— vordenker-archive —

Rudolf Kaehr

(1942-2016)

Title

Actors, Objects, Contextures, Morphograms

Readings and Reflections about Hierarchy and Heterarchy in Programming Paradigms

Archive-Number / Categories

2_22 / K07

Publication Date

2006

Keywords

Actors, Objects, Morphograms, Paradigm, Programming, Hewitt, Agha, Wegner

Disciplines

Cybernetics, Artificial Intelligence and Robotics, Systems Architecture and Theory and Algorithms

Abstract

Systematic and historic overview and critics of actor and object oriented programming.
Readings and Reflections about Hierarchy and Heterarchy in Programming Paradigms: Actors, Objects, Contextures, Morphograms - Leibniz' Monadology - The Scientific Community Metaphor for Actors - The Society Metaphor and Addressability - Peter Wegners : Everything is interaction - Niklas Luhmann: Everything is communication - Beyond communication

Citation Information / How to cite

Rudolf Kaehr: "Actors, Objects, Contextures, Morphograms", www.vordenker.de (Sommer Edition, 2017) J. Paul (Ed.), URL: http://www.vordenker.de/rk/rk_Actors-Objects-Contextures-Morphograms_2006.pdf

Actors, Objects, Contextures, Morphograms

– Ultra-DRAFT –

Rudolf Kaehr

ThinkArt Lab Glasgow 2006

http://www.thinkartlab.com

Actors, Objects, Contextures, Morphograms

Readings and Reflections about Hierarchy and Heterarchy in Programming Paradigms

Actors, Objects, Contextures, Morphograms

Readings and Reflections about Hierarchy and Heterarchy in Programming Paradigms

1 Actors, Objects, Contextures, Morphograms

What's about?

Zeitgeist:
Monadology (Leibniz),
Category Theory (Lawvere, Goguen)
Actor Model (Hewitt / Agha),
Object Model (Alan Kay, Smalltalk),
Autopoiesis Theory (Maturana / Varela),
System Theory (Luhmann)
Interactive Computing (Wegner / Goldin)
Polycontexturality (Gunther / Kaehr)

Problems:
Hierarchy / heterarchy in computation and programming,
Infinite regress, antinomy, non-well founded set theory, paraconsistency, mediation, togetherness
local / global, observer theory, open systems
inheritance / delegation / multiple inheritance
autonomy / autarky / persistency

Hidden strategies:
distinction of primitive actors / actors / meta-actors despite the denial of distinctions
hiding the "slaves" of liberalism

Style:
prospective reconstruction of readings and constructions,
Internet based research
Original papers from the beginnings

1.1 Hierarchy and heterarchy

The text *"Actors and Objects"* can be read as a historic and systematic part of the paper *"From Ruby to Rudy"* concerning the topics of hierarchic and heterarchic organization of actor- and object-based programming languages.

As in other papers, the focus of my thematizations and conceptualizations is on the interplay of hierarchic and heterarchic organizational structures and dynamics. The hierarchic approach is well known, but the problems it create are less known and seldom presented. Mainly because of a lack of an alternative or complementary concept for organizational structures.

In *"From Ruby to Rudy"* I have given a complex approach to objects and aspects, involving new concepts like *abjects*, *injects* and *projects*. But this has been done in a highly abbreviated form.

It is not common that computer scientist are studying the history of their topics. In this sense I feel free to use citations wherever I find them to elude the structure, conflicts and paradoxes of Actor and Object based programming.

On the other hand, this study is a kind of a recapitulation and a trial to present my own experiences with the conceptual promises and models of Actor and Object oriented programming paradigms during a history of well 20 years.

Nevertheless, the presentation will not have the maturity of a monographic study, simply because there is no funding for it and also no pressure to do it.

Today, I have to apologize to not to continue this work anymore!!!

The Open System approach of the Actor Model of programming started with a strong rejection of a pre-established hierarchic structure. But failed to give a positive answer to its rejection. Object-oriented programming had no other chance to be successful than to implement hierarchic forms of organization. Later, the Actor Model had to follow this decision to be successful (Agha). Thus, the liberation from hierarchy is still a promise and in no way realized.

According to Paul Feyerabend, we have to make a strict distinction between text book knowledge presentation (for students and other believers) and the original research papers of the people who invented the stuff. Without doubt, they too had to make some propaganda (cf. Galilei) but with the aim to promote their research and not their propaganda.

One propaganda is insisting on the paradigm change in programming by the Actor and the Object paradigm. Also insisting, that there are no problems for say multi-inheritance, interaction and reflection, etc.

The other propaganda tells us, that all is fine and save under the established concept of programming, based on the lambda calculus, and that their programming language is covering all know and surely also the not yet know styles of programming without producing any inherent problems to the underlying paradigm of programming.

Obviously, we have to distinguish between a more theoretical approach to programming paradigms and a more practical oriented approach which has other concerns than the theoretical approach.

1.2 Where has the paradigm shift gone?

After I learnt how easy it is to implement object-oriented and object-based programming in the framework of lambda calculus-based languages I try to emphasize now, again, the *incommensurabilities* of the proclaimed paradigm shift to classic approaches in programming (Hewitt, Kay) introduced by the original ideas of the Actor model of computation (Hewitt, Agha) of interactive computation (Wegner, Goldin, Milner) and on the other side of the new paradigm of OOP.

I should not forget that I have written an eBook in German about these topics not long ago, *"Strukturationen der Interaktivität"*. With the magic second title: *"Skizze eines Gewebes rechnender Räume in denkender Leere."* This book is focused mainly on the interactional models of computation, in the sense of *"the interactional revolution in computation"* (Wegner) and on new polycontextural models of trans-computation, but not much on the corresponding programming paradigms, like the Actor Model and object-oriented programming.

http://www.thinkartlab.com/pkl/media/SKIZZE-0.9.5-medium.pdf

1.3 OOP as a style of programming

OOP as a *style* of programming in a typical propaganda fashion.

> **What is Caml?**
>
> Caml is a general-purpose programming language, designed with program safety and reliability in mind. It is very expressive, yet easy to learn and use. Caml supports functional, imperative, and object-oriented programming styles.
>
> **Objective Caml**
>
> The Objective Caml system is the main implementation of the Caml language. It features a powerful module system and a full-fledged *object-oriented layer*. It comes with a native-code compiler that supports numerous architectures, for high performance; a bytecode compiler, for increased portability; and an interactive loop, for experimentation and rapid development.
>
> http://caml.inria.fr/

Functional programming is well embedded in the mathematics of the Lambda Calculus. The Lambda Calculus is both, a paradigm of programming and a model of computation. It is even a model of (constructive) mathematics itself.

Object-oriented programming is not (yet) based on a commonly accepted mathematical model. Nevertheless, it is presented and propagated as based on the Lambda Calculus.

On the other hand, different calculi had been developed to model and formalize the OOP paradigm. A first classification suggests, that functional programming is *algebraic*, while the interactional aspects of object-oriented programming are best understood as *co-algebraic*. Algebra and coalgebra are both well founded in 1-category theory. In contrast to the new "movement" of n-category theory 1-categories are well known, simply as categories without n for dimensions.

To add to the confusions, let's mention the idea the other way round: *Mathematics is object-oriented.* Proof: Math has abstract/concrete, hierarchy/inheritance, polymorphism, encapsulation. So, where is the problem?

http://www.cut-the-knot.org/Mset99/math.shtml

2 Inheritance vs. delegation and composition

"In classic inheritance, as found in Java, Smalltalk, and the single-inheritance subset of C++ (***what about Perl & Python?), an object is an instance of a class, while a class is a subclass of a parent class, which in turn is a subclass... on up to a root class. The resulting object contains state (instance variables) and behavior (methods) according to all the classes in this ancestry chain."
http://www.erights.org/elang/blocks/inheritance.html

BD: "... [I] defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritance, respectively).

"An alternative approach to inheritance involves a language mechanism called **delegation**, in which objects are viewed as prototypes (also called exemplars) that delegate their behavior to related objects, thus eliminating the need for classes."

NA: "A class may inherit from one class (single inheritance), from several classes (multiple inheritance), or several times from the same class (repeated inheritance). Inheritance is simply defined as the inclusion in a class ... of operations and contract elements defined in other classes ..."
"Delegation[:] Refers to shared behavior in object-oriented systems using prototypes instead of classes ..." as in the language, Self. [See Self entry--ed.]
http://www.objs.com/x3h7/sec8.htm

Simulating Multiple Inheritance

"In COM, multiple inheritance between interfaces is not supported. However, by using the derived members capability, multiple inheritance can be simulated.

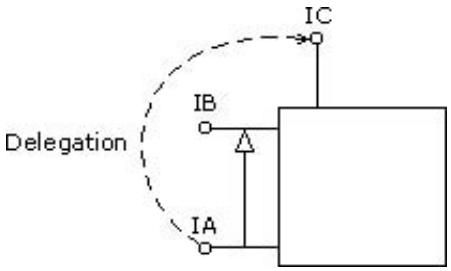
"For example, the following figure shows an interface IA that inherits from IB, and implies IC (meaning that any class that supports IA must also support IC). According to COM, IA cannot inherit from IC because it already inherits from IB. However, with **delegation**, the members of IC could be made available on IA. This is not inheritance, because IC members are not explicitly mapped into IA. Nevertheless, the result is the same because IA now includes members of both IB and IC.

"For more information about other ways of combining interface members, see Flattening Interfaces and Supporting Multiple Interfaces With Overlapping Functionality."
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/repospr/rpdefininginh_921x.asp

summarize here:

1. Inheritance encourages the designer to widen an object's interface.
2. Inheritance encourages the designer to use complex language syntax to bundle many objects into one.
3. Inheritance camouflages the fact that an object's interface is getting too large.
4. Inheritance encourages white box reuse that can cause both super and sub classes to be corrupted.
5. Inheritance camouflages delegation.
6. Inheritance, with its access protections, can make it impossible to apply certain design patterns to solve problems.
7. Inheritance encourages the construction of complex, overly restrictive class taxonomies.
8. Inheritance lessens the designer's focus on object interactions.

http://www.eros-os.org/~majordomo/e-lang/0166.html



«**Delegation** was introduced [by Lieberman 1981] as a message forwarding mechanism. The basic idea of delegation is to forward messages that cannot be handled by an object to another object called its parent in Self or proxy in Act1 … Indeed, the key-point of delegation is that the pseudo-variable "self" still points to the original receiver of the message, even if the method used to answer the message is found in one of its parent[!]» [PBL, 203]

«More than message forwarding, delegation can also be interpreted as an extension mechanism. An object B that delgates to another object A, can be viewed as an extension of A.» [PBL, 203]

In implicit delegation all non-implemented messages are delegated to declared parent objects; in explicit delegation each to-be delegated messages is named with the object handeling it [PBL].

Methods **forwarded** from A to object B are executed in B's context (not in A's as in Delegation) [EnvAcq]

(A) --fwd--> (B) ------.
^--self--'

http://www.cs.mun.ca/~ulf/pld/mocplus.html

Delegation and inheritance in ARS

In A++ in its original form inheritance is not offered as a language feature identical to the one described above, because there is no predefined syntax to define classes. It would most likely be possible to define abstractions implementing such a class system. Nobody has done it yet, because there is a very simple mechanism available that can be considered functionally equivalent to inheritance. It is called `delegation'. In our second example of object-oriented programming we will implement the inheritance using delegation.

Delegation can be very briefly explained like this: An object of a derived class receives a message from a client. If the object is equipped to provide the service it will do it. If the object does not have the proper methods to respond to the message it will delegate the message to another object, which is an instance of its super class. Messages are this way delegated up in the hierarchy until nobody can answer it, which would result in an error return.

http://www.aplusplus.net/bookonl/node67.html

Besides general data abstraction, abstract data types and object abstractions are ``two distinct techniques for implementing abstract data'' [OO/AD, 152]. There is a ``**duality** [in the categorial sense] between abstract object types on the one hand and

abstract data types on the other hand'' [CoAlg, 4]:

``Whereas abstract data types in the initial approach may be formalized as minimal soulutions (fixed points) of recursive type equations, object types may be understood as maximal solutions (fixed points) of recursive type equations'' [CoAlg, 2].

More generally:

``The distinction between algebra and coalgebra pervades computer science and has been recognized by many people in many situations, usually in terms of data versus machines. **A modern, mathematical precise way to express the difference is in terms of algebras and coalgebras. The basic dichotomy may be described as construction versus observation''** [TCACI, 4].

``The initial algebras and terminal coalgebras ... can be described in a canonical way: an initial algebra can be obtained from the closed terms (i.e. from those terms which are generated by iteratively applying the algebra's constructor operations), and the terminal coalgebra can be obtained from the pure observations'' [TCACI, 3].

Interacting-Components MOCs, i.e., MOCs with an interacting-components architecture[^], are characterized by a "signal-response" type of interaction: Computation proceeds by virtual computing units interacting with each other. Different styles of interaction and computing units lead to different models of the computation.

1. Message-passing Models, Client/Server Models or Object Models [MCOM, Quib 99-104] are closely connected with the object abstraction:

«In all other languages we've considered [Fortran, Algol60, Lisp, APL, Cobol, Pascal], a program consists of passive data-objects on the one hand and the executable program that manipulates these passive objects on the other.

Object-oriented programs replace this bipartite structure with a homogeneous one: they consist (partially in Simula, exclusively in Smalltalk) of a set of data systems [i.e. objects, c.f. p. 43], each of which is capable of operating on itself.» [PLing, 249]

(-) Inappropriate for arithmetics:

«For most people, it is natural to think of arithmetic in terms of expressions that state "do something to the following numbers." It is less natural to think of arithmetic as being performed by one number ... upon the rest.

But in Smalltalk, the expression 4 + 5 is to be understood not as "apply the plus operation to 4 and 5," but instead as "send a message to the object 4 asking it to add the value of the object 5 to its own value." ...

This will strike many people as a strange and artificial way to think about arithmetic and about integers» [PLing, 246].

5.3.1 Class Object

Smalltalk organizes all of its classes as a tree hierarchy. At the very top of this hierarchy is class Object. Following somewhere below it are more specific classes, such as the ones we've worked with--strings, integers, arrays, and so forth. They are grouped together based on their similarities; for instance, types of objects which may be compared as greater or less than each other fall under a class known as Magnitude.

One of the first tasks when creating a new object is to figure out where within this hierarchy your object falls. Coming up with an answer to this problem is at least **as much art as science**, and there are no hard-and-fast rules to nail it down. We'll take a look at three kinds of objects to give you a feel for how this organization matters.

5.3.3 The bottom line of the class hierarchy

The goal of the class hierarchy is to allow you to organize objects into a relationship which allows a particular object to inherit the code of its ancestors. Once you have identified an effective organization of types, you should find that a particular technique need only be implemented once, then inherited by the children below.

http://www.gnu.org/software/smalltalk/gst-manual/gst_31.html

http://www.ifi.unizh.ch/richter/Classes/oose2/05_Metaclasses/02_smalltalk/02_metaclasses_smalltalk.html#2%20Hierarchy%20of%20Smalltalk%20Classes

Lambda-tree graph

http://prog.vub.ac.be/Publications/1994/vub-tinf-tr-94-03.pdf

2 INTERFACES

Interfaces are fundamental aspects of object-oriented programming [Cann89], [Abad96]. The term Interface is central to object-oriented methodologies and is one manifestation of what commonly referred to as Type. Unlike a Class, which is a concrete type, an interface is an abstract type. An interface specifies messages an object will understand but has no method implementations for those messages, where a class specifies how those messages will be executed by having concrete method implementations for those messages.

f) Interfaces form heterarchies.

They relate to each other using the family tree metaphor. An interface can have parents - the immediate interfaces it extends, and ancestors - the progenitor of its family lineage. An interface can have children - the immediate interfaces extending it, and descendants - family lineage emanating from it. An interface can have twins - the interfaces equivalent to it, siblings - the interfaces who share all of its parents, and stepsiblings - the interfaces that share some of its parents.

g) At the top of the heterarchy are root interfaces, which are parentless interfaces; they extend no other interfaces.

At the bottom of the heterarchy are the leaf interfaces which are childless interfaces; no other interfaces extend them.

These classifications are not mutually exclusive (Consider the case where an environment contains a single interface - that interface is both a root and a leaf at the same time.)

http://www.iam.unibe.ch/~scg/Archive/Papers/Sade02aDynamicInterfaces.pdf

OOP-terminology and definitions

Interface

http://java.sun.com/docs/books/tutorial/java/concepts/index.html
==

The term Interface is central to object-oriented methodologies and is one manifestation of what commonly referred to as Type.

Unlike a Class, which is a concrete type, an interface is an abstract type.

An interface specifies messages an object will understand but has no method implementations for those messages, where a class specifies how those messages will be executed by having concrete method implementations for those messages

Ducasse

Six Generations of Programming Languages by Jeff Prothero

We may distinguish six generations of programing languages to date:

1. Machine Language: Coding directly in binary, octal or hexadecimal.
Alan Turing shouting up the stairway for the next mercury delay line data block to be shipped to the CPU.

2. Assembly Language: Coding abstract assembly language syntax.
I have a Mercury Autocoder manual calling this "automatic programming". :)

3. High Level Language: Fortran. Coding in algebraic expressions abstracted away from any particular assembly language.
For "programming in the large", later versions of Fortran even had -- subroutines!

4. Structured Programming: Algol, C.
"GOTO considered harmful": Strong typing, recursion, while/switch/if control constructs. Modula started us up the slow climb to fully modern module systems. Algol 68 -- the first C++ -- proved that we could define languages too complex to be implementable.

5. Object-oriented programming: Smalltalk, C++. Classes and objects.
For the first time, useful tools for really programming in the large!
A hodgepodge of good and bad ideas with no underlying theory.
As a result, every OOP language proudly proclaims that it is "the only fully object oriented programming language". Which is true -- according to that designer's particular definition of "fully object oriented".

6. Mostly-functional programming: Ocaml, SML. HOT (Higher Order Typed) languages. Milner-Hindley type deduction, higher-order functions, modern module systems which take programming in the large to an entirely new level.
http://muq.org/~cynbe/ml/ml.html

Brian Hayes on *The Post-OOP Paradigm*

"Every generation has to reinvent the practice of computer programming. In the 1950s the key innovations were programming languages such as Fortran and Lisp. The 1960s and '70s saw a crusade to root out "spaghetti code" and replace it with "structured programming." Since the 1980s software development has been dominated by a methodology known as object-oriented programming, or OOP. Now there are signs that OOP may be running out of oomph, and discontented programmers are once again casting about for the next big idea. It's time to look at what might await us in the post-OOP era (apart from an unfortunate acronym).

The classic challenge in writing object-oriented programs is finding the right decomposition into classes and objects.

Most diet books, somewhere deep inside, offer sound advice: Eat less, exercise more. Most programming manuals also give wise counsel: Modularize, encapsulate. But surveying the hundreds of titles in both categories leaves me with a nagging doubt: The very multiplicity of answers undermines them all. Isn't it likely that we'd all be thinner, and we'd all have better software, if there were just one true diet, and one true programming methodology?"
http://www.americanscientist.org/template/AssetDetail/assetid/17307/page/7;jsessionid=baa_yNAYUClzXN

3 "Everything is an object."

> "The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request.
>
> The receiver is the agent to whom the message is send. If the receiver accepts the message, it accepts the resposability to carry out the indicated action. In response to a message, the receiver performs some method to satisfy the request." Handbook

From a contextural point of view, to be able to accept/reject a message by an actor it has to have its own logical operators of transjunctions which are defining the rules of acceptance and rejection. Otherwise rejection simply means lack of compatibility, like "wrong address".

Thus, accept/reject in message-passing models are of a different kind. They are restricted by their definition as placed in a system, mostly in a hierarchy. And not by their sovereignty or autonomy as Actor models (Carl Hewitt) are suggesting. Objects and actors may have their own memory but they are not performing their own logic.

The local client/server model is embedded in a global logic. This is the case even for actors who have an own control structure as it is necessary for real concurrency.

Nothing, until now, is said about the methods of establishing a communication (acceptance/rejection) between actors. How are actors accepting/rejecting a message?

"Everything is an object."

> "Java programs are build from *classes*. From a class definition, you can create any number of *objects* that are known as *instances* of that class.
>
> A class contains two members, called *fields* and *methods*."
>
> Handbook, p. 749

At the root of everything is the class "object".

The decision, i.e. the process of deciding to set something as a root is not involved into the definition of the programming system. It starts by decision of the general designer. The designer as the ultimate decision-maker is excluded from the system.

Thus, an object as the root, is blind of its positioning by the decision of a designer.

This constitutes the *Blind Spot* of all programming. Here, of OOP.

As sketched before, the Blind Spot Problem is responsible for the exclusion of "real" interactionality and reflectionality.

Classic programming with its computational model of Turing Machines is taking a *global* view, while the "non"-classic programming paradigm of actors and object orientation is focussing on *local* and intrinsic conceptualization.

Thus, there is an interesting antagonism between the global "syntactic" model of computation, well established by classic programming paradigms, and the "societal" model of OOP.

The global approach comes back in OOP with the global structure of the distribution of classes, i.e. its pre-established hierarchy.

Less hierarchic systems may be based on a pre-established harmony (Leibniz).

3.1 Abstraction/data-oriented paradigms

1. *POP*: Problem-oriented paradigms.

2. *OOP*: Message passing actors in a singular mono-contextural computational space. Its focus can be on objects or on message passing.

For *Smalltalk-80* with its focus on objects, three principles are introduced to define the OOP language:

a) All the entities of the language are *objects*.
b) Each object is the *instance* of a class defining its structure and behaviour.
In particular, a class is an instance of another class, called its *metaclass*.
c) *Sending messages* is the unique means of communicating with objects.
Masini et al., Object-Oriented Languages, 1991, p. 53

Actors (Carl Hewitt) with the focus on message passing.

"The actor model associates the principle of encapsulation, which underlies the notion of an object, with the principle of *activity*: each object, called an actor, is an autonomous active agent communicating freely with the other actors. Actors exist in a dynamic world where activities are created, proceed and are completed. They participate in these activities and communicate with each other to transmit data and to delegate tasks."

"The actor model is completely *uniform*. It includes a single kind of entity, actors, just as the Smalltalk-80 model only includes objects.

[…] This uniformity raises the problem of *infinite regress*: if any access to information should be performed by message sending, messengers themselves would have to send a message in order to access to the message they carry and would deliver it to the receiver, and so forth.

So-called *primitive* actors, which do not need to send a message to respond a request, are provided to deal with this difficulty."

Masini et al., Object-Oriented Languages, 1991, p. 299

Actors = Objects + Activity (minus Hierarchy)
Actors = (Objects + Activity) + Interface (minus Hierarchy)

"The *interface* of an actor is called its intention and defines a contract between the actor and the outside world." Masini, p. 301

Delegation in Act1

"Delegation is not supported by Carl Hewitt's model. Like inheritance, it is a means of sharing knowledge and behaviour: an actor can have a general behaviour and be the proxy of other actors with more specific behaviour.

A specific actor, OBJECT, is the universal proxy and can be seen as the root of the "delegation tree". It is the default receiver of all the messages that are not understood by other actors.

To avoid *infinite regress* of delegation, so-called rock bottom actors never delegate and do not send messages. They correspond to the *primitive actors* of the Hewitt's model and represent entities of a few specific types: for example, numbers, symbols and lists, in the lisp implementation. Their script is held by the interpreter." Masimi, p. 312

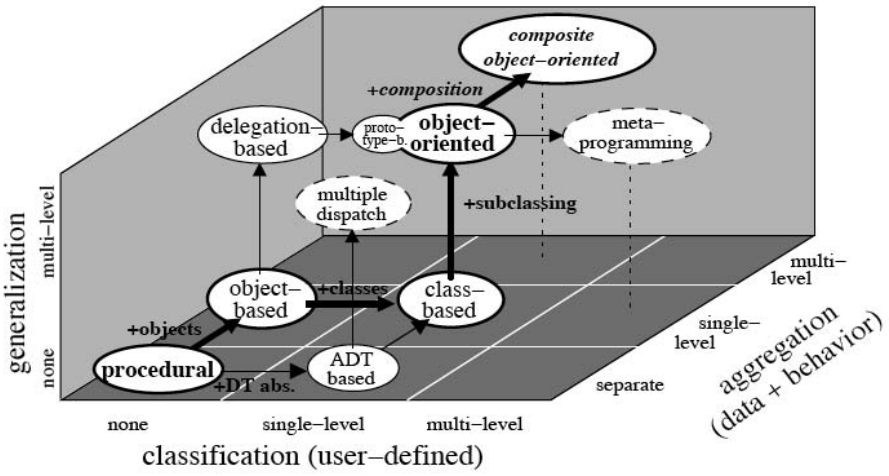


Figure 2.1: Space of programming paradigms

ulf-diss-ch2.pdf

JAR on Object-Oriented, Jonathan Rees, 2001

I think you might want to define what you mean by object and by OO. Here is an a la carte menu of features or properties that are related to these terms; I have heard OO defined to be many different subsets of this list.

1 *Encapsulation* - the ability to syntactically hide the implementation of a type. E.g. in C or Pascal you always know whether something is a struct or an array, but in CLU and Java you can hide the difference.

2 *Protection* - the inability of the client of a type to detect its implementation. This guarantees that a behavior-preserving change to an implementation will not break its clients, and also makes sure that things like passwords don't leak out.

3 *Ad hoc polymorphism* - functions and data structures with parameters that can take on values of many different types.

4 Parametric polymorphism - functions and data structures that parameterize over arbitrary values (e.g. list of anything). ML and Lisp both have this. Java doesn't quite because of its non-Object types.

5 *Everything is an object* - all values are objects. True in Smalltalk (?) but not in Java (because of int and friends).

6 *All you can do is send a message* (AYCDISAM) = Actors model - there is no direct manipulation of objects, only communication with (or invocation of) them. The presence of fields in Java violates this.

7 *Specification inheritance* = subtyping - there are distinct types known to the language with the property that a value of one type is as good as a value of another for the purposes of type correctness. (E.g. Java interface inheritance.)

8 *Implementation inheritance / reuse* - having written one pile of code, a similar pile (e.g. a superset) can be generated in a controlled manner, i.e. the code doesn't have to be copied and edited. A limited and peculiar kind of abstraction. (E.g. Java class inheritance.)

9 *Sum-of-product-of-function pattern* - objects are (in effect) restricted to be functions that take as first argument a distinguished method key argument that is drawn from a finite set of simple names.

So OO is not a well defined concept. Some people (eg. Abelson and Sussman?) say Lisp is OO, by which they mean {3,4,5,7} (with the proviso that all types are in the programmers' heads). Java is supposed to be OO because of {1,2,3,7,8,9}. E is supposed to be more OO than Java because it has {1,2,3,4,5,7,9} and almost has 6; 8 (subclassing) is seen as antagonistic to E's goals and not necessary for OO.

The conventional Simula 67-like pattern of class and instance will get you {1,3,7,9}, and I think many people take this as a definition of OO.

Because OO is a moving target, OO zealots will choose some subset of this menu by whim and then use it to try to convince you that you are a loser.

http://mumble.net/~jar/articles/oo.html

reflective uniformity

A related architectural issue is reflective uniformity. To what degree can every object in the system be itself reflective? To what extent is this desirable? There a clearly limits on reflective uniformity, since towers that reach to infinity can be envisioned along several dimensions. The issue of how regress is resolved can have a major impact on the simplicity, power, and efficiency of the resulting architecture.

I've identified three distinct strategies for resolving regress:

Induction
Circularity
Reify on Demand

Inductive regress terminates with a base-case object for which additional regress is not possible. This might be thought of as a second-class object, since it does not play by the same rules as other objects for the property in question. Such base cases are often considered primitive.

Circularity, or Idempotence, can be used to resolve structural regress. A default Class object might be though of as its own class, or a cycle of two or more objects as in the Smalltalk-80 Class/Metaclass relation might be defined. Idempotence can also be used to justify inductive-base cases, by arguing, for instance, that a primitive behaves the same way for each case from here to infinity.

Reification-on-demand, or Lazy Reification, can be used where every object in the system has, by definition, an implicit relationship with a certain short of metaobject. These relationships can be such that the object need not be explicitly reified until an explicit request to access or modify them is made. Lazy reification can permit conceptually infinite towers of metaobjects to be made available on an as-needed basis. 3-KRS uses this strategy to provide per-instance metaobjects. Reify-on-demand strategies can be used to deny users access to second-class objects, by dynamically interposing a first-object when access to a base-case object is attempted.

Brian Foote, An Object-oriented Framework For Reflective Metalevel Architectures
http://www.laputan.org/reflection/ooffrmla.html

3. *Subject-Oriented Programming* (SOP): depending on viewpoints, roles, aspects, etc. in a single complex mono-contextural programming framework including multi-paradigms.

Subject-oriented programming: overview of concepts

Subject-oriented programming is a *program-composition* technology that supports building object-oriented systems as compositions of subjects. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects.

Subject-oriented programming thus supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another (perhaps with "glue" or "adapter" subjects).

The flexibility of subject composition introduces novel opportunities for developing and modularizing object-oriented programs. Subject-oriented programming-in-the-large involves determining how to subdivide a system into subjects, and writing the composition rules needed to compose them correctly. It complements object-oriented programming, solving a number of problems that arise when object-oriented technology is used to develop large systems or suites of interoperating or integrated applications.

http://www.research.ibm.com/sop/sopoverv.htm

4. In-between: TOP Table-oriented programming

Table-Oriented Programming (TOP for short) can be characterized as a programming language and/or development method that makes dealing with tables, lists, indexing, sorting, searching, filtering, etc. a native and direct part of language design and complexity management. This is a contrast to the clumsy collection API's and attribute management techniques such as set/get made popular by object oriented programming vendors. Table-Oriented Programming can also be characterized by using tables to organize program logic, not just data. Such tables are called Control Tables. They offer potential organization benefits over both raw procedural programming and object oriented programming.

http://www.geocities.com/tablizer/top.htm

It is a philosophy of Table Oriented Programming to control as much of relationships and behavior dispatching using Boolean expressions or expressions in general as possible. Tables-Of-Rules are a lot easier to change and manage and view than programming code in most cases.

http://www.geocities.com/tablizer/subtypes.htm#patterns

All, OOP, AOP, TOP and SOP, are dealing with *"more and more abstract notions of data"*, i.e. more and more powerful abstractions over data to reach more powerful computations. All three, and many other too, are, like POP, abstraction/data-oriented paradigms.

"Programming is data-driven: to design a given software system, a programmer begins defining the appropriate types of object, with their specific operations. Each entity manipulated by the system is an *instance* of one of these types and hold for the particular keys to its behaviour."

Masini, p. 4

From IBM:

> Subject-oriented programming is a program-composition technology that supports building object-oriented systems as compositions of subjects. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. Subject-oriented programming thus supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another (perhaps with “glue” or “adapter” subjects).

http://jaxn.org/blog/archives/1791-subject-oriented-development

SOP, AOP and TOP may emphasis the fact that modeling can happen in very different ways leading to different implementations. Thus first, a programmer is not concerned with data-driven interests but with the conceptualization of the domain under consideration. Also SOP and TOP may support multi-paradigm approaches, what is still missing is a simultaneous implementation of the different interacting models of thematization in a complex framework.

Multiparadigm

Contextural programming in contrast is not data-driven but is focused on the management of the interaction modi of different ways of programming. The modi of the interplay between paradigms are prior to what these paradigms are intended to program. Thus, paradigms are becoming the new objects of programming. But this contextural approach has to be separated from the *"compositional approach"* of dealing and programming different paradigms (Diomidis D. Spinellis, Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming, Diss., London, 1993). This approach is dealing paradigms as objects, thus producing a simple circularity: The paradigm of objects is itself an object; including other paradigms as objects.

Short: http://www.spinellis.gr/pubs/conf/1994-PLSA-Multipar/html/paper.html

What do we learn?

To avoid paradoxes and to stop infinite regress we have to introduce some urgency devices, often well hidden security agents, which have special privileges and are guaranteeing the functionality of the system. But it is declared, that the postulated principle of homogeneity of objects or actors is not tangled by this strategy because these security agents are agents, too. Thus, everything is still one thing: agents. Except we start to have problems with the small and hidden security agents. Then we need some more distinctions: security agents of security agents for our ordinary agents; and so on.

Grammatologically, or more precisely from a graphematic point of view, the difference of use and mention is obsolete. It is not enough to mention that all actors are actors independently of being primitive or not. This mentioning happens in the mind of the programmer and not in the programming text. The text is avoiding circularity only if working with the distinction of first and second class actors. To sublime this distinction, in saying both are anyway nothing else but actors, is an abstraction in the mind of a programmer and is not inscribed in the text. Written, it works only with the distinction of a two-tiered class system of first and second-order citizens.

Why do we need general object-schemes?

If an objects definition depend on a subjective selection, say by an aspect, role, view-point, etc. then there is no positive characteristics involved to define the object "as such", i.e. independent of its thematization. Because each attempt to give an abstract definition of an object would be a decision involving a relative and local application or realization of a viewpoint.

One solution is to define this abstract object as a zero, nil or empty object. An example is given by the aspect-oriented implementation of a bank account.

This is a kind of a "Big Bang Theory". Out of nothing, the whole world. In other words, it is a creatio ex nihilo conception.

It says, instead of some characteristics, which all are depending on subjective decisions, we chose to have an empty set of decision, say, no decisions. But this manoeuvre remains in the domain of subjective decisions. Nil decision is also a decision.

The other problem, which seems to be more serious for programming is the total loss of structure and structuration by the nil-approach. Similar to the Buddhist statement: The emptiness itself is empty.

From a morphogrammatic "point of view", this withdrawal is at once not necessary and not decisive enough.

The nil-decision also has to be withdrawn.

The morphogrammatic abstraction is a subversion which is giving up all positive characteristics of the definition of an object but is keeping all the structure of its distribution in the kenomic space.

As improvised before, this is introducing the non-concept of *"general object-schemes"*.

Thus, subjective roles of thematizations are played on the arena of morphograms.

People fear this subversion, thus they declare the arena as a role, too. For them, the arena is a zero-role. And a zero-role is nevertheless a role.

Linguistic metaphor: Subject, Object and Context

If we take the linguistic model or metaphor of conceptualization we end up with the subject/object-structure of rational sentences.

> Some experts say that the original definition of object-orientation came from the *object* in *grammar*. The requirements for software are always *subject*-oriented. However, since the requirements for the subject are often complicated, subject-oriented programs end up tending to be complicated and monolithic. Therefore, as an alternative, some researchers started thinking in an object-oriented way. This represented a *paradigm shift* from the usual or previous subject-oriented mode of thinking.
>
> According to object-oriented principles, the *verb* in a program statement is always attached to the object, and the logic associated with a requirement is likewise handled in the object. The following are some examples of the ways by which a subject-oriented requirement is translated into object-oriented thinking:
>
> * *Subject-oriented:* The Sales Application saves the Transaction
> * *Object-oriented:* The Transaction saves itself upon receiving a message from the Sales Application
> * *Subject-oriented:* The Sales Application prints the Receipt
> * *Object-oriented:* The Receipt prints itself upon receiving a message from the Sales Application
>
> http://en.wikipedia.org/wiki/Object-oriented_programming

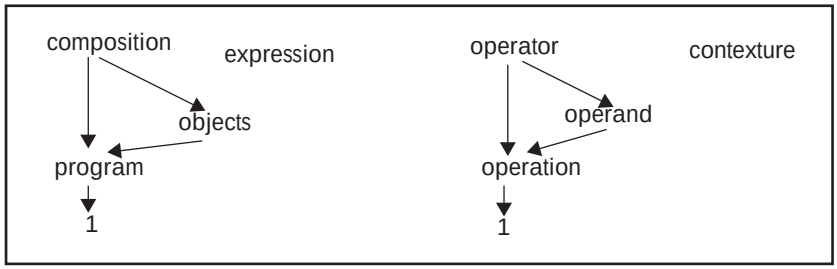
Procedural programming is concerned with the *subject*. The subject is the operator which is manipulating its objects, the operands.

Object-oriented programming is turning the order, it is the *object* which is manipulating itself after it received a message to do it. The object can be an object, class, proto-type, pattern, etc.

Subject-oriented programming is considered with the *context* in which the linguistic model of subject / object or operator / operand is stated. To emphazise aspects, roles, view-points as the opening activity which allows to conceptualize and programm in a procedural or objectional manner, can be called *context-oriented* programming. Context-orientation is collecting subject-, role-, aspect-, table-, pointview-, etc oriented programming.

Algebra and Co-Algebra of Programming

From a structural point of view, procedural programming is algebraic and object-oriented programming is co-algebraic both can be expressed in 1-category theory.



Syntax of the ς-calculus

a,b ::=		terms
x		variable
$[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$	(l_i distinct)	object
a.l		field selection / method invocation
$a.l\Leftarrow\varsigma(x)b$		field update / method override

Here, an object $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ has method names l_i and methods $\varsigma(x_i)b_i$. In a method $\varsigma(x)b$, x is the self variable and b is the body.

Notation

- $o.l_j:=b$ stands for $o.l_j\Leftarrow\varsigma(y)b$, for an unused y. We call $o.l_j:=b$ an *update* operation.
- [..., l=b, ...] stands for [..., $l=\varsigma(y)b$, ...], for an unused y. We call l=b a *field*.
- We identify $\varsigma(x)b$ with $\varsigma(y)(b\{x\leftarrow y\})$, for any y not occurring free in b.

To complete the formal syntax of the ς-calculus we give the definitions of free variables (FV) and substitution ($b\{x\leftarrow a\}$) for ς-terms.

Martin Abaldi and Luca Cardelli, A Theory of Primitive Objects

Abstract : Concrete Coalgebra

Coalgebras are the formal dual of algebras and have over the last decade or so become popular in both mathematics and computer science as unified models of various state-based dynamical systems, including: automata, transition systems, (infinite) data types, object- and component-based systems, Kripke models, discrete event systems, and many many more. In these lectures, we shall briefly review the basic notions of the discipline of coalgebra, which are bisimulation, finality, and coinduction. Next we shall mention, more concretely, a few particularly interesting and useful (final) coalgebras, notably streams, formal languages, and formal power series. Finally, we shall sketch a recent application of a coinductive calculus of bitstreams to sequential digital circuits, and show how this relates to the semantics of Reo, a framework for the compositional construction of component connectors, recently developed by Farhad Arbab.

http://cs.ipm.ac.ir/OneDayWorkshop.htm#BoerAbstract

There are well-known foundations for different programming paradigm such as those for imperative and functional programming. The object-oriented paradigm is an example of one that lacks a commonly recognized formal foundation.

A formal semantic for the description of systems of concurrent objects should comprise at least two different views on types. On the one hand it is possible to see a type, i.e. a functor, as a signature for an initial datatype. In this case the initial algebra forms the weakest or least solution for the given type equation. Elements of the initial algebra are finite.

On the other hand the formalization of possibly infinite behavior of objects requires infinite structures. The greatest solution of a type equation formed by the terminal coalgebra to the corresponding functor provides tools for the description of the behavior of objects. The carrier set of the terminal coalgebra can be viewed as the collection of the behavior of all automata satisfying the type specification.

> The framework of algebras and coalgebras of a functor offers for example a good way to construct objects with an initial datatype as the output alphabet as well as lists of object behavior.
>
> http://pll.cpsc.ucalgary.ca/charity1/www/people/ulrich.html

Programs belong to initial algebras, their solutions are finite, i.e they have a termination, which is the solution of the problem. This is modeled along the metaphor of a *calculator*.

Objects belong to terminal algebras, their behavior is not finite, i.e., they may not have a termination. This is modeled along the metaphor of the *postman* (or mail system).

To solve a *problem* with a program, the algorithm has to terminate and to deliver a finite result. To maintain a system, the system has to stay pertinent and to provide information to requests.

3.2 Thematization / contextures realizing / evoking paradigms

5. *Contextural Programming*: Chiastic interplay between contextures.

Subjectivity is a name for the chiastic interplay between cognition and volition, including motion and other categories, in complex architectonics of programming and computation.

Subjects are distributed complementarities, i.e. systems with environments.

Subjects, also called agents, have an inner and an outer environment.

Subjectivity consists of distributed objects and subjects and their interplay.

3.3 Subjectivity matters; everywhere.

From "everything is an object" we move to "everything is a subject" to "everyone is into subjectivity" to "subjectivity matters; everywhere".

That is, from object-oriented to subject-oriented to contextural programming.

As a consequence, we have to deal not with a polymorphic language but with a polycontextural one.

Blind Spot of Oriented Orientation

Data / object / subject-*oriented* programming is not reflecting the fact of its orientation. That is the act to decide to orient interest, focus, conceptualization, etc. towards a selected notion, category or aim is not included in the range of its orientation.

Object-orientation is oriented towards objects, excluding the act of orientation.

In other words, x-oriented paradigms are first-order systems, while systems which are including the action of orientation into their range of orientation are second-order conceptualizations.

Object-oriented programming is not oriented on its own orientation, or orientedness, but on objects. And the process of orientation which is guiding object-oriented programming is itself not an object of being oriented for. This kind of processuality can not be objectified in OOP. To try it would produce immediately paradoxes, i.e. antinomies. Processes of orientation, inside the system, can be named and therefore objectified only secondarily as a meta-orientation. But then, the game starts again on the meta-level; and can be "enjoyed" ad infinitum.

Thus, the statement, "everything is an object" is either a contradiction if true or simply wrong. As many slogans are.

In other words, OOP is not offering conceptual and programming means to thematize at once the process of orientation and the goal of orientation, its object.

That is, the orientation of orientation of object-oriented programming is taboo.

> "Object-oriented programs replace this *bipartite* structure with a *homogeneous* one: they consist [...] of a set of data systems [...], each of which is capable of operating on itself."

Logically speaking, this homogeneous or "type-free" approach which is *"capable of operating on itself"* is self-contradictory.

But, because there is not much information avaible about what exactly is meant by this statement, its savety will probabily be garanteed in some other unsafe heavens.

On the other hand, it sounds like a misleading exageration. Because, additional to the notion of objects there are also some messages around.

Traditionally, there are about 3 strategies to deal with such paradoxes:

1. mapping of the contradictional construction onto a *time* axis,
2. mapping of it into a topology of *space*, or
3. domesticating it with the help of some *paraconsistent logics*, or similar.

Those strategies are aimed to avoid the problem of paradoxes without strictly disallowing its construction by a *vicious circle principle* (Russell) which is in favor of type theory.

http://plato.stanford.edu/entries/russell-paradox/

Again, in OOP, *"everything is an object"*. Is this sentence an object of OOP?

But, *time* and *space* are not necessarily good logical principles, nor is it reasonable to disallow the construction of paradoxes (antinomies) or to accept contradictions and to weaken formal soundness by its domestication.

Seven Paradoxes of Object-Oriented Programming Languages

David Ungar, Sun Microsystems

Although many of us have worked to create good object-oriented programming languages, it would be hard to say (with a straight face) that any of our creations have totally succeeded. Why not? I believe that this endeavor is essentially paradoxical. Thus, whenever a language designer pursues a particular goal and loses sight of the lurking paradox, the outcome is an all too often fatally flawed result. One way to think about this is to explore the following seven paradoxes:

1. Because programming languages, development environments, and execution engines are intended for both people and computers, they must both humanize and dehumanize us.
2. Adding a richer set of concepts to a programming language impoverishes its universe of discourse.
3. Putting a language's cognitive center in a more dynamic place reduces the verbiage needed to accomplish a task, even though less information can be mechanically deduced about the program.
4. The most concrete notions are the most abstract, and pursuing comfort or correctness with precision leads to fuzziness.
5. Although a language, environment, and execution engine are designed for the users' minds, the experience of use will alter the users' minds.
6. Object-oriented programming has its roots in modeling and reuse, yet these notions do not coincide and even conflict with each other.
7. A language designed to give programmers what they want may initially succeed but create pernicious problems as it catches on. However, a language designed to give programmers what they really need may never catch fire at all.

Many of these assertions seem nonsensical, misguided, or just plain wrong. Yet, a deeper understanding of these paradoxes can point the way to better designs for object-oriented programming languages.

http://www.oopsla.org/oopsla2003/files/key-4.html

3.4 Abstraction vs. mediation

I often find that real world "abstraction" needs to be **relative** (at least with regard to custom business software). The "irrelevant details" for one "viewer" may be very relevant to another.

I see nothing inherent in OOP that assists this process more than other paradigms. If you do "see it", then please point it out.

OOP modeling tends to assume that **relevancy is global.**

On the other hand, procedural/relational handles this issue by making abstraction be more ad-hoc using queries, and generally limiting an abstraction to a particular task or event.

OOP modeling tends to be "nested" and/or hierarchy based. Procedural/relational abstractions tend to be graph-based (networked) and set-based. Graphs are a more generic modeling tool than nesting. A particular nesting can be a temporary or virtual view under relational or Boolean graphs.

http://www.geocities.com/tablizer/abstract.htm
http://www.geocities.com/tablizer/oopbad.htm

EXAMPLE

What we can learn from these examples can be summarized in the sentence: *It is not the hierarchy which is the problem it is the belief in its* uniqueness *which is at the root of the conflicts.*

In other words, in all single examples, locally, a hierarchy is doing the job well. The problem is the interacting of the different hierarchies, the mix and overlapping of the different decision of relevancy of hierarchies which can not be treated properly in one single growing hierarchy.

Heterarchies are not denying the principle of hierarchic order but are giving space to a parallelity of different hierarchies and their interaction. Not only the relativity of the hierarchies have to be taken into account but also the fact that they exist at once together and are involved in an interacting game. Heterarchies are managing the play between global and local aspects of relevancy in organizing programming.

Thus, heterarchies are the mechanism of the mediation of hierarchies.

3.5 Composition vs. inheritance

Composition is contrasted with subtyping, which is the process of adding detail to a general data type to create a more specific data type. In composition, the composite type "has an" object of a simpler type, while in subtyping, the subtype "is an" instance of its parent type. Composition does not form a subtype but a new type.

Inheritance — a mechanism for creating subclasses, inheritance provides a way to define a (sub)class as a specialization or subtype or extension of a more general class: Dog is a subclass of Canidae, and Collie is a subclass of the (sub)class Dog. A subclass inherits all the members of its superclass(es), but it can extend their behaviour and add new members. Inheritance is the "is-a" relationship: a Dog is a Canidae. This is in contrast to composition, the "has-a" relationship: a Dog has a mother (another Dog) and has a father, etc.

Multiple inheritance – a Dog is both a Pet and a Canidae – is not always supported, as it can be hard both to implement and to use well.
http://en.wikipedia.org/wiki/Object-oriented_programming#Fundamental_concepts

The easy way:

is-abstraction:
inheritance; element-set-relation, inclusion, partonomies, class-inclusion
has-abstraction:
delegation, set-inclusion-relation, part-whole, taxonomies, class-membership
as-abstraction:
mediation.

Delegation can be termed "run-time inheritance for specific objects".

In OOP the emphasis is not on how computation is organized, but on how **program text** is decomposed into modules, because it is this decomposition that matters as to the program text's comprehensibility and maintainability.

OOP is based on the assumption that the program text's comprehensibility and maintainability are improved by decomposing the **text** into modules, and that the best way to decompose it into modules is to minimize dependencies among modules and maximize the cohesion of functions inside each module, and that this is best achieved by encapsulating the representation of a data type in each module.

Some critics allege that the relational model is a superior categorization and large-scale structuring tool due to its mathematical foundations in predicate calculus, relational calculus, and set theory. OOP is not based on a mathematical model.

Attempts to find a consensus definition or theory behind objects have not proven very successful, and often diverge widely. For example, some definitions focus on mental activities, and some on mere program structuring. One of the simpler definitions is that OOP is the act of using "map" data structures or arrays that can contain functions and pointers to other maps, all with some syntactic and scoping sugar on top. Inheritance can be performed by cloning the maps (sometimes called "prototyping").
http://en.wikipedia.org/wiki/Object-oriented_programming

The crucial inheritance feature with its possible hierarchy and all deriving problems out of it are conceptually not necessary for the definition of the OOP paradigm. There is OOP without inheritance and hierarchy. But, who pays the price for this liberation?

> "It would be possible to stop with the encapsulation concept and build a language that provides classes and instances, but not inheritance.
>
> Inheritance is not a necessary feature of an object-oriented language, but it is certainly a extremely desirable one.
>
> Also, the simple linear scheme to be described here is not the only way to provide inheritance. For example, some languages provide multiple inheritance [...]."
>
> "Inheritance is a tool for organizing, building, and using reusable classes. Without inheritance, every class would be a free-standing unit, each developed from the ground up. Different classes would bear no relationship with one another, since the developer of each provides methods in whatever manner he chooses. Any consistency across classes is the result of **discipline on the part of the programmer.**"
>
> Brad J. Cox, Object-Oriented Programming, p.69

Classes as "free-standing units", i.e. autonomous and autarkic entities, are very similar to Leibniz Monads.

In a complex (programming) world it seems to be a dangerous luxury to trust in the discipline of programmers. Leibniz solved the problem of interaction between his Monads not with delegation of the discipline to the monarchs but with the help of a unique and ultimate Central-Monad, known as God. The central monad is guaranteeing the rationality of interaction between the autarkic monads by establishing a general, pre-established harmony.

It is only natural, that today, some kind of a hierarchic topology of the distributed classes, enabling inheritance, is presumed.

After hierarchy is established, the counter-movement of heterachizing the harmony has started, say with AOP.

> "A language will be called object-oriented if it is object-based and additionally requires that objects have classes and classes have inheritance:
> *object-oriented = objects + object classes + class inheritance.*"
> -- Wegner. The Object-Oriented Classification Paradigm. 1987

In contrast:

> In our view, the essence of object-oriented programming is not inheritance (multiple or otherwise), nor is it message passing (which is after all just a metaphor for procedure calling), but is rather *the organization of memory into local objects*, as opposed to having a single global store. This property makes object-oriented programming easier to understand and modify, and also explains its relevance to distributed computing.
>
> -- Goguen & Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics, 1987

Both in: http://www-users.cs.york.ac.uk/susan/cyc/o/object-o.htm

Bruce D. Shriver, Peter Wegner, editors. Research Directions in Object-Oriented Programming. MIT Press. 1987

4 Leibniz' Monadology

Leibniz' Monads

7. Further, there is no way of explaining how a Monad can be altered in quality or internally changed by any other created thing; since it is impossible to change the place of anything in it or to conceive in it any internal motion which could be produced, directed, increased or diminished therein, although all this is possible in the case of compounds, in which there are changes among the parts. **The Monads have no windows**, through which anything could come in or go out. Accidents cannot separate themselves from substances nor go about outside of them, as the 'sensible species' of the Scholastics used to do. Thus neither substance nor accident can come into a Monad from outside.

http://oregonstate.edu/instruct/phl302/texts/leibniz/monadology.html

Smalltalk and Monads

Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealisations of concepts--Ideas--from which manifestations can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea--which is a-kind-of itself, so that the system is completely self-describing-- would have been appreciated by Plato as an extremely practical joke [Plato].

http://www.smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_Introduction.html

Early History of OOP by Alan C. Kay

I recalled the **monads of Leibniz**, the "dividing nature at its joints" discourse of Plato, and other attempts to parse complexity. Of course, philosophy is about opinion and engineering is about deeds, with science the happy medium somewhere in between. It is not too much of an exaggeration to say that most of my ideas from then on took their roots from Simula--but not as an attempt to improve it. It was the promise of an entirely new way to structure computations that took my fancy. As it turned out, it would take quite a few years to understand how to use the insights and to devise efficient mechanisms to execute them.

http://accesscom.com/~darius/EarlyHistoryST.html

Monads have no windows but they are representing the world from their own perspectives.

Monad = ((cognition + perception) + perspective) + harmony

Object = ((methods + data) + interface) + hierarchy (inheritance)

Dual nature of objects = (data + behavior) as an identifiable unit (Schunemann)

Problems: organization of interaction of autonomous monads (objects)

Solution: Leibniz + Central monad (God), OOP = hierarchy

Duality of object reference:

Contrast: Leibniz against Descartes, OOP against operator/operand-constructs (functional programming)

Monads don't have windows or doors, they have interfaces.

4.1 Operator/operand

"Actors blur the conventional distinction between data and procedures." **H. Lieberman**

Lambda Calculus based programming languages are well supported by functional and operational mathematical concepts, like operator, operand and operation with all its laws of constructional systems, i.e algebras.

Conceptualizations and formalizations of objects as monads, involved in message passing with other monads, tend to deny this mathematical structure.

Before accepting the co-algebraic structure of the concept of objects, a "pheomenological" description of what's going on with objects can be given in applying operational terms.

```
Object = (fields + methods) + interface.
```

It is said, that objects, in their isolation, that is, as autarkic monads, are programs in themselves, maybe as an abstraction from modules. This can be transcribed that objects are realizing a classic (functional) operator/operand-distinction in themselves.

Thus first, between methods and data a classic operator/operand mechanism is established and working.

Second it is said, that autarkic objects are interacting with other autarkic objects via message passing. Thus, between objects and message-passing a second operator/operand relation exists. Obviously, it is the object which is sending a message and not the message is objectifying an object. This case is not totally excluded but happens on a further level of operator/operand distinctions where a message is seen as an object.

Third, to be able to address another object, the object needs a representation of the other object(s) in itself. This is realized by an *"object identifier"* which is ruling the *"object reference"* to another object which itself has an "object identifier" of the addressing object.

In other words:

> "[...] a message includes the object, called the receiver of the message, the selector of the method to be activated and the arguments to which the method applies.

```
send(receiving-object, selector, argument-1,... argument-N)
```

More explicit:

> "We should, however, not forget that all instances of a given class share the same method *dictionary*, held by the class from which they arise. The selector should thus be located in the dictionary of the class of the receiving object:

```
apply(method-associated-with(selector, class-of(receiving-object)),
agument-1,
...,
argument-N)
```

Masini, p.24/25

cit. ulf

Duality of objects is not ambivalency of objects.

All those maneuvers are part of the pre-established harmony of the system. They are the pre-given requisites on an architextural level for the functioning of the interactions as sending and receiving messages between objects.

Even if new objects can be created by the system these prerequisites are not interactional and reflectional in the sense of contextural programming. Simply in contextural programming the harmony is never pre-given by an external desiner but are always constructed by the system and in risk of failing and in chance of creation of new possibilities. This trivially is not excluding the persistence and accumulation of successful harmonizations.

Interface

Unlike a Class, which is a concrete type, an interface is an abstract type. An interface specifies messages an object will understand but has no method implementations for those messages, where a class specifies how those messages will be executed by having concrete method implementations for those messages.

abstract type = interface
concrete type = Class
concrete Class = object

According to the construction of the concept of an object with its constituents "data", "methods" and "interface" we observe a tri-partite application of the denied operator/operand dualism.

As a result we can say, the object-oriented conception of objects is not beyond the operator/operand dualism but is involved, at least, in a 3-level application of it.

It seems that this "phenomenological" result is well mirrored by the different attempts to formalize object-orientedness by means of typed-lambda calculi.

This is, obviously, in some contrast or even contradiction to the statement of the co-algebraic nature of the object-oriented approach to programming.

Contextural programming is not based on the operator/operand dualism but on their proemiality, i.e., distribution and mediation, which is beyond dualism. On the other hand, my own use of the operational terminology is not restricted to functional programming concepts.

5 Everything is an Actor

5.1 More metaphors: Actors in a Society

> "First, *knowledge must be distributed* among the members of the society, not *centralized* in a global data base. Each member of the society should have only the knowledge appropriate to his (or her) own functioning."
>
> [We will sometimes "antropomorphize" actors by referring to "he" instead of "it". We could also say "she".]" Lieberman, p. 10

Everything, we should say, everyone, is an actor.

When I was studying Hewitt's society model of computation in the late 70th I was quite happy to find a close connection between computer science and sociology. But two or three problems had been unanswered within Hewitt's concept, until now:

1. Message passing as a simultaneous, mutual and reciprocal exchange relation between autonomous Actors as addressed and as addressee. In contrast, this kind of exchange relation was perfectly analyzed by Karl Marx and some formalization had been realized by Gotthard Gunther (1968) and later by myself.

2. As Lieberman has pointed out much later (1987) we can distinguish, in a "antropomorphic" sense by speaking of Actors between "he" and "she" Actors. But, in his text, it makes no difference. There is no tension between he-actors and she-actors. It is not more than a nice gesture of a nice man that we could, still in an antromorphic way, choose instead of he "she". This could be fine if our lieber Mann would not have forgotten that there is, in an "sociomorphic" sense, no society without "it". That is, the irreducible difference of *she* and *he;* as actors.

3. Problems of infinite regress. Because of the "type-free" language, everything is one, very old patterns of antinomies and paradoxes and other loops and circularities are asking for treatment.

> "There's only one kind of thing that happens in an actor system – an *event*."
>
> "An event happens when a *target* actor receives a *message*. Messages are themselves actors, too."

Thus, what's missing as an event is the medium which is the message.

Philosophically, or simply linguistically, everything depends on the inconspicuous term "are" or "is". In applying the sentence (slogan, axiom) "Everything *is* an actor", the "is" is used in many ways without getting an adequate treating. In contrast to this highly idealistic understanding of "is", in a more behavioral linguistic the meaning of a word, "is" defined by its use (Wittgenstein). Obviously, the is-word is used in many different ways without its traces being inscribed. Its generalized meaning is in the head of the user and not realized in the script.

This idealistic and formalistic position is well developed in Curry's Combinatory Logic and philosophy. It should be mentioned that pure combinatory logic (calculus) has a higher abstractness than logic and is producing contradictions if interpreted logically.

Also it is possible to create new actors, in a "biomorphic" sense, there is no evolution in such an event-system of actor creations.

"How does an actor system keep from getting caught in an *infinite loop* of sending messages to actors, causing more messages to be send to other actors, without any useful computation being performed? The recursion of actors and messages must stop somewhere, some primitive data types and procedures are needed. Yet the implementation should remain faithful to the theory, which says that all components of the system are treated as actors and obey the message passing protocol."

Here, finally enters the chance to give the he/she difference a crucial meaning not having existed before while the system was designed. For a clean housekeeping we can now, without violating the principles of the actor system, separate the he-actors as the actors with unrestricted, say speculative power of creating events, from the she-actors who are concerned with the economy of such speculations. Or *vice versa*, too. Without such a corrective realized by she-actors the event-culture would surpass all limits of given resources. She-actors are guaranteeing the realization of the event system by preventing them from overheating in the joy of the infinite regress (or progress?).

Lieberman's French Connection

Liberty: each actor has full control on its own behaviour.
Equality: any entity is an actor and none is priviledged.
Fraternity: actors share knowledge and send each other messages.

"[...] the Actor model might be described by some as before its time."
http://en.wikipedia.org/wiki/Actor_model_early_history

rock bottom actors and delegation in Act1

"Delegation is not supported by Carl Hewitt's model. Like inheritance, it is a means of sharing knowledge and behaviour: an actor can have a general behaviour and be the proxy of other actors with more specific behaviour.

A specific actor, OBJECT, is the *universal proxy* and can be seen as the root of the "delegation tree". It is the default receiver of all the messages that are not understood by other actors.

To avoid *infinite regress* of delegation, so-called *rock bottom actors* never delegate and do not send messages. They correspond to the *primitive actors* of the Hewitt's model and represent entities of a few specific types: for example, numbers, symbols and lists, in the LISP implementation. Their script is held by the interpreter." Masimi, p. 312

And nevertheless, the liberal illusion is dominating the differences. *"Whereas classes are hierarchically arranged in an inheritance graph, actors all coexist at the same level, without discrimination."* Masimi, p. 332

Again, we learn, that liberalism is depending on the work of a class of not-privileged but exploited actors: *rock bottom* or *primitive actors.*

Later, an extension in the other direction happens: meta-actors.

Hierarchy: rock-bottom actors –> actors –> meta-actors.

Infinit regress solution strategies

Hewitt: paraconsistency of open systems

Maturana/Varela: re-entry and domestication of antinomies.

Does it matter?

Structurally, without such nice metaphors, like Lieberman's, this event of exploiting some special objects happens in all existing (theories of) programming languages. There is always some crucial work to do which is not given similar rights and status in the system as the others are enjoying. To put it in metaphors, again.

Thus, the saying *"Everything is an actor"* or similar statements, are all speculative exaggerations.

In other words, a mail(man) system, even including chain letters, is not yet defining a society.

Without playing with metaphors, we simply could have asked the system: "Is the statement *"Everything is an actor"* itself an actor?".

In other words, an actor system is a homogeneous system and has as such no environment. Without systemic environments, no reflectionality and no interactionality, in the sense of contextural programming, is accessible.

An important aim of the society metaphor is to surpass the hierarchic military command model of programming.

Further steps in linguistic metaphors:

me / myself

you / theyself (yourself)

+ chiasm between both

Collection of historic papers for Actor system:

http://www.cypherpunks.to/erights/history/actors.html

"Actors: A Model of Concurrent Computation in Distributed Systems"
Gul Agha, MIT AI Lab Technical Report 844.
http://www.cypherpunks.to/erights/history/actors/AITR-844.pdf

6 Anything said is said by an observer: Humberto R. Maturana

The Observer

Anything said is said by an observer. In his discourse an observer speaks to another observer who could be himself, and whatever applies to one applies to the other as well.

Maturana, H.R. 1970, "Neurophysiology of cognition", in Garvin, P. (ed.) 1970, Cognition: A multiple view, New York/Washington, 3-23, p. 4

The starting point.

My starting point is our use of language: Everything said is said by an observer to another observer that could be himself.

Unity (or operational whole): an observer defines a unity by specifying the operations of distinction that separate a discrete entity from a background. The observer may thus distinguish a simple or a composite entity. If he distinguishes a unity as an "atom", as an entity without parts, he distinguishes a simple unity. If he distinguishes a unity as made of components, he distinguishes a composite unity.

Maturana, Cognition 1978

http://www.enolagaia.com/M78bCog.html

Observer

An observer is a human being, a person, a living system who can make distinctions and specify that which he or she distinguishes as a unity, as an entity different from himself or herself that can be used for manipulations or descriptions in interactions with other observers.

An observer can make distinctions in actions and thoughts, recursively, and is able to operate as if he or she were external to (distinct from) the circumstances in which the observer finds himself or herself. Everything said is said by an observer to another observer who can be himself or herself.

Does it matter for the theory of cognition and language to draw a gender distinction? Maturana's text goes on:

Unity. A unity is an entity, concrete or conceptual, dynamic or static, specified by operations of distinction that delimit it from a background and characterized by the properties that the operations of distinction assign to it. A unity may be defined by an observer either as being simple or as composite.

Maturana, Biology of Language: The Epistemology of Reality

http://www.enolagaia.com/M78BoL.html

There is no difference in the theory after the introduction of the gender distinction.

Both paragraphs about the crucial term "unity" are more or less declaring the same situation. Thus, it is correct to state: *"A unity may be defined by an observer either as being simple or as composite."* without implying any distinctions derived from the gender distinction.

For short, a unity is a unity, independent of the difference between the he/she-observers. That is, a unity depending at once on a he-observer and a she-observer would not only be "simple" or a "composite" but it would also be of some ambivalent complexity as an "indecidable" not being identified in a single and simple act of distinction and naming. Because it would have to be distinguished from the positions of the he- and/or she-observers. That is, the gender distinction for the observer(s) introduced by Maturana is made before the definition of the term unity, therefore the gender distinc-

tion should make a difference in the definition of "unity". This difference is not simply "inherited" by "unity", it has to be developed explicitly.

To calm tempers, I can mention here, that I will develop below, that the difference has not to be "antropomorphically" limited and thus can be "deliberated" from possible "feminist" implications. In polycontextural systems, observers comes in in a heterachic plurality. Neutral, or he/she-observers, can be introduced as applications of the general theory of polycontexturality.

Even if the definitions of Maturan's theory would take into account some differences between two positions, there is no guarantee that they are not appearing as much more than a differentiation of a neutral standpoint, anthropomorphized as a "person" or sociologically as an "individuum".

A distinction like ego and alter-ego still remains in the homogenous definition of a individuum.

> Thus, "In his discourse an observer speaks to another observer who could be himself, and *whatever applies to one applies to the other as well*." Maturana

"Draw a distinction!" (Spencer-Brown) is never a double inscription.

historic parallelism of wordings

Also this kind of wordings are not occurring surprisingly, they occur at the very beginning of two influential theories of the late 70th and early 80th: Carl Hewitt's Actor Theory of computation of (also in the Object Theory of programming, Smalltalk) and Humberto Maturana's Autopoiesis Theory of Living Systems. What we observe is a kind of a historic parallelism in the wordings of their main statements.

Remember:

> *"The actions it may perform are: Send communications to itself or to other actors.* [...]." Hewitt/Agha, p. 39

> *"Everything said is said by an observer to another observer who can be himself or herself."* Maturana, Biology of Language: The Epistemology of Reality

Paraphrase of Hewitt's statement, with Lieberman's additions, into Maturana's:

> *"Everything send is sent by an actor to another actor who can be himself or herself."*

Both theories or models start with this epistemological axiom. For both theories, the gender distinction in the axiom is of no relevance at all.

Translation for Actor Theory and Autopoiesis Theory

send::said
actor::observer
interface::structural coupling
Actor system:: living system
commmunication::autopoiesis
encapsulation :: closure
mail system
recursivity
circularity

autopoietic machine

"An autopoietic machine is a machine organized (defined as a unity) as a network of processes of production (transformation and destruction) of components which: (i) through their interactions and transformations continuously regenerate and realize the network of processes (relations) that produced them; and (ii) constitute it (the machine) as a concrete unity in space in which they (the components) exist by specifying the topological domain of its realization as such a network." (Maturana, Varela, 1973, p. 78)

Actor model of computation

Architectonic differences in the Actor Theory

Hierarchy: primitive actors –> basic actors –> meta-actors

It seems that the basic concept of the Actor Theory is not the Actor (event, message) but the *differences* between primitive / basic / meta, i.e., the architectonic distinction of different Actor types. The common trick of generalization / specification is not working. That is, a primitive actor is not simply a special case of a basic actor because the system of basic actors can not be defined consistently without the help of the primitive actors. Nor is it possible to define the meta-actor as a generalization of the common basic actors.

Thus, the differences in architectonics of the Actor Theory, up to now, is three-fold:
primitive / basic,
basic / meta- and
primitive / meta-.

Is the Actor system a theory of living systems? Or, is the theory of autopoiesis an Actor theory?

Common epistemology

All three problems mentioned obove about Actor Theory are reinstalled with Maturana's Biology of Cognition: Chiasm, Gender, Loop.

"Whereas classes are hierarchically arranged in an inheritance graph, actors coexist at the same level, without discrimination. Delegation can, however

??

Client/server model

A further antropomorphic model of interaction is the *client/server model*. Its conceptual analysis as the societal Herr/Knecht-Verhältnis, again, is done at the highest possible level of explication by Hegel and Marx. Client/server model are hierarchic organizations of task distribution. A server will not be at once a server and a client like in a heterarchic system of chiastic mediations of task distribution.

How does it work for message passing systems?

7 Everything is a Ereignis

Martin Heidegger: Ereignis ist Er-eignung

This is a key element to the theme of *Ereignis*, and what justifies its translation as "*enownment*": in the question of being the being of the questioner is sent into its "own". Ownness is not what "belongs to my essence", but what belongs to this encounter with myself, with this being sent to the space of my own truth: the space in which my essence is encountered.
http://www.linguakinetica.com/blogs/2006/03/lecture-5-feb-27th.html

In more philosophical terms, a progressive reading of the actor terminology is turning the event notion into the non-notion of Ereignis.

Everything is a Ereignis and a Ereignis is itself a Ereignis.

Postmasters universe: call, send, Ruf des Seins, Arvital Rondell

7.1 Er-eignis as Er-aügnis

Ereignis, not event or occurence, not enowing or property but "to show by opening up"

Beiträge and later works make it clear that Ereignis is not an "event" in any usual sense of the term (i.e., Vorkommnis und Geschehnis: SD 21.27) and that what Heidegger meant by Ereignis is not primarily "appropriation" or "enowning." In the forthcoming GA 71 (Das Ereignis, 1941-42) Heidegger shows that the original etymon of Ereignis is not eigen ("own," parallel to the Latin proprium, from which derive "appropriation" and "enowning") but rather eräugen/ereugen, "bringing something out into view." Heidegger got much of this from Jacob and Wilhelm Grimm.14 More importantly, however, in GA 71 (section "Das Ereignis," sub-section "Er-eigen -- Er-eignen," ms. 100a), Heidegger annotated the Grimm etymologies, thereby providing his own understanding of Ereignis. The noun Ereignis ("event, occurrence") points back to the reflexive verb sich ereignen, "to happen, occur." The etymology of the verb is quite complex; in what follows I have added the hyphens for the sake of clarity.

In GA 71 Heidegger accepts the Grimms' etymology, including the fact that eigen/proprium is not the original etymon.

Er-eignen (dasselbe [wie Er-eigen]) eu in ei -- und dazu Verwirrung mit dem unverwandten "eigen", proprium, d.h. mit "an-eignen", "zu-eignen."

Heidegger likewise accepts that the primary meaning of sich ereignen is "to come into view, to appear, to

be brought forth and revealed":

Er-eigen: er-eugen - er-äugen - ostendere, monstrare,
in die Augen, Blick, Anblick
fallen - erscheinen
sich offenbaren, zu-tragen,
be-geben.

Most significantly, he glosses all this with a verb that does not appear in the Grimms' et-

ymology. In apposition to Grimms' erweisen and erzeigen Heidegger places lichten, "to disencumber and free up, to open up or clear": [16] "lichten - erweisen - erzeigen." Thus, in the reflexive, sich erweisen and sich erzeigen ("to show up or appear as what one is") mean the same as sich lichten, "to be opened up and cleared." Sich ereignen ("to occur") means that something is brought out into the open, comes into the clear: "in die Lichtung einbeziehen." Heidegger reinforces this when he states that das Er-eigen (which he glosses as Eraigen) has the transitive sense of "lichtend - weisen" -- "to show by opening up" (in the reflexive: "to appear by having been opened up").

http://www.stanford.edu/dept/relstud/faculty/sheehan/pdf/parad.pdf

What we learn, again, is that translations of primary philosophical texts are nearly impossible. Such translations are involved in a complex accompanying of interpretations and explanations justifying the chosen translation.

Sheenan's interpretation and translation is emphasing Heidegger's attempt to surpass the Occidental paradigm of thinking by going back to a primordial and pre-Aristotelian Greek understanding of the basic philosophicall terms.

Also this endaveaur is much to ambiguous to be helpful for the desing of progamming paradigms it could nevertheless give some hints for what in an different and much less philosophical terminology is called software as "reality construction".

Leaving the antropomorphic metaphors and models

"In his discourse an observer speaks to another observer who could be himself, and *whatever applies to one applies to the other as well*." Maturana

Similar wordings can be found in the OOP literature.

This sounds liberal, in fact it is declaring Kantian liberalism, and it is better than a subordinative "class system" between actor types, but it is not enough to realize the suggested promises of a free interaction between autonomous actors.

If there is an unavoidable difference in the notion of actors we have to accept both side of the distinction and allow them to play with equal rights on stage; not as the same, but as different, accepting their difference.

To stop, circularities, at first, a tricky distinction may help: objectivity vs. (objectivity).

trans-classical logic is a system of distributed rationality

It was no surprise, but great luck, that Warren McCulloch and John W. Campell was intrigued by Gunther's theory of distributed rationality.

McCulloch Archive: Gunther, Gotthard: 1960-1966: 34 items

Again,

To sum it up: *A non-Aristotelian or trans-classical logic is a system of distributed rationality. Our traditional logic presents human rationality in a non-distributed form. This means: the tradition recognizes only one single universal subject as the carrier of logical operations. A non-Aristotelian logic, however, takes into account the fact that subjectivity is ontologically distributed over a plurality of subject-centres.* Gunther, 1962

See: Logical Parallax, Astounding Science Fiction 52:3, November 1953

As a consequence of this distribution of *"subject-centres"* and the lack of a unifying *"single universal subject"* a new kind of relationship between distributed subjects has to be elaborated. Instead of subordination under the idea of the equality of the agents, the interlocking mechanism of mediation between different agents has to be realized.

Gunther introduced differences in the notion of subjectivity and rationality which can be considered as differences in the concept of observers. But this logical distinction is not depending on a trivial gender distinction nor to a fashion based on this distinction.

In contrary, Gunther's distinction of "subject-centres" and of "distributed rationality" could give a philosophical foundation to feminism beyond bio-, socio- and anthropological constructions because it offers to make distinctions beyond logo-phallo-centrism. In fact it even had some impact.

Interaction between autarkic agents is not necessarily a process of mediation. Interaction as message passing is not considered with the epistemological and ontological status of its agents. It is presumed that they are of equal standard. The exchange happens on the level of message passing.

An actor can create another actor, but this will not be a creation out of a co-operation between different actors. Creation of actors is not super-additive.

Agents in ConTexTures

An *agent* in ConTeXtures is a subject (actor, entity) with an environment realizing that in its environment there are other subjects with their own environments containing himself in their environment. Thus, an agent as a subject is both at once an agent and part of an environment of an agent. An agent *has* an environment and *is part* of an environment. With this, a theory of agents in ConTeXtures starts conceptually not with one but with at least two agents in the game. This duplicity of agents is not defined by superposition of actors but as architectonic simultaneity of interacting agents.

Architectonics

Architectonics is prior to the quadruple of (agent, interaction, organization, environment)

and is attempting to conceptualize the structure of *togetherness* of subjects in societal systems. The architectonic conditions for agents building a societal system is not yet guaranteeing successful communication, co-operation, interaction, etc. on an informational level but is conceived as its pre-conditions.

Further more, an agent has an *inner* and a *outer* environment. In his outer environment he confronts other agents and an environment neutral to agents. In his inner environment he reflects the outer environment in its two ways as neutral and as actional. He also reflects in his inner environment his own behavior to his environment, especially to his interactional environment, that is, to other agents and their behaviors.

Coalitions of agents to societal systems are architectonically *super-additive*. The co-operation of two agents is producing a new interactional space (contexture) which is modeling the difference and mediation of the two acting agents as the realization of co-operation. Two co-operating agents are realizing co-operation as a third actional space (object, contexture).

Before actors may be able to perform communications as message passing they have to realize their mutual togetherness. In all known programming languages and computing models this is pre-given, on all levels of conceptualization and realization, by the programming designer. He/she plays the God of Leibniz Monads.

The actions they may perform are:

Build coalitions

Send communications to themselves or to other actors and at once receiving messages from other actors.

Actors in the sense of Actor Theory and OOP are reductions from interactional/reflectional mediations.

An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

Is it an Agent, or just a Program?:
A Taxonomy for Autonomous Agents
http://www.msci.memphis.edu/%7Efranklin/AgentProg.html

Composite objects

n fields

m methods

k objects

"Object-oriented programming languages support only the elementary object view. But object composition can be "simulated" [...]". Schunemann

$$\left[\begin{array}{l} contextures^{(m)} \\ \left[\begin{array}{l} \textbf{identify}\ composite-objects \\ \left[\begin{array}{l} \textbf{components}\ objects \\ \left[\begin{array}{l} \textbf{apply} \\ \left[\begin{array}{l} \textbf{method - associated - with} \\ \left(\begin{array}{l} \textbf{selector} \\ \left(\begin{array}{l} \textbf{class - of} \\ \left(\begin{array}{l} (receiving-object) \\ \{arguments\} \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

Each part of the formula is defining the kind of the system.

components: modi of composition

apply: modi of application

method: different programming algorithm

selector: modi of communication from hierarchic to net structure depending on the selector mechanism

class-of:

arguments:

Paradoxes of homogeneity

In a system where everything is one thing there is a thing which is no thing, the statement, that everything is one thing.

In a system where everything is one there is one which is another one.

In a system where everything is one there is one which is not one.

In an Actor system, the sentence *"Everything is an actor"* is not part of the actor system.

But is the sentence working if we reduce it to: "Everything in an Actor system is an actor"?

EverythingIsa: Everything is a EverythingIsa.

http://c2.com/cgi/wiki?EverythingIsa

Autopoiesis and Actors in contextural systems

Each contexture (actor, object) is an autonomous computational system like an autonomous organism. As there is no living organism in singular, contextures comes as distributed plurality. Autonomous living organism are realizing their togetherness by structural coupling. Contextures, as distributed autonomous computing units, are coupling their togetherness by mediation.

A living system is the togetherness of organisms. Disseminated systems are the togetherness of distributed and mediated computational units.

A model of a computational unit is an actor model.

A model of a disseminated computational unit is a polycontextural model.

Living system = (togetherness, organisms), organisms = (togetherness, organs)

Actor model

Actor model theory

Actor model implementation

composition vs. mediation

Is an actor system an actor?

Can the system as such be named by the system?

Self-reflectional systems are not defined as self-naming systems.

> "The interface of an actor is called its intention and defines a contract between the actor and the outside world." Masini, p. 301

Space and Place for Actors and Agents

If the locatedness for a classic actor in his midleware theater is an URL, based on URI, etc., thus a fixed identity address, then the locatedness of a contextural agent is the morphogram of such an address. The morphogram of the locatedness of an Agent is guaranteeing the liveliness of the Agent and is preventing it to be considered as a physical object. An Agent is a reflectional/interactional unit and therefore not addressable and nameable by a single and simple identity producing and identifiable name. An Agent can have a name but it isn't a name. Classic Actors are much more defined by their name and their name is used as if it would be the Actor. In this sense an Actor is a name and is not just having a name. An Actor is defined by a name-giving abstraction, i.e., the is-abstraction.

"Each actor has a unique mail address."

An Agent as having a name is not an Actor (as) being a name.

morph(URL) = LOC

> A naming service is in charge of providing object name *uniqueness*, *allocation*, *resolution*, and *location transparency*. Uniqueness is a critical condition for names so that objects can be uniquely found given their name. This is often accomplished using a name context. Object names should be object location-independent, so that objects can move preserving their name. A global naming context supports a universal naming space, in which context-free names are still unique. The implementation of a naming service can be centralized or distributed; distributed implementations are more fault-tolerant but create additional overhead. Gul A. Agha, Carlos A. Varela, Worldwide Computing Middleware
> http://www-osl.cs.uiuc.edu/

8 Object Interfaces as chiasms

Definition of a minimal object

Object1 = ((fields + methods) + identifier) + interface
Object2= ((fields + methods) + identifier) + interface
Object3= ((fields + methods) + identifier) + interface

selector = identifier

$$Object^{(3)} = \begin{bmatrix} chiasm^{(3)} \\ \begin{bmatrix} (fields + methods)^{1} \\ (fields + methods)^{2} \\ (fields + methods)^{3} \end{bmatrix} \end{bmatrix}$$

$$Object^{(3)} = \begin{bmatrix} chiasm^{(3)} \\ \begin{bmatrix} (fields + methods + selector)^{1} \\ (fields + methods + selector)^{2} \\ (fields + methods + selector)^{3} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} thematize\ chiasm(methods,\ fields)^{(3)} \\ \begin{bmatrix} contextures^{(3)} \\ \begin{bmatrix} identify\ Object^{1} \\ \begin{bmatrix} method \\ \begin{bmatrix} field \\ \begin{bmatrix} elect\ \ method^{2} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \begin{bmatrix} identify\ Object^{2} \\ \begin{bmatrix} method \\ \begin{bmatrix} field \\ \begin{bmatrix} elect\ \ field^{1} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \begin{bmatrix} identify\ Object^{3} \\ \begin{bmatrix} method \\ \begin{bmatrix} field \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

fields and *methods* are distributed locally
identifier are the address of the neighbor system
interfaces are trans-contexturally modeled as chiasm

intra-contexturally: interfaces are localized interfaces

trans-contextural

elector
intra-contextural

8.1 Hierarchic Actor construction

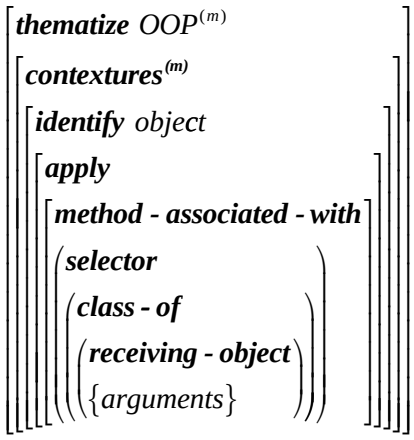
send(receiving-object, selector, argument-1,... argument-N)

apply(**method-associated-with(**selector, **class-of(**receiving-object)),
agument-1,
...,
argument-N)

$$\left[\begin{array}{l}\textbf{\textit{apply}}\\ \left[\begin{array}{l}\textbf{\textit{method - associated - with}}\\ \left(\begin{array}{l}\textit{selector},\ \textbf{\textit{class - of}}\\ \left(\begin{array}{l}\textit{receiving} - \textit{object}\\ \{\textit{arguments}\}\end{array}\right)\end{array}\right)\end{array}\right]\end{array}\right]$$

$$\left[\begin{array}{l}\textbf{\textit{thematize}}\ OOP^{(m)}\\ \left[\begin{array}{l}\textit{contextures}^{(m)}\\ \left[\begin{array}{l}\textbf{\textit{identify}}\ \textit{object}\\ \left[\begin{array}{l}\textbf{\textit{apply}}\\ \left[\begin{array}{l}\textbf{\textit{method - associated - with}}\\ \left(\begin{array}{l}\textit{selector},\ \textbf{\textit{class - of}}\\ \left(\begin{array}{l}\textit{receiving} - \textit{object}\\ \{\textit{arguments}\}\end{array}\right)\end{array}\right)\end{array}\right]\end{array}\right]\end{array}\right]\end{array}\right]\end{array}\right]$$

The attribut "selector" can be transformed into an operator **selector.** As an operator selector gets a dynamic specification.



elect-contexture (apply(…))

or

elect-contexture(send(…))
general:
sops(apply(…))

trans-contextural sops

8.2 Heterarchic dynamics

Coalitions of objects is super-additive in contextural prpogramming.

Chiasm between he-actors and she-actors as a heterarchic distribution of the "grounding" problem of actor systems.

Contextural actor systems are disseminations of "homogeneous" actor systems.

Interactionality is not first between actors but between actor systems, i.e. between contextures.

Actor systems are based on message passing, contextures on interactionality / reflectionality of disseminated actor systems.

9 The Scientific Community Metaphor for Actors

All communication in Ether is done by *disseminating* messages. There are several kinds of communication that might take place. Scientists often communicate *results* of their own researches. We will refer to these kinds of messages as *assertions*. There are other kinds of messages that must be communicated. At any time the system has certain *goals*, either to demonstrate the validity of a proposition or to find a method for solving a given problem. The goals of one part of the system must be communicated to parts of the system embodying expertise that can help achieve the goal. This communication is done with messages.

```
(DEFINE (ASSERT = x)
   (DISSEMINATE (ASSERTION x))).
```

Additionally, command fragments of the form

```
(DISSEMINATE (GOAL ⟨g⟩))
(ACTIVATE (WHEN (ASSERTION ⟨g⟩)
   ⟨COMMAND_1⟩
   ...
   ⟨COMMAND_k⟩))
```

In the Science Community Metaphor of computation two main actions are considered: *assertions* and *goals*. Between both a clear dichotomy exist: assertion / goal.

A goal is not an assertion, and an assertion is not a goal.

A sentence like *"What's your assertion is my goal and what's my goal is your assertion"* is not reasonable in this world model.

It could be introduced by viewpoints. But Kornfeld / Hewitt's *viewpoints* are not dealing with chiastic exchange relations.

The Ether examples of the previous sections did not involve any concept of *relativized belief*; in a sense, all assertions present were "believed" by the entire system. Relativized beliefs arise in accounting for the plurality of adherence in Ether. *Viewpoints* are a construct used in Ether to relativize messages as to assumptions, approaches, etc. From the sprites' point of view they represent *access points* to the messages in the system.

The "only a special case" strategy

From the epistemological viewpoint there exists a strict incommensurability between Newtonian physics and Relativistic physics. Both have, even if they use common linguistic termini like time and space, strictly incommensurable axiomatics.

From a third epistemological point of view, say in every day understanding of physics for engeneering, it is acceptable to think not only of a commensurability but even of a hierarchic order between both kind of physics. In this case, Newtonian mechanics is a special case of Einsteinian physics, for time with small velocities. *"In fact many engineering disciplines find "relativistic effects" insignificant enough that they can be safely ignored."* (Kornfeld / Hewitt). This kind of thinking is denying the epistemological differences, which appears in all terms of the theories, in favour of a nivellistic approach. Strategies of this kind seems to be necessary if one and only one contexture is accepted. If there is only one rationality and truth, the new theory has to be more general and to subsume the old into its framework. Or it has to be abandoned for ever.

Translations

Translations from one theory to another are necessary if the theorems don't fit into the simple strategy of the *special case* approach.

> ***B. Translation***
>
> **In many cases viewpoints although closely related do not inherit all information from one another. For example Newtonian mechanics is a special case of relativistic mechanics. However the frame transformations of Newtonian mechanics are not inherited from special relativity. The Newtonian frame transformations are *translations* of the relativistic ones. The following sprite translates all the relativistic transformations into their Newtonian counterparts:**
>
> **(WHEN (ASSERTION (TRANSFORMATION $= E$ (WITH RELATIVEVELOCITY $= v$)) (WITH VIEWPOINT RELATIVISTIC))**
> **(ASSERT (TRANSFORMATION (LIMIT$_{v \to 0}$ E)) (WITH VIEWPOINT NEWTONIAN))).**

Interaction between incommensurable theories

The paper *"Scientific Community Metaphor"* is referring to, at this time, new trends in history and epistemology of science developped by Popper, Kuhn, Feyerabend and Lakatos. Especiallly Feyerabend has shown that the reduction approach to the history of science is a myth. He strongly emphasized the *incommensurability* thesis and also denied the possibility of a solution in a general logical framework. His dialectics was realized in a rhetorical form only and denied any attempt to develop a new idea of formal rationality.

If theories don't share, in a strict sense, any common informations and methods, but are nevertheless in a historic and epistemological interaction, the mechanism of chiastic mediation seems to be more appropriate than other strategies.

A chiastic model of the interaction between communities is not needed if the communities, or research groups inside communities and thier viewpoints, are totally separated and strictly in "parallel". Also there is no need for chiastic mediation if the

communities or viewpoints are sharing a common pool of specific information.

Negotiation

> It has rather to do with one of those features typical for classic AI which Carl Hewitt called the problem of „logical indeterminacy“ (Hewitt 1977):
>
> What would happen, he asked, when two “microtheories”, both equally internally consistent and thus in full accord with deductive logic, lead to contrary results?
>
> In logical terms this problem is unsolvable; as Hewitt showed, it can only be settled by recourse to “*negotiation*”. This is why “negotiation” was one of the first social metaphors to gain general recognition in DAI (Davis / Smith 1983).
>
> The main thrust of DAI research, then, is directed towards overcoming the limits of individual machine intelligence by making use of distributed and coordinated problem-solving techniques. Directed towards developing programs for highly complex knowledge domains, it is based on the principles of negotiating conflict and managing dissent in an intelligent way. Since such principles cannot be invented merely by „computational introspection“ alone, it is at this juncture that sociology can step in.
>
> Thomas Malsch, Naming the Unnamable: Socionics or the Sociological Turn of / to Distributed Artificial Intelligence, January 2000
> www.tu-harburg.de / tbg / Deutsch / Mitarbeiterinnen / Thomas / unnamable.pdf
>
> http: / / www.model.in.tum.de / ~weissg / Docs / malsch-weiss-CONFLICT00.pdf
>
> ftp: / / publications.ai.mit.edu / ai-publications / pdf / AIM-410.pdf

> In logical terms this problem is unsolvable; as Hewitt showed, it can only be settled by recourse to “*negotiation*”.

This position, which Hewitt now is defending lively in WiKiPedia discussions is highly questionable.

Comments

"...lead to contrary results"

For whom, surely not for one of the involved micro-theories as such. Thus, the contradiction appears in a third, comparing, theory. At least a third position has to be involved to observe the contradiction.

Negotiations, too, take place from a different point of view. It may be the third position or an additional one.

Thus, observation of contradiction and negation, are conceived from an external observer which is not belonging to the observed system.

An assertion/goal chiasm

To boil the discussion down to elemental terms I focus on the distinction between assertions and goals appearing in different communities. At first, I will not thematize that this modeling is accepting that the linguistic terms "assertion" and "goals" are used as similar in both communities. It seems then easy to be accepted that between different scientific communities situations of chiastic interchanges between assertions and goals can arise. The wording could be *"What's your assertion is my goal and what's your goal is my assertion."*

$$
\left[\begin{array}{l}
\textbf{\textit{thematize}}\ chiasm\left(assertion,\ goal\right) \\
\\
\left[\begin{array}{l}
\textbf{\textit{identify}}\ community^{1} \\
\left(\begin{array}{l} define\ assert = x \\ \left(disseminate\ \left(assertion\ x\right)\right) \end{array}\right) \\
\left(\begin{array}{l} disseminate\ \left(goal\ \langle g\rangle\right) \\ \left(\begin{array}{l} activate \\ \left(\begin{array}{l} when\ \left(assertion\ \langle g\rangle\ \right) \\ \left\langle command_{1}\right\rangle \\ .... \\ \left\langle command_{k}\right\rangle \end{array}\right) \end{array}\right) \end{array}\right) \\
\left[\begin{array}{l} if\ \left(assertion^{1}\right) \\ \left(\textbf{\textit{elect}}\ goal^{2}\right) \end{array}\right]
\end{array}\right] \\
\\
\left[\begin{array}{l}
\textbf{\textit{identify}}\ community^{2} \\
\left(\begin{array}{l} define\ assert = x \\ \left(disseminate\ \left(assertion\ x\right)\right) \end{array}\right) \\
\left(\begin{array}{l} disseminate\ \left(goal\ \langle g\rangle\right) \\ \left(\begin{array}{l} activate \\ \left(\begin{array}{l} when\ \left(assertion\ \langle g\rangle\ \right) \\ \left\langle command_{1}\right\rangle \\ .... \\ \left\langle command_{k}\right\rangle \end{array}\right) \end{array}\right) \end{array}\right) \\
\left[\begin{array}{l} if\ \left(goal^{2}\right) \\ \left(\textbf{\textit{elect}}\ assertion^{1}\right) \end{array}\right]
\end{array}\right] \\
\\
\left[\begin{array}{l}
\textbf{\textit{identify}}\ community^{3} \\
\left[\begin{array}{l} if \\ \left(\begin{array}{l} goal^{1} \parallel goal^{2} \\ assertion^{1} \parallel assertion^{2} \end{array}\right) \end{array}\right] \\
\left(\textbf{\textit{elect}}\ assertion^{3} goal^{3}\right) \\
\left(\begin{array}{l} define\ assert = x \\ \left(disseminate\ \left(assertion\ x\right)\right) \end{array}\right) \\
\left(\begin{array}{l} disseminate\ \left(goal\ \langle g\rangle\right) \\ \left(\begin{array}{l} activate \\ \left(\begin{array}{l} when\ \left(assertion\ \langle g\rangle\ \right) \\ \left\langle command_{1}\right\rangle \\ .... \\ \left\langle command_{k}\right\rangle \end{array}\right) \end{array}\right) \end{array}\right)
\end{array}\right]
\end{array}\right]
$$

What is intended with this short contextural program is to sketch the essential mechanism of the chiasm between *assertions* and *goals* distributed over two different communities. In the third community this simultaneous interplay is modeled as a mediation between both positions and mapped into a third realization of the assertion/goal program script. In system "community3" assertion and goal can be understood as a generalization of the positions of the first two communities. On the other hand, from community3, the positions of community1 and community2 appear as possible specializations of the position realized at community3.

In other words, the generalization in community3 can also be understood as the place where a reflection on the similar use of linguistically same terms happens.

A popular example for such a situation of incommensurability and strange interactions can be found in the perennial disputation about the free-will problem produced by the neuro-scientists and the phenomenological philosophers in the framework of the mind/body distinction.

ConTeXtures:

Actor-based object-oriented programming paradigm in polycontextural constellations.

A contextural point of view is positioning a contexture. This corresponds in the conceptual graph to the uniqueness of the designed system.

"Message passing is *elemental* to computation in actors." Agha, p. 120

Togetherness is *fundamental* to trans-computation in ConTeXtures.

- Gul Agha and Prasanna Thati. An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language, From OO to FM (Dahl Festschrift) LNCS 2635. Springer-Verlag. 2004.

Carl Hewitt. The repeated demise of logic programming and why it will be reincarnated What Went Wrong and Why: Lessons from AI Research and Applications. Technical Report SS-06-08. AAAI Press. March 2006.

- Carl Hewitt: What is Commitment? Physical, Organizational, and Social COIN@AAMAS. April 27, 2006.

http://en.wikipedia.org/wiki/Actor_model#Compositionality

ability to create other actors

There are also issues more fundamental to the architecture which prove hard to realise. One of the key features of actors, their ability to create other actors as part of their behaviour, has the potential to dramatically change the state of the system at any particular time. The decisions of where to store and execute newly created actors is important to overall performance, and will require records to be kept of what is executing and where. If the system is very distributed, (e.g. the Internet) then the communications involved between different locations must also be noted, as they will affect performance drastically between remote locations.

http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/

9.1 Actors, Logic and Lambda Calculus

The development of the Actor model has an interesting relationship to mathematical logic. One of the key motivations for its development was to understand and deal with the control structure issues that arose in development of the Planner programming language. Once the Actor model was initially defined, an important challenge was to understand the power of the model relative to Kowalski's thesis that *"computation can be subsumed by deduction"*. Kowalski's thesis turned out to be false for the concurrent computation in the Actor model (see Indeterminacy in computation). This result is still somewhat controversial and it reversed previous expectations because Kowalski's thesis is true for sequential computation and even some kinds of parallel computation, e.g. the lambda calculus.

Nevertheless attempts were made to extend logic programming to concurrent computation. However, Hewitt and Agha [1991] pointed out that the resulting systems were not deductive in the following sense: *computational steps of the concurrent logic programming systems do not follow deductively from previous steps* (see Indeterminacy in computation).

http://en.wikipedia.org/wiki/Actor_model#Relationship_to_mathematical_logic

Actor system = Actors + Concurrency

Actor system = (Objects + Activity) + Indeterminancy

A shift in contextures is not deductive because deduction is an intra-contextural concept. Trans-contextural shifts are not deterministic, they are in some kind spontaneous. They happen rule-guided but are not ruled by a system of deduction rules.

An important aspect of life is that inconsistency is the norm. Hewitt

But what does "inconsistency" mean? Is it logical contradiction or is it dialectical contradiction?

Contradiction is present in the process of development of all things; it permeates the process of development of each thing from beginning to end. This is the universality and absoluteness of contradiction which we have discussed above.

Every form of society, every form of ideology, has its own particular contradiction and particular essence. Mao Tse-tung

contradiction = ñµèÇ == spear shield
ñµèÇê•ë∂ç›ç›èäóLéñìI∑¢ìWπ˝íˆíÜ;

Long before the event of paraconsistent logics, attempts to formalize dialectics and dialectical logic had been disturbing the quietness of academic research.

Direct Logic

This talk explains how Direct Logic has been developed to be tolerant of contradictions by imposing a **couple of intuitive restrictions on classical logic**. In Direct Logic, there are theories which are inconsistent but nevertheless nontrivial because irrelevancies cannot be inferred from them. Goedel's incompleteness theorem applies only to consistent theories. A contribution of this talk is to generalize the incompleteness theorem to nontrivial inconsistent theories. The upshot is that the Direct Logic consequences of (inconsistent) commitments in large software systems are undecidable even with infinite computing power.

Monday, June 12, 2006
Commitments: Inherently Inconsistent and Incomplete
At the MIT Media Lab, Carl Hewitt
http://hewitt-seminars.blogspot.com/

Direct Logic is a logic proposed by Carl Hewitt in 2006, which he conjectures to be **paraconsistent**. The goals for Direct Logic include: formalizing a notion of "direct" inference, not exploding in the face of inconsistency, support for all "natural" deductive reasoning that does not explode, and increased safety in reasoning about large software systems.
http://en.wikipedia.org/wiki/Direct_logic

Whatever *Direct Logic* may exactly be (cf. discussion at Wiki), if it is a *paraconsistent* logic, all my criticism towards it has to be repeated. And there is even some more support now for my old criticism. Hewitt's and Lieberman's distinction between proper and primitive actors is in concordance with the two-level definition of paraconsistent logics.

The logic of primitive actors is guaranteeing the soundness of the system. After that, actors may be free to produce contradictions, which are well managed by the work of the restricted houskeeping primitive actors.

There are many attempts to formalize the idea of paraconsistency which itself ranges between a hype from the 70/80s, disseminated between St. Andrews (Scotland), Australia and Brazil, to some serious philosophical and logical considerations. Hewitt's *Direct Logic* is not giving any hint from which of all these directions his logic is conceived.

Greg Restall, Paraconsistency Everywhere,
Notre Dame J. Formal Logic 43, no. 3 (2002), 147–156.
Overview: http://plato.stanford.edu/entries/logic-paraconsistent/

Actor's computational system

A computational system in the Actor Model, called a *configuration*, consists of a collection of concurrently executing actors and a collection of messages in transit [1]. Each actor has a unique name (*the uniqueness property*) and a behavior, and communicates with other actors via asynchronous messages. Actors are *reactive* in nature, i.e. they execute only in response to messages received. An actor's behavior is *deterministic* in that its response to a message is uniquely determined by the message contents. Message delivery in the Actor Model is *fair* [9]. The delivery of a message can only be delayed for a nite but unbounded amount of time.

An actor can perform three basic actions on receiving a message (see Figure1):
(a) create a nite number of actors with universally fresh names,
(b) send a nite number of messages, and
(c) assume a new behavior.
Furthermore, all actions performed on receiving a message are concurrent;
there is no ordering between any two of them.
The following observations are in order here.
First, actors are persistent in that they do not disappear after processing a message *(the persistence property)*.
Second, actors cannot be created with well known names or names received in a message *(the freshness property*).
The description of a configuration also defines an interface between the configuration and its environment, which constrains interactions between the two.
An *interface* is a set of names, called the *receptionist* set, that contains names of all the actors in the configuration that are visible to the environment.
Agha, Festschrift 2004

Summary
Computational system, configuration:
collection of concurrently executing actors, reactive, deterministic
collection of messages, asynchronous, fair
(the uniqueness property)
(the persistence property)
(the freshness property)

In ConTeXtures, "actors" (agents) can have different modi of identification (thematization), they can die or disappear (reduction), they can be cloned (replication), etc.

Again, togetherness is not message passing.

> Actor behaviors are represented as *lambda abstractions.* Delivery of a message m is simply the application of actor's behavior b to m denoted by app b m The motivation behind the actor constructs is to provide the minimal extension that is necessary to lift a sequential language to a concurrent one supporting object style encapsulation of state and procedures and coordination.
>
> Actors A Model for Reasoning about Open Distributed Systems
> Gul A Agha, Prasannaa Thati, Reza Ziaei

Conceptual modeling, Actor Model and Lambda Caculus

"The Actor Model unifies the conceptual basis of both the lambda calculus and the object-oriented schools of programming languages."

"Actors provide a unified conceptual basis for functions, data structures, classes, suspensions, futures, objects, procedures, processes, etc., in all the above programming llanguages. *The Actor Model is mathematically defined and thus independent of all programming languages.*" Open Systems, p. 153

Clinger in chapter II.2. *Global Time is Necessary* of his thesis:

"Nonetheless it turns out that some notion of *global time* is essential to any model of concurrent computation."

> **object** actor-machine
> **loop** "single input stream with temporally overlapping execution"
> read next message in input queue
> perform action concurrently with already executing messages
> actions may create new actors, send messages to existing actors, or become a next state
> **end loop**

Wegner, 1995

> There are several fundamental differences between actors and other formal models of concurrency.
>
> First an actor has a unique and persistent identity although its behavior may change over time. Second communication between actors is asynchronous and fair messages sent are eventually received. Third an actor's name may be freely given out without for example enabling other actors to adopt the same name. Finally new actors may be created with their own unique and persistent names. These characteristics provide reasonable abstraction for open distributed systems
>
> Agha, Actors A Model for Reasoning about Open Distributed Systems

9.2 Communication vs. encounter (Begegnung)

The realization of togetherness is not done by message passing, i.e., communication between identifiable actors, but as *encounter* of agents accepting the modi of being addressed.

An actor has an identifiable address, represented by an identic sign, say a number. Such addresses are avaible in the actor system as countably infinite sets of numbers.

The addresses are not localizing the actors in a topological sense, actors can move in the actor system, but they keep their identity and are not changing it. That is, strictly, they are not changing their address, wherever they may occur in the system.

The system mentioned above can easily be considered as the Internet, thus a more explicit definition of an address as an *identifier* and as a *locator* has to be given. This developed by Agha in "Middleware".

In contrast, the *existential* address or addressability of an agent, i.e., a living system, is its behavioral structure or pattern. The behavioral pattern of an agent may be recognized and "addressed" by the morphic abstraction.

Addressing by identification of a name of whatever kind is only an external possibility of objectifying an agent, that is a living system, to its uniqueness.

The addressability of a living system is made accessible by the structure of its behaviour. A living system is what it is by its way of living. And this is opening up the possibility to address it as this or that. Such an addressability is, at first, independent of space and time, i.e., of location and migration.

The question arise, as what is an agent addressed and as whom is he accepting being addressed as that?

Informational identification of an agent by its name is in contrast to existential evocation of its individuality ("Selbstheit").

An addresser has to find out how to address an agent. To know a name would not be enough because the addressed agent could deny to be addressed by this name. If an agent is defined not by its identical uniqueness but by its individuality, the agent has to decide if it accepts the modi of the addressing agent. With this possibility to decide the way of being addressed the agent has the intrinsic capability of self-protection. Say, for an attacker it is not enough to know the address of the attacked to attack it but he has to be able to get the permission, i.e., the acceptance from the attacked agent to try the attack by finding out a working address.

Individuality as the morphogram of an agent is an autonomous system closed to information processing. Its autonomy as such is not addressable by message passing. On the other side, an autonomous system, like an agent, has to interpret, i.e., to thematize his environment. If he is able to discover an informational pattern which he can interpret as an attempt to address him in a way he can accept, he can enter into the process of interaction and communication.

The individuality of a living system, understood as a morphogram, is not free in a unlimited sense, it has its complexity and complication which determines the possibility of being addressed. In other terms, the dynamism of the architectonics of the living system is realizing the framework of possible encounters and thus, communications.

This more "existential" analysis of the encountural behavior of agents has to be translated into technical terms of programming.

Obviously, this complex and dynamic manoeuvre of encounter is not excluding, at

least temporary and partially, the more common way of communication. If some of these manoeuvres have approved and found useful and safe they can be domesticated, accumulated and established as modi of communication. But there is no need for an agent to give up the flexibility of changing the communicational patterns again.

The complex interactionality and reflectionality of an agent towards his structure of encounter and the possibility of being addressed has not to be confused with the problem of what kind of referential mappings between names and agent are established. That is, encounter modi are not simply, say, many-to-many mappings between a name-pool and an agent and his repertoire of aspects.

In abstract terms, category theory is not dealing with the acceptance and rejection of morphisms between object systems.

10 The Society Metaphor and Addressability

Communication and addressability as basic terms of sociological system theory.

Fuchs, P., Adressabilität als Grundbegriff der soziologischen Systemtheorie, in: Soziale Systeme, Jg.3, H1., 1997, S.57-79

Also, inaddressability, no *representatio identitas*, no *cor et unctus*
http://hannah-arendt-hannover.de/media/fuchs_vortrag.pdf

10.1 Modern societies

Addressability in post-modern societies (Fuchs):

The UNI-queness of the address is demolished. It becomes polycontextural, heterarchic, hyper-complex, it takes the form of an endless list.

http://www.fen.ch/texte/gast_fuchs_zoegling.pdf

Addressability in modern societies as opposed to post-modern societies, obviously is unique, mono-contextural, hierarchic, hyper-complicated but of simple complexity, centralized and represented by a finite list.

Addressability is understood as a fundamental concept of sociological system theory. Depending on the society model, the modi of addressability are changing.

In sociological terms, modern societies are ruled by unique, hierarchic and global addressing systems. In this sense the mail system as a model of Actor Theory is modeled along the conception of a codified and fair addressing system corresponding to a kind of a modern society.

Its prolongation is sight as the society model of the Internet and WWW. Even the ideas of the Semantic Web are considered in the addressability mode of modern society. Some people may dream of the WEB and the Semantic Web as modeled by characteristics of post-modern societies.

First with addresses as URLs, then with semantic-ontological additions, URIs, and as a further advance the address systems of global mobile computing with its universal naming system (UAN, UAL).

Universal Actor Names (UAN) are identifiers that represent an actor during its life-time in a location-independent manner. An actor's UAN is mapped by a naming service into a Universal Actor Locator (UAL), which provides access to an actor in a specific location. When an actor migrates, its UAN remains the same, and the mapping to a new locator is updated in the naming system. Since universal actors refer to their peers by their name, references remain consistent upon migration. Agha

JAVACT: a Java middleware for mobile adaptive agents
http://www.irit.fr/recherches/ISPR/IAM/JavAct.html

10.2 Post-modern societies

In-addressability in post-modern societies

The UNI-queness of the address is demolished. It becomes polycontextural, heterarchic, hyper-complex, it takes the form of an endless list.

It is also thought that post-modern societies are *de-centralized*, *poly-centred* and excluding global *self-reference* or *self-thematization* of the system. But are involved in extensive local "self-reflection". Post-modern societies are aware about the problem of the "Blind Spot" of there reflections and observations. But donn't have a functioning theory and logic to deal with it. Modern societies in contrast are not reflecting this problem. Problems of the relation between *localism* and *globalism, internal* and *external observers,* are disturbing the epistemology of modern thinking in front of post-modern societies, too.

Communication and addressability are still the main categories of post-modern societies. For Luhmann, everything is communication between binary oppositions.

Even if the uni-queness of the address is demolished and an endless list is replacing unique addressability, this observation is not able to worry in any sense mathematical thinking of computer science. Who fears endless lists?

It seems that the society model of the WEB and the Semantic Web is running into conflicts with the way real-world actions are organized.

I have to remember, and it is still no surprise, that nearly all of these reflections about society models and addressability in sociological system theory is fully un-aware about the related work of computer scientists.

Some kind of a connection between sociological models in computer science and computing models in sociology has coming recently into the focus by the research program of "Sozionik" (Socionics, Malsch). This exception tries to close a circle between Hewitt's Society Metaphor, Luhmann's system theory and the trend of Multi-Agent Systems (MAS).

In Hewitt's terms, his society model is an *open system*. Thus, the modi of addressing inside the Actor Model are defined in the boundary of the notion of its Open System.

> "There are no global objects in Open Systems."
> "In Open Systems concurrency stems from the parallel operation of incremently growing number of multiple, independent, communicating sites."
> "Self-reference, self-knowledge, and self-development will be important capabilities for the effective utilization of Open Systems."
>
> "*Open distributed systems* are required to meet the following challenges:
> **Monotonicity** Once something is published in an open distributed system, it cannot be taken back.
> **Pluralism** Different subsystems of an open distributed system include heterogeneous, overlapping and possibly conflicting information. There is no central arbiter of truth in open distributed systems.
> **Unbounded nondeterminism** Asynchronously, different subsystems can come up and go down and communication links can come in and go out between subsystems of an open distributed system. Therefore the time that it will take to complete an operation cannot

be bounded in advance (see *unbounded nondeterminism*).

Inconsistency Large distributed systems are inevitably inconsistent concerning their information about the information system interactions of their human users." WiKi

10.3 Trans-classic societies

If this description of the addressability of post-modern societies which is in the tradition of Luhmann's system theory is correct then we have to ask which kind of society is corresponding to the morphic thematization of addressability for living systems.

The modi of encounter of agents are not determined by message passing, addressing and naming. Encounter is not following any kind of calls. Philosophically, togetherness and proximity is not listening to a call of the ultimate Being (Ruf des Seins, Heidegger).

There is no propositional model of togetherness.

If communication in societies, natural and artificial, is connected to addreassability of any kind, then the change from one addressability mode to another is not part of communication. In other words, changes in code systems are not codes. The possibility of a code system is per se not a code system. This way of thinking is not supported by post-modernism. Because they work under the halluzination that "Everything is anything". Say, everything is communication.

Different modi of being addressed by agents can be collected and resumed, condensed or mediated to a complexion of modi of addressability and recognized as a morphogrammatic pattern of possible encounter.

Morphic abstractions can be subdivided into trito-, deutero- and proto-morphic abstractions.

10.3.1 Chiasm of open and closed systems

The characteristics of Open Systems are in many sense negations and rejections of classical closed, hierarchic, centralized, etc. systems. This is well shown with the denial of the logical approach to complex systems. Concepts like paraconsistency and logical indetermination or even contradiction are all depending via refutation (negation) on the classic rationality.

Hewitt is using the strategy of *negotiation* to deal with antagonistic (sub)systems. But this negotiation is not happening on the computational level. Similar Luhmann's use of polycontexturality. He is avoiding everything which was developed by Gunther which could serve as a mechanism of mediation.

Both fear to fall back into closed and hierarchic systems of past history. But mediation is beyond the closed/open dichotomy because it is just the "transcending" mechanism of dealing with the interplay between the openness and closeness of systems. In contrast to the use of the term polycontexturality as a term with a fixed and stable semantics by Luhmann's system theory, the term as introduced by Gunther and guiding my own studies is involved in a dynamic dialectic of complex interactions between the parts of its own differentiation into poly-/inter-/intra-/trans-contexturality; and more.

10.3.2 Again, Diamond Strategies

The easiest way to understand the polycontextural approach is to accept that terms like hierarchic, closed, centralized, global, consistent, etc. are dual and even complementary to their counter-parts like heterarchic, open, de-centralized, local, para--consistent, etc. As a consequence, again, the new terms should not be considered as simple and also not only as dual, but involved in the "quadrupling" game of the Diamond Strategies. That is, for example:

closeness of closeness,	local as local,
closeness of openness,	local as global,
openness of closeness,	global as local,
openness of openness.	global as global.

As we learnt, addressability in the society model of actors is not only a question of organizing the identifiers and ruling the dynamics of message passing but also involving reflections, and self-reflection, on the systemic structure (as a society of a special kind) in which addressability can happen.

Hence, polycontextural systems of actionality and addressability are necessarily intertwined by the diamond structure of its basic terms. Obviously, the strategy of polycontextural thinking is to inherit and to accept the preceding basic distinctions and to bring them (back) into a tabular order, in contrast to its (historic) successive occurrence. What is denied is the belief in the simple succession from the old to the new terms and in its connection with a progression to the better.

10.4 Four modi of addressability

Said that, we have to distinguish at least four different types of addressability and communication according to our society metaphor and its distinctions.

First, the classic model of communication as an exchange of information (Shannon) between sender and receiver in a closed mono-contextural system.

Closed Numeric Single static address structure

Metaphor: Hierarchic command society

Second, the post-modern model of communication as message passing between actors in an open but mono-contextural system (Hewitt, Luhmann).

Open Numeric Multiple static address structure

Metaphor: Decentralized asynchronic mail delivery system

Third, the trans-classic model of interactionality and reflectionality of disseminated agents in the architectonics of polycontexturality.

Contextural Complex dynamic (open/closed) address structure

Metaphor: Direct mutual exchange between agents

Because of its asynchronous exchange mode, a mail delivery system needs buffering, say a queue. There is no direct mutual exchange. Mutuality is independent of the synchronous or asynchronous mode of message-passing. It seems that the analysis of the "direct exchange" mode is placed on a deeper systematic level than the message-passing mode. It is a structural analysis leading to a kind of a logic of exchange which is independent of the question of distribution of agents. The first full description of this kind of exchange is realized in Karl Marx dialectical analysis of the economic ex-

change. Its logic can be described by Gunther's Context-value Logics. Such a logic describes the deep-structure of exchange, while the message-passing model is more concerned with the surface-structure of exchange as message-passing.

http://www.vordenker.de/ggphilosophy/gg_struk-min-theor-obj-geist.pdf

The dynamics between different modi of addressability is itself not a mode of addressability. Such a dynamic change between modi is based on a non-addressable pattern of morphograms.

Fourth, the morphogrammatic dynamics of togetherness of disseminated agents in the architectonics of graphematics.

Morphogrammatic addressing dynamics (patterns, Gestalt)

These four models of communication are well modeled and can be incorporated into the framework of the *4 world models*, as developed in "Structuration of Interaction" (in German).

10.5 Society Model of Computation as a Global System

The Society Model of Computation is is encapsulated in a one-society paradigm. There are no concepts and techniques to interact and mediate between different societies and different society-models of computation.

This may look as a philosophical question. But it isn't. It is also not simple a multicultural approach to different cultures of different companies in a merging and fusion process.

What we understand as a society is a construction. Societies are not naturally pregiven.

10.6 Back to real-world naming strategies for actors

1.2.5 Universal Naming

Since universal actors are mobile—their location can change arbitrarily—it is critical to provide a universal naming system that guarantees that references remain consistent upon migration.

Universal Actor Names (UAN) are identifiers that represent an actor during its life-time in a location-independent manner. An actor's UAN is mapped by a naming service into a Universal Actor Locator (UAL), which provides access to an actor in a specific location. When an actor migrates, its UAN remains the same, and the mapping to a new locator is updated in the naming system. Since universal actors refer to their peers by their name, references remain consistent upon migration.

1.2.5.1 Universal Actor Names

A Universal Actor Names (UAN) refers to an actor during its life-time in a location-independent manner. The main requirements on universal actor names are location-independence, worldwide uniqueness, human readability, and scalability. We use the Internet's Domain Name System (DNS) [Mockapetris, 1987] to hierarchically guarantee name uniqueness over the Internet in a scalable manner. More specifically, we use Uniform Resource Identifiers (URI) [Berners-Lee et al., 1998] to represent Universal Actor Names. This approach does not require actor names to have a specific naming context, since we build on unique Internet domain names.

The universal actor name for a sample address book actor is:

uan://wwc.yp.com/~smith/addressbook/

The protocol component in the name is uan. The DNS server name represents an actor's home. An optional port number represents the listening port of the naming service—by default 3030. The remaining name component, the relative UAN, is managed locally at the home name server to guarantee uniqueness.

1.2.5.2 Universal Actor Locators

An actor's UAN is mapped by a naming service into a Universal Actor Locator (UAL), which provides access to an actor in a specific location. For simplicity and consistency, we also use URIs to represent UALs. Two universal actor locators for the address book actor above are:

rmsp://wwc.yp.com/~smith/addressbook/

and

rmsp://smith.pda.com:4040/addressbook/

The protocol component in the locator is rmsp, which stands for the *Remote Message Sending Protocol*. The optional port number represents the listening port of the actor's current theater, or single-node run-time system—by default 4040. The remaining locator component, the *relative UAL* is managed locally at the theater to guarantee uniqueness.

While the address book actor can migrate from the user's laptop to her personal digital assistant (PDA), or cellular phone; the actor's UAN remains the same, and only the actor's locator changes.

The naming service is in charge of keeping track of the actor's current locator.

1.2.5.3 Universal Actor Naming Protocol

When an actor migrates, its UAN remains the same, and the mapping to a new locator is updated in the naming system. The Universal Actor Naming Protocol (UANP) defines the communication between an actor's theater and an actor's home, during its life-time: creation and initial binding, migration, and garbage collection.

UANP is a text-based protocol resembling HTTP with methods to create a UAN to UAL mapping, to retrieve a UAL given the UAN, to update a UAN's UAL, and to delete the mapping from the naming system.

The following table shows the different UANP methods:

10.7 Uniqueness of the deep-structure of addressability

As it is well known and accepted, the WWW is highly complex, decentralized, ubiquitous, etc. It is also known that, albeit the contents of the WEB are more or less free, depending on political interventions, the organizational structure of the WWW is controlled by a centralized hierarchic authority.

Post-modern thinking, which fears any hierarchy, is not able to see a crucial difference between the content-structure and the organizational deep-structure of the WEB. There is no regression into old metaphysics if we understand to use the distinction between surface-structure and deep-structure of a system. The surface-structure of the WEB may well be understood in its open and decentralized way. But the deep structure, not.

It may be very difficult to see another possibility to organize the WEB than the existing one based on identity structures. The question remains, how could the deep-structure of the WEB, as an example, be heterarchic, decentralized, open, dynamic, etc.? Even if we don't have an answer to this question we should at least accept the reasonability of it.

Additional to Agha's addressability system for mobile actors, designed as a *"universal naming system"*, desires for a Semantic WWW is demanding for even more types of addresses, and a much deeper intervention into addressability than ever known, to *control the meaning* of the actions of mobile actors. Until now, message-passing was neutral to its content, with the Semantic approach also the meaning of the messages have to be addressed. This marks a crucial change of the understanding of addressability, communication and interaction in the technical and the sociological sense. Even if it doesn't work (yet) properly, this is one radical step further in the "total control" of behavior. To mention our Metaphor of Society citations we have to ask which Model of Society is involved with this step of mobile, ubiquitous and semantic possibility of control. Interestingly, the concept of control itself is still not yet deconstructed and remains taboo.

Remembering some arguments from *"Dynamic Semantic Web, §. On Deconstructing the Hype"*:

> *The Web is dynamic.*
> *The Web is massive.*
> *The Web is an open world.*
>
> *The Web is distributed. One of the driving factors in the proliferation of the Web is the freedom from a centralized authority.*
> *However, since the Web is the product of many individuals, the lack of central control presents many challenges for reasoning with its information.*
> *First, different communities will use different vocabularies, resulting in problems of synonymy (when two different words have the same meaning) and polysemy (when the same word is used with different meanings).*
> in: Towards The Semantic Web: Knowledge Representation In A Dynamic, Distributed Environment, Heflin 2001

There is no reason to deny this description at least as a starting point. Remember, the description of the *weather system* sounds very similar. But all these emphasis of the openness and decentralized distributedness of the Web is describing not much more than the very surface structure of the Web. It emphasizes the *use* of the Web by its users not the definition and structure, that is, the *functioning* of the Web. There are no surprises at all if we discover

that the structure of the Web is strictly centralized, hierarchic, non-distributed and totally based on the principle of identity of all its basic concepts. The functioning of the Web is defined by its strict dependence on a *"centralized authority"*.

If we ask about the conditions of the functioning of the Web we are quickly aimed at its reality in the well known arsenal of identity, trees, centrality and hierarchy.

Why? Because the definition of the Web is entirely based on its identification numbers. Without our URIs, DNSs etc. nothing at all is working. And what else are our URIs then centralized, identified, hierarchically organized numbers administrated by a central authority?

Again, all this is governed by the principle of identity.

"We should stress that the resources in RDF must be identified by resource IDs, which are URIs with optional anchor ID." (Daconta, p. 89)

What is emerging behind the big hype is a new and still hidden demand for a more radical centralized control of the Web than its control by URIs. The control of the use, that is of the *content* of the Web. Not on its ideological level, this is anyway done by the governments, but structurally as a control over the possibilities of the use of all these different taxonomies, ontologies and logics. And all that in the name of diversity and decentralization.

It is not my intention to deny the massive complexity of the Web and the growing Semantic Web on its surface structure. Again, remember:

The World Wide Web currently links a heterogeneous distributed decentralized set of systems.

Some of these systems use relatively simple and straightforward manipulation of well-characterized data, such as an access control system. Others, such as search engines, use wildly heuristic manipulations to reach less clearly justified but often extremely useful conclusions. […] In order to achieve its potential, the Semantic Web must provide a common interchange language *bridging these diverse systems.*

http://www.w3.org/2000/01/sw/DevelopmentProposal

Nevertheless, it is important not to confuse the fundamental difference of deep-structure and surface-structure of the Semantic Web. This fundamental difference of deep/surface-structure is used in polycontextural logic not as a metaphysical but as on operational distinction. And all the Semantic Web "cakes" are confirming it.

Beyond the layer of Unicode and URI we have to add their arithmetical and code theoretical layers. The Semantic Web Cake is accepting the role of logic, down to its propositional logic, but is not mentioning arithmetics. As we have seen in *Derrida´s Machines*, arithmetics and its natural numbers are pre-given and natural. There is not much to add. There are many possible open questions with Unicode and URI, but not with its common arithmetics.

The open question which comes back to my proposal is *"Why should the deep structure of the Web be questioned?"*. At least, it is working. A simple answer, it is not enough. There are too many problems open which cannot be solved properly in the framework of the existing paradigm.

In contrast to the complex weather system, the Web is artificial. Despite of Philip Wadler's beautiful paper about numbers and computability, *"As Natural as 0, 1, 2"*, nobody until now has given a proof that the *nature of artificiality* is of the same nature as the concept of nature in all these naturalities (naturalness), like natural deductions, natural numbers, natural computations, etc. which are the base of the deep-structure of the Web. The main problem of the increasing accessibility and usability of the Web is the fact that its corresponding deep-structure is necessarily coupled with increasing non-negotiable control.

10.8 From the Society Metaphor to the Cosmic Living Conjecture

There is no society metaphor behind the conception of Gunther's theory of polycontexturality. Neither is there an ultimate Mailer or Caller presumed.

fairness vs. liveliness

In living systems, fairness is not guaranteeing the liveliness of a living system. There are also good reasons for a living system to accept, conceptually and computationally, the principle of fairness. In bio-morph computing fairness should be replaced by liveliness of the computing system.

Marcottage and modularity (Lieberman)

http://lieber.www.media.mit.edu/people/lieber/Lieberary/OOP/Marcottage/Marcottage.html

A class is the basis of modularity and structure in an object-oriented computer program.

communication vs. togetherness

addressability vs. reflectionality/interactionality

encapsulation vs. autonomy

homogeneity vs. discontexturality

Solipsism of Actors

Addressability in the Society Model has an additional meaning, too. It postulates that the addressed actor exist, i.e., has an existence without being addressed. Such an actor can freely decide to communicate or not to communicate. It is not essential for actors to communicate. Obviously, the whole system of message-passing actors is designed for communication. But nevertheless, there is no structural need, no deep-structure of a necessity to communication for an actor to exist. This is supported, not only by the definition of actors with their interfaces, addresses and references, but also by the fact of the necessity of non-message-passing primitive actors which are not involved in communication but are necessary for the system to run.

> To avoid *infinite regress* of delegation, so-called rock bottom actors never delegate and do not send messages. They correspond to the *primitive actors* of the Hewitt's model and represent entities of a few specific types: for example, numbers, symbols and lists, in the lisp implementation. Their script is held by the interpreter." Masimi, p. 312

This may be adequate for a specific kind of computing and computation, but it seems not to be sufficient for bio-morph computing and shouldn't be used as a society model by sociologists.

Actors don't need to act. Like for Monads, there is no need for Actors to act.

This autonomity of actors is importand for modularity to work. Modularity is only working in an Actor system if it is not super-additive.

Similar arguments holds for Agent theories, like MAS and MIC.

10.8.1 Algorithmetization of hermeneutics, theory of living systems and cosmic logic

> What is urgently required is an *algorithmetization of hermeneutics*. Such algorithmetization would fall within the domain of transcendental logic. But-although cybernetics so far has completely ignored transcendental logic - the great irony is that it has willingly adopted a fateful prejudice to which the instigators of transcendental logic (Kant, Fichte, and Hegel) paid homage. It is the seemingly unshakeable prejudice that hermeneutical processes are entirely incapable of formalization.[65]

Cybernetics, here, is what we call today computer science, AI, AL, robotics, etc. It would be a misunderstanding to think that Gunther's argument would be wrong if confronted with the research, say, of artificial intelligence. AI undoubtedly made many successful attempts to "formalize" aspects of understanding, knowledge and even reflection, which are parts of "hermeneutical processes". But there is also no doubt that the underlying logic, semiotics and arithmetics was never doubted, this despite the multitude of differnet "deviant" logics, from modal, non-monotonic to paraconsistent logics, etc. used in AI research and applications

On the other hand, Kant's critique of classic logic was concerned with Aristotelian logic. It is well known that intuitionist logic and its further developments is philosophically not in the tradition of Aristotle but of Kant. But it would be a misunderstanding to identify intuitionistic logic with Kant's concept of a transcendental logic. And hermeneutics is much more different from Aristotelian logic than intuitionistic logic is.

> A dialectic theory of organism, based on the principle of conceptual complementarity, has not yet been developed. We have pointed out that cybernetics, up to now, has favored an entirely one-sided nondialectical concept of organism, obtaining remarkable but equally one-sided results.

This applies also to paraconsistent logics and its dialectical versions.

> So far Western scientific tradition has been exclusively concerned with the theory of a universe which presents to us an aspect of unbroken contexturality. The theory of such a universe is equivalent with the theory of *auto-referential objects*. Their nature was explored and so exhaustively described that we have practically come to the end of this epoch of scientific inquiry. A living organism, on the other hand, is a cluster of relatively discontextural subsystems held together by a mysterious function called *self-reference* and *hetero-referentially* linked to an environment of even greater *discontexturality*. In order to integrate the concept of discontexturality into logic we have introduced the theory of *ontological loci*. Any classic system of logic or mathematics refers to a given ontological locus; it will describe the contextural structure of such a locus more or less adequately. But its statements -valid for the locus in question-will be invalid for a different locus. To put it crudely: true statements about a physical body will not be true about the soul and vice versa.
>
> A philosophic theory of cybernetics would imply that the total discontexturality between dead matter and soulful life which the classic tradition assumes may be resolved in a hierarchy of relative discontexturalities. We repeat what we stated at the beginning: our system of *natural numbers* is valid within the context of a given ontological locus, but it is not valid across the discontexturality which separates one ontological locus from the next.
>
> The philosophical theory on which cybernetics may rest in the future may well be called an *inter-ontology*.
>
> http://www.thinkartlab.com/pkl/archive/GUNTHER-BOOK/Natural%20Numbers.html#998712

11 Peter Wegners : Everything is interaction

Both, Hewitt and Wegner, claim the limits of the Turing model of computing. Both have hopes in paraconsistent logics.

Wegner is analyzing Turing machines.
Instead of primitive actors, non-well-founded set theory.

Hewitt shows the limits of Lambda Calculus and logical deduction (Kowalski).

Interaction is everything.

Algorithmic computation is zero-interaction.

12 Niklas Luhmann: Everything is communication

"Only communication is able to communicate."
"Only communication is communicating."
"Only communication communicates communication."
"Nur die Kommunikation kann kommunizieren."

[31] Niklas Luhmann, Wie ist Bewußtsein an Kommunikation beteiligt? in: H.-U. Gumbrecht u. K.L. Pfeiffer (Hg.), Materialität der Kommunikation, Frankfurt a. M. 1988, S. 884.

Second-order turn:
Only an observed observer is an observer.

I would like to leave this exercise to the reader. You may enjoy it!

12.1 Chiasms, again

Alan J. Perlis, Epigrams on Programming

1. One man's constant is another man's variable.

SIGPLAN Notices Vol. 17, No. 9, September 1982, pages 7 - 13.
http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html

If this situation occurs simply as one or the other man's opinion then it is surely a harmless joke. If we take the epigram literally and translate the described situation into a programming context, as it may thought of, we have to introduce different viewpoints or different contextural positions into the game. We have to accept, that this difference has a strict methodological meaning, wherever it is introduced in the tectonics of a formal system. It is establishing a hierarchy which can not be reversed without producing contradictions. It also has to be recognized that both positions are correct at once. Further, a mechanism of an exchange is observed. To put all the elements together we can model this epigram as a chiasm between the involved four terms, i.e., man1, man2, constant, variable.

In other words, an object occurs which is understood at once as a *constant* and as a *variable*. This difference is realized from two different positions, representing the different men. Obviously, a constant is not a variable and vice versa. But a variable can figure *as* a constant and a constant *as* a variable. This wording is working in a system of distributed contextures. Each is giving place for a viewpoint, the realization of the terminology (variable, constant) and the mirroring of the other as the opposite position. If we would model this chiasm inside the contexture of a single formal system then we are violating the strict hierarchy between constant and variable.

Is such a situation as stated in Perlis' epigram a paradox and a non-realistic situation? From the position of contextural programming it seems to be one of the most common patterns. Simply because it is an example of a *chiastic interaction/reflection* which is basic for contextural programming.

Don't forget the gender problem

One person's constant is another person's variable.
Susan Gerhart, Microelectronics and Computer Technology Corporation

Clarifications
[December 1997] I received the following clarification from Susan Gerhart regarding the attribution of the quotation to her (thanks, Susan):
No big deal, but the quote attributed to me is a *non-gender* version of Alan Perlis' *"one man's constant is another man's variable"*. Somehow the Perlis attribution got dropped. (Susan Gerhart)

And with some more charme:
http://majagarava.blog.hr

Careful concern about gender problems has overlooked the interesting mechanism of Perlis' epigram.

Political correctness has lively proven its dullness.

13 Beyond communication

> "The purpose of any language is to communicate; that of a programming language is to communicate to a computer actions it ought to perform. There are two different sorts of objectives one can emphasize in the design of a programming language: *efficiency* 'in execution, and *expressiveness*." (Agha, 1985)

Before language can be a communicational tool it has to open up a world delivering a horizon in which statements can be expressed and meanings communicated. From this understanding of language and languaging, the world and its entities are not simply be pre-given and ready to be labeld, i.e, referenced and named.

A consultation of Google about the term "evocation" shows clearly how far away our cultural, i.e., mainly anglo-saxon, understanding of language is from an understading of language as evocation.

Hermeneuticas, grammatology and polycontextural studies have long shown the inapproriaty of this concept of language as a tool.

> Transcending Language: The Rule of Evocation
> http://www.bu.edu/wcp/Papers/Lang/LangSpad.htm#top

Kenogrammatics are not a kind of neuro-logic

> Thesis: *Neural evidence exists for predicate-argument structure as the core of phylogenetically and ontogenetically primitive (prelinguistic) mental representations. The structures of modern natural languages can be mapped onto these primitive representations.*

> The brain having a complexity far in excess of any representation scheme dreamt up by a logician, it is to be expected that the basic PREDICATE(x) formalism is to some extent an idealization of what actually happens in the brain. But, conceding that the neural facts are messier than could be captured with absolute fidelity by any formula as simple as PREDICATE(x), I hope to show that the central ideas embodied in the logical formula map satisfyingly neatly onto certain specific neural processes.
> James R Hurford, The Neural Basis of Predicate-Argument Structure
> http://www.ling.ed.ac.uk/~jim/newro.htm

Contextural interactionality/reflectionality is not sharing any kind of data between its interacting agents.

What could be called a common property of contextural systems is the fact of a mutual offer to give space, locations, inside the interacting agents, to each other, and to themselves.

On the structural level of enabling communication which is in the focus of contextural programming, questions of sharing information and of buffering communications are not yet to be addressed.

Peter Fuchs, Vom Zögling zum Formen-Topf

Die Adresse der Erziehung – weltgesellschaftlich, 2006

"Die EIN-heit der Adresse wird gesprengt. Sie wird polykontextural, heterarch, hyperkomplex, sie nimmt die Form einer unabschließbaren Liste an.

Die soziale Adresse ist, wie wir sagten, nicht die Eigenschaft von Menschen, sondern eine im genauesten Sinne: soziale Struktur. Sie realisiert sich nicht neben der Gesellschaft, sondern in ihr und mit ihr, und insofern kann sie nur die Form haben, die in der jeweiligen Gesellschaft und ihrer Differenzierungsform möglich ist. Man kann daher beispielsweise von einem archaischen Adressenformular sprechen, von einem Zentrum/Peripherie-Formular (etwa dem der Großreiche), von einem Formular der Stratifikation und eben auch: von einer für funktionale Differenzierung typischen Adresse.

Wenn wir uns auf funktionale Differenzierung beziehen, bekommen wir es mit überaus denkwürdigen Verhältnissen zu tun. Zunächst fällt auf, daß diese Gesellschaft selbst wie ihre Funktionssysteme keine eigene Adresse hat. Keines dieser Systeme ist ansteuerbar, mit keinem dieser Systeme kann Kommunikation aufgenommen werden, weil sie nicht über einen *cor et punctus* verfügen, der sie repräsentiert. Man kann keine Botschaften an die Gesellschaft richten, nicht an das Recht, die Wirtschaft, die Wissenschaft, also auch nicht: an die Erziehung. Oder anders ausgedrückt: Da gibt es keine Instanz der Zurechenbarkeit, keinen Ort in solchen Systemen, von dem aus gleichsam ‚legale' oder gar erschöpfende Selbstbeobachtung und Selbstbeschreibung möglich wäre. Der Begriff, der diese Denk- und Merkwürdigkeit bezeichnet, ist: Polykontexturalität.

Sie fügt sich nicht den Einheits- und Autonomieerfordernissen der Erziehung. Statt dessen läßt sich schnell sehen, daß das Adressenformular der Moderne die spezifischen ‚Erwartungscollagen' der Erziehung assimiliert, oder besser: sie hinzufügt zu dem, was man nicht mehr eine einheitliche Adresse nennen kann, sondern eher als so etwas wie eine polykontexturale, heterarche, mitunter hyperkomplexe Liste auffassen muß. In gewisser Weise gleicht das ‚Auslesen' dieser Liste dem Lesen eines Buches, bei dem der Autor laufend vergißt, worum es eigentlich geht, oder besser: Es gleicht dem Auslesen einer ‚Zeitung'.

Es ist nicht mehr die EINS im Jenseits der sozialen Systeme, sondern selbst, wenn man es in einem ungewöhnlichen Bild sagen darf: ein Implex. Oder jedenfalls ‚etwas', das auf sich selbst nicht *einen* Zugriff hat, sondern viele *und* inkompatible, und: keinerlei Zugriff auf *den* ‚Zugreifer', den Beobachter.

Aber welche Wörter, Bilder, Begriffe man hier auch wählen mag, im Ergebnis hieße dies alles: Unter Bedingungen der Hochmodernität ist die erziehungstypische Adresse (die Eins des Adressaten, Individualität, Autonomie, Selbstreferenz etc.) in gewisser Weise nur noch ein Phantasma."

http://www.fen.ch/texte/gast_fuchs_zoegling.pdf

Fuchs, P., Adressabilität als Grundbegriff der soziologischen Systemtheorie, in: Soziale Systeme, Jg.3, H1., 1997, S.57-79

inaddressability, no *representatio identitas*, no *cor et unctus*

http://hannah-arendt-hannover.de/media/fuchs_vortrag.pdf

Addressability in post-modern societies:

The UNI-queness of the address is demolished. It becomes polycontextural, heterarchic, hyper-complex, it takes the form of an endless list. (Fuchs)

As we know, "no" isn't enough. And who likes lists?