. Technical report, Dept. of Computer Science, University of British Columbia, Vancouver, Canada, october 1991.

[IST03] ISTAG. Ambient intelligence: from vision to reality. http://www.cordis.lu/ist/istag.htm, September 2003. Draft consolidated report.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Kam88] S. Kamin. Inheritance in smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–87. ACM Press, 1988.

[KB02] A. Kaminsky and H-P. Bischof. Many-to-many invocation: a new object-oriented paradigm for ad hoc collaborative systems. In *Proceedings of the OOPSLA2002 Onward! Track*, july 2002.

[KCE98] R. Kelsey, W. Clinger, and J. Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[Lea99] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, November 1999. Online Supplement at http://gee.cs.oswego.edu/dl/cpj.

[Lie86] H. 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] H. Lieberman. Concurrent object-oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[Lis88] B. Liskov. Distributed programming in argus. *Communications Of The ACM*, 31(3):300–312, 1988.

[LO98] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[LS94] C. Lucas and P. Steyaert. Modular inheritance of objects through mixin-methods. In *Proceedings of the 1994 Joint Modular Languages Conference*, pages 273–282, 1994.

[LSU87] H. Lieberman, L. Stein, and D. Ungar. Treaty of orlando. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 43–44. ACM Press, 1987.

[M. 03] M. Wolczko and R. B. Smith. Prototype-based application construction using Self 4.0, 2003.

[Mac87] B. J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Holt, Rinehart and Winston, second edition, 1987.

[McA95] J. McAffer. Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214. Springer-Verlag, 1995.

[MD03] P. Mougin and S. Ducasse. Oopal: integrating array programming in object-oriented programming. *SIGPLAN Not.*, 38(11):65–77, 2003.

[MMF01] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. *Lecture Notes in Computer Science*, 1962:349–??, 2001.

[MY93] S. Matsuoka and A. 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.

[Nor98] P. Norvig. Design patterns in dynamic programming. http://norvig.com/design-patterns/, 1998.

[NTM98] J. Noble, A. Taivalsaari, and I. Moore. *Prototype-Based Programming: Concepts, Languages and Applications.* Springer, 1998.

[PH99] M. Philippsen and B. Haumacher. More efficient object serialization. In *IPPS/SPDP Workshops*, pages 718–732, 1999.

[PS97] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.

[PSH04] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for Java futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages and Systems (OOPSLA)*, October 2004. To appear.

[RTL$^+$91] R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91–118, 1991.

[SCD$^+$93] P. Steyaert, W. Codenie, T. D'hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested mixin-methods in Agora. *Lecture Notes in Computer Science*, 707:197–??, 1993.

[Sch86] David A Schmidt. *Denotational Semantics A Methodology for Language Development.* Wm. C. Brown, Dubuque, Iowa, USA, 1986.

[Sch03] K. Schougaard. Language support for distributed computation, 2003.

[SD95] P. Steyaert and W. De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, volume 952 of *Lecture Notes in Computer Science*, pages 127–144. Springer, August 1995.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languagges and Applications*, pages 268–285. ACM Press, October 1996.

[Smi98] W. R. Smith. Newtonscript: Prototypes on the palm. In A. Taivalsaari J. Noble, I. Moore, editor, *Prototype-based Programming*. Springer-Verlag, 1998.

[SMY99] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, pages 211–226, 1999.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45. ACM Press, 1986.

[Ste87] L. Stein. Delegation is inheritance. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 138–146. ACM Press, 1987.

[Ste94] P. Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, 1994.

[SU98] R. B. Smith and D. Ungar. Programming as an experience: The inspiration for Self. In A. Taivalsaari J. Noble, I. Moore, editor, *Prototype-based Programming.* Springer-Verlag, 1998.

[Sue00] T. Suezawa. Persistent execution state of a Java virtual machine. In *ACM Java Grande 2000 Conference*, 2000.

[Sun03] Sun Microsystems Inc. Project JXTA, 2003. http://www.jxta.org/.

[SV97] Springer-Verlag, editor. *Security and Communication in Mobile Object Systems*, number 1222 in LNCS, 1997.

[Tai93] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming.* PhD thesis, University of Jyvaskyla, 1993.

[Tai98] A. Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. In A. Taivalsaari J. Noble, I. Moore, editor, *Prototype-based Programming.* Springer-Verlag, 1998.

[Tho97] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, 1997.

[TK02] R. Tolksdorf and K. Knubben. Programming distributed systems with the delegation-based object-oriented language dself. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 927–931. ACM Press, 2002.

[TOH99] Y. Tahara, A. Ohsuga, and S. Honiden. Agent system development method based on agent patterns. In *Proceedings of the 21st international conference on Software engineering*, pages 356–367. IEEE Computer Society Press, 1999.

[TRV+00a] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in Java. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 29–43. Springer-Verlag, 2000.

[TRV+00b] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in Java. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 29–43. Springer-Verlag, 2000.

[UCCH91] D. Ungar, C. Chambers, B. Chang, and U. Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, 1991.

[US87] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

[VD00] W. Van Belle and Theo D'Hondt. Agent mobility and reification of computational state, an experiment in migration. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, number 1887 in Lecture Notes in Artifical Intelligence. Springer Verlag, June 2000.

[VDD94] E. Van Paesschen, W. De Meuter, and T. D'Hondt. Domain modeling in Self yields warped hierarchies. In *Proceedings of the 3rd International Workshop on Mechanisms for Specialization, Generalization and Inheritance*, pages 65–71, 1994.

[VF01] W. Van Belle and J. Fabry. Experiences in mobile computing: The CBorg mobile multi agent system. In *Proceedings of Tools Europe2001*, March 2001.

[Vig04] G. Vigna. Mobile Agents: Ten Reasons For Failure. In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM '04)*, pages 298–299, Berkeley, CA, January 2004. Position Paper.

[VM04] T. Van Cutsem and S. Mostinckx. A prototype-based approach to distributed applications. Master's thesis, Vrije Universiteit Brussel, 2004.

[VMD$^+$04] T. Van Cutsem, S. Mostinckx, W. De Meuter, J. Dedecker, and T. D'Hondt. On the performance of SOAP in a non-trivial peer-to-peer experiment. In *Proceedings of the 2nd International Working Conference on Component Deployment*, Lecture Notes In Computer Science. Springer Verlag, May 2004.

[vol] Volano chat. http://www.volano.com/.

[voy] Voyager. http://www.recursionsw.com/.

[Wal] Waldemar. Javascript 2.0. http://www.mozilla.org/js/language/js20/.

[WF88] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988. Also to appear in Lisp and Symbolic Computation.

[Whi96] J.E. White. Telescript technology: Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.

[Wik04] Wikipedia. Personal Area Network. http://en.wikipedia.org/wiki/`Personal_area_network`, 2004.

[Wol88] M. I. Wolczko. *Semantics of Object-Oriented Languages*. PhD thesis, University of Manchester, 1988.

[YBS86] A. Yonezawa, J.P. Briot, and E. 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.