Colored-Object Programming : about the Programming Activity

Henry J. Borron

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Colored-Object Programming : about the Programming Activity

Henry J. Borron

N° 2880
Avril 1996

THÈME 2

Colored-Object Programming : about the programming activity.

Henry J. Borron *

Programme 2 — Génie Logiciel et Calcul Symbolique
Action LeTool

Rapport de Recherche N° 2880 — Avril 1996 — 32 pages

Abstract. This paper discusses the cognitive aspects of the programming activity in "colored object programming", a refinement of object oriented programming.

The paper first reports works done by cognitive psychologists about object oriented programming and browes a number of tools and researches meant for overcoming the difficulties encountered by programmers.

The salient features of colored object programming (ex. : visual formalism, simulation capability, declarative programming, ...) are then analysed from a cognitive point of view. Color graphs appear to have distinct advantages for abstracting essential properties and capturing informations that were left implicit in traditional object oriented programming. Besides bringing progresses in information structuring and layering, colored object programming is notably expected to better support the opportunistic programming activities.

Keywords. Cognitive psychology, opportunistic planning, cognitive dimensions, visual formalism, language design, color graphs, object oriented programming, state, transition, inheritance, combination.

(Résumé : tsvp)

* borron@chris.inria.fr

INRIA, Unité de Recherche de Sophia-Antipolis
2004, route des Lucioles. BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 — Télécopie : (33) 93 65 77 65

Programmation par Objets Colorés : à propos de l'activité de programmation.

Résumé. Ce papier discute les aspects cognitifs de l'activité de programmation en "programmation par objets colorés", un raffinement de la programmation par objets.

Le paper commence par rapporter les travaux effectués par les psychologues cognitivistes concernant la programmation par objets et décrit un certain nombre d'outils et de recherches visant à pallier aux difficultés rencontrées par les programmeurs.

Les traits saillants de la programmation par objets colorés (ex. : formalisme visuel, possibilité de simulations, programmation déclarative, ...) sont ensuite analysés d'un point de vue cognitif. Les graphes colorés apparaissent comme ayant des avantages particuliers pour abstraire des propriétés essentielles et capturer des informations qui étaient laissées sous forme implicite en programmation par objets traditionnelle. En plus des progrès en structuration et en ordonnancement par niveaux des informations, il est notamment attendu de la programmation par objets colorés un meilleur support des activités de programmation opportunistes.

Mots-clés. Psychologie cognitive, planification opportuniste, dimensions cognitives, formalisme visuel, conception de langage, graphes colorés, programmation par objets, état, transition, héritage, combinaison.

Colored-Object Programming: About the programming activity.

"Computer Programming is many faceted. (...) It is an art. Fine programs are the result of more than routine engineering. They require a refined intuition, based on a sense of style and aesthetics that is both personal and practical. As an artistic medium, programming is highly plastic, unconstrained by physical reality. In programming, perhaps more than in any other arts, less is more. Simplicity is nowhere more practical than in programming, where the bane is complexity. When just the right abstraction for a problem has been found, it may be a thing of beauty." [Springer-Friedman, 1989], pp. 3-4.

1. INTRODUCTION

Colored object programming (COP for short) is a refinement of object-oriented programming (OOP) : the answer of a colored object to a "message" depends not only on its class, but also on its current state. A class is abstracted as a whole in the form of a graph, termed color graph, describing possible instance states and transitions between them. In a preceding report [Borron, 1996a], we argument the design of the visual formalism from an ergonomic point of view. As a complement, this article more generally considers the activity of the programmer, relates COP to the work notably done by cognitive psychologists, and describes difficult points and expected results.

Next sections are organized in the following way : first, we focus on OOP and its difficulties (section 2), then we browse a number of tools and researches meant for overcoming the OOP difficulties (section 3). In section 4, we present the salient features of COP notably with respect cognitive activities, style and methodology. Appendices A to F contain much more details.

2. OOP : A DIAGNOSTIC

2.1 Problem : an absence of deep structuration.

"The art of programming is the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible" wrote Edsger W. Dijkstra in the late sixties in his Notes on Structured Programming ([Dahl-Dijkstra-Hoare, 1972], page 6).

Structured programming and, more recently, object-oriented programming with its emphasis on reuse [Lewis-Henry-Kafura-Schulman, 1991] were milestones in coping with the intrinsic difficulty of programming. Both can be interpreted as a successful seek for structure. Our purpose is not to draw the borderline between the two and we shall not attempt to[1]. Rather, we recognize the existence of useful structuring mechanisms at different levels : at the (rather low) levels of control structures and data structures, as well as between relatively large stable units of code (classes) via meta and inheritance links.

We note, however, that the intermediate scale level has remained largely unaltered by the advent of OOP : the set of methods of a given class is still organized as a simple list, or a list of lists as with the notion of protocol supported by Smalltalk [Goldberg-Robson, 1983]. The concept of (programming) interface is accordingly restricted to a list of method names with corresponding formal arguments, exactly like in antiquated programming languages. This absence of deep structuration may astonish who seeks order whenever possible.

2.2 This problem is part of a larger problem.

This absence of deep structuration does not yield a single problem per se, but rather participates to a syndroma : whereas OOP is intended to ease code reusability and extensibility, many a programmer –even an experienced one– has encountered difficulty in locating, comprehending and modifying code.

---

[1] Structured programming should not be perceived as being restricted to the sequencing discipline proposed by Dijkstra in his already cited monograph (decomposition by concatenation, selection and repetition). The real concern of his monograph is in fact the composition of large programs and their correctness. Interestingly enough, the same book contains also "Notes on data-structuring" by C.A.R. Hoare and "Hierarchical program structures" by Ole-Johan Dahl and C.A.R. Hoare, a monograph based on SIMULA 67 which early featured a number of object-oriented concepts.

A number of papers reports the problem : [Bellamy-Carroll, 1990] shows the difficulty and time spent to locate code in a traditional OO system (Smalltalk-V), leading *"all too often programmers find that the search process is so complex that it is easier just to reinvent a class and method"* (page 4) ; about a single-subject study, [Lange-Moher, 1989] observes :*"reuse through inheritance and source code borrowing was the overwhelmingly predominant style, and reflected an overall approach of comprehension avoidance, in spite of the fact that the subject was modifying the code that she herself had written earlier"* (page 69)[2] ; the same copying practice was also observed by [Détienne, 1990a] (both for novice and experienced programmmers). These observations are consistent with studies about reuse, for example : [Woodfield-Embley-Scott, 1987] tells the difficulty encountered by programmers untrained in reuse in deciding when to reuse abstract data-type code versus when to recreate it ; observing the reuse of Jackson (JSD) specifications, [Sutcliffe-Maiden, 1990] notes a *"mental laziness"* consisting *"in copying rather than reasoning"*...

Another problem is linked to the program design activity : the hierarchical structure of classes in a OOP library is more easily developed top-down than bottom-up or opportunistically. *"Early studies used small problems and did not look at expert design behavior on realistically large tasks"* and consequently *"presented results without explicitely challenging the traditional model of top-down software design"* ([Curtis, 1990], p.xxxv). Since 1990, a number of studies have better characterized the design process as being at least partly opportunistic [Guindon, 1990.a] [Guindon, 1990.b] [Davies, 1991.a]... Concerning reuse in OO design, [Burkhardt-Détienne, 1995a] observes experts to have a top-down approach *"to expand their current solution"* and a bottom-up approach *"to link the previously defined entities through a new elicited (to-be-reused) entity which was more abstract"*.

3. OOP : SEARCHING SOLUTIONS

3.1 Tools

Various support tools have been built to remedy these problems, mainly to improve the information accessibility and the usability of browsing tools, for example :

- the Browser and Navigator companion tools of Objtalk make the usual inheritance-based information easier to retrieve, by *"letting users locate resources without knowing and remembering names"* [Fischer, 1987] ;

- the improved (*"case-based"*) Smalltalk-V environment proposed by [Bellamy-Carroll, 1990] : its context-sensitive browser shows only the classes and methods used in a selected work activity (classes showing the hierarchical context are greyed out) ;

- Track, a visual tool tightly integrated into the Smalltalk-80 programming environment, for observing messages between communicating objects at run-time, for intercepting these messages and discovering their contents [Böcker-Herczeg, 1990] ;

- GraphTrace, a system providing several perspectives on the dynamic behaviour of an OO system by displaying and concurrently animating different views of it, either structural (ex. : the *"progeny"* and *"ancestry"* graphs) or behavioral (ex. : the *"method invocation view"*) [Kleyn-Gingrich, 1988]. According to the authors, this is*"a powerful means for understanding"* OO systems.

3.2 Cooperation with cognitive psychologists

Cognitive psychologists now cooperate in system designs. Besides the Esprit 3 SCALE project [Oquendo et alii, 1994], let's mention two such cases :

- the (already cited) improved Smalltalk-V environment [Bellamy-Carroll, 1990] is based on psychological claim analysis [Carroll-Kellogg, 1989]. A claim describes both the positive and negative cognitive consequences of system features. An example of a claim is : *"Finding code by browsing a large library of code supports serendipitous search of classes and methods, but distracts from the original task and can lead to disorientation"* ;

- in relation with the Solve language [Roberts-Wei-Winder, 1990], [Green-Gilmore-Blumenthal-Davies-Winder, 1992] considers the triple issue of locating, comprehending and modifying OO code, and proposes to provide a *"description level"*, i.e. a type of knowledge base, *"to record both transient facts and long-term facts about program components and program relationships"*. (Real experts are effectively known to be able to motivate their workings in great depth [Petre-Winder, 1988] and this type of information is not captured into the actual code. Externalizing memory is of

---

[2] As an implementer, we note however that the task (building a demonstration), the alloted time (one month) and the environment (a small company) constrained the programmer in such a way that she might have decided not to produce a polished software and to privilege rapidity above all.

great value for comprehending and locating code, but also –dynamically– for modifying it in difficult situations[3], i.e. when the modification process puts a prohibitive load on the programmer's working memory.)

Cognitive psychologists are interested in these difficulties because most of the problems lay at the cognitive level and also because of the complexity of OOP systems : *"Designs that are successfull in small-scale systems often fail in a larger environment, because it turns out that they place demands on long-term and working memory, and on intellectual resources"* [Green-Gilmore-Blumenthal-Davies-Winder, 1992].

4. COP : ITS SALIENT FEATURES

4.1 A better support for opportunistic programming activities

As previously mentionned, the design process is now thought as being at least partly opportunistic [Guindon, 1990.a] [Guindon, 1990.b] [Davies, 1991.a]. [Green, 1989] also stresses that *"it has repeatedly been shown that users prefer opportunistic planning rather than any fixed strategy such as top down development"*[4], and shows that the *"three strong corollaries"* that follow this observation (structure perception, multi-level definitness, fluid modification) are not always well supported by traditional OOP especially when considering the inheritance hierarchy (premature commitment, viscosity) and the relationships between methods (limited role expressiveness)[5].

Conversely, three important characteristics of COP seem to be, for each class :
- the emergence of a structure, the color graph, relating the occurrences of messages (specifying their conditions of acceptance (states) and their effects (next states)) ;
- the separation between two different levels, a specification one (the color graph expliciting states and transitions) and an implementation one (methods and memory represenations, possibly defined in a hierarchy of classes).
- the abstract specification of instances behaviour in a single place, the color graph (no class hierarchy to take into account at the specification level) ;

Thus, in first analysis, both above specific criticisms about OOP are to be lessened. While structure perception and multi-level definitness are clearly improved, fluid modification is more difficult to talk about (it depends on the modification to be done and on the programming support).

In this respect, the most interesting property of COP is thus to offer in one place, the color graph, a complete abstract specification of the behaviour of the instances of a class (cf. Appendix B). In practice, this means :
(1) that the specification is stable regarding whatever implementation is chosen ;
(2) that the specification is stable regarding the class hierarchy ;
(3) that the specification is independent of the strategy used for developing the class hierarchy (top-down, bottom-up or opportunistic).
For example, the programmer may first decide to implement the behaviour in question in a flat way (in a single class) and/or in an unoptimized way, and then change his/her mind, using a sophisticated hierarchy of classes and/or a sophisticated implementation : the abstract description of the class (the class color graph) will not be modified (if the behaviour of the class is not altered, of course). The clients of the class (i.e. the other classes having instances that send messages to the considered class instances) are not altered by the modifications of the considered class implementation. The effect of a modification has a much narrower scope than usual in OOP and is thus much more easily mastered. The programmer may decide to define the color graph of the class from scratch, or as a more or less complex composition of pre-existing color graphs (top-down process) ; he may also decide to factorize the color graphs of similar classes into a new class color graph (bottom-up process) : all these operations will not change a color graph. The clear distinction between the specification of a class (the class color graph) and its implementation is thus quite valuable.
An appropriate methodology will take advantage of this distinction : designers will be encouraged to define color graphs and to extensively simulate the system behaviour (see subsection 4.4) ; low cost implementations will also be encouraged ; afterwards, the system implementation may progressively evolve so as to obtain, for example, a better response time... Even when the system is supposed to be fairly stable, the effects of (specification or implementation) modifications will be easier to track than usual in OOP. (Of course, the underlying system does not impose a particular order for defining color graphs and their implementations.)

The thiner granularity of code (micro-methods) is also beneficial to the programmer : the context to take into account is narrower than in traditional OOP (as it was when passing from traditional programming to OOP) ; each grain is easier to build and less fragile to modifications. (Of course, this has a drawback : the complexity due to the number of grains,

---

[3] This is considered as a tentative point by the authors who note that *"There has been very little investigation from cognitive scientists of techniques to ease the modification of code or other design material"*.

[4] The ordering of mental steps bears little relationship to the hierarchical structure of the final code. See [Rist, 1986].

[5] "premature commitment", "viscosity" and "role expressiveness" are cognitive dimensions [Green, 1989] [Green, 1990] [Green-Borning, 1990].

i.e. micro-methods, to be managed. Our guess is that the proposed change is globally beneficial because the color graph concept appears as a superior tool for organizing this complexity. The inheritance rules we propose are intended to be simple to use. See Appendix F about the properties required from a linearization algorithm.)

Finally, the abstract implementation capability is another positive aspect : the user can express a new class as a combination of existing classes using their specifications, thus without being obliged to consider their whole hierarchies as it is the case in OOP. (The combination may include an increment but the remark is still valid.) This abstract implementation capability eliminates (when appropriate) the need to examine the code, mimicking in some way –at a more abstract level– what programmers do (*"avoidance of comprehension"* strategy). For more details, refer to [Borron, 1995b] [Borron, 1996d].

4.2 A uniform and sound methodology

4.2.1 object and state typologies

In an empirical study of the software activity with the C++ language, the paper [Burkhardt-Détienne, 1995b] finds out that *"there were more static mental representations than dynamic representations"*. the formers involving *"static statements like states and properties (e.g. object-based representations)"* and the latters, the process of *"simulating and enacting a mental model (e.g. control flow based representations)"* (section 3.3.2). Subsection 4.4 will focus on the COP simulation capability ; this one is about states.

[Aksit-Bergmans, 1992] presents an overview of methodologies (termed methods in the paper) and obstacles in OOP. In section 4.2.4 (p. 350), one can read : *"Most methods (...) consider states as an important aspect of object-oriented software development. States are used to capture the dynamic behavior of systems, and are also used as a means for identifying operations of objects. In general, a state represents the condition of an object at a certain moment. (...) Although these methods consider states as an essential issue in object-oriented programming, they do not address the integration of states with inheritance. (...) Only OMT (...) considers the issues of generalization and/or specialization of state specifications as significant. Specialization of state diagrams is partially possible. (...) We claim that a notation for the specification of state diagrams should be suitable for extension by subclasses. (...)"*.

COP provides such a mechanism. In a first step, it exhibits inheritance rules inside the color graph of one class ("local inheritance"): such a graph represents instance states and transitions between these states independently of any class hierarchy. In a secon step, the local inheritance rules are extended so as to take into account the lattice structure of classes ("class inheritance"). The proposed mechanism is called inheritance by dimensions [Borron, 1996d] : it corresponds to interpreting color graphs in a N-dimensions space. Two important operations are proposed for creating new classes from existing ones. The first one, termed the **composition**, makes a new classes inherit from several (base) superclasses : it basically composes the inherited color graphs ; the second one, termed the **derivation**, builds a derived class from a base color graph and a mixin one. In the first case, all the dimensions are combined[6] (cartesian product) ; in this second case, the extension corresponds to the addition of one or several dimensions in one point of the base space. (For more details, refer to the Appendix C : the first example corresponds to a composition ; the second one, to a derivation.)

Besides overcoming a serious technical problem, COP induces a fairly noticeable change in the way programming may be analysed (and teached). The most important aspect is the definition of typologies around which all the programming activity is organized (state typologies in addition to the traditional class typology). In this form of programming, explicit test control structures disappear (and loop control structures as well). (For more details, refer to the Appendix A.)

Concerning "plans" ([Soloway-Erlich, 1984] [Rist, 1986]...), we wonder about the possible effect of our proposal : because a new level of programming (the color graph) emerges, may observations at this level bring useful informations, possibly at a higher level than currently ?

4.2.2 purely declarative programming

A question which is somewhat disturbing in traditional OOP is the existence of a non-modular construct (termed **call-next-method** in CLOS, **super** in Smalltalk, ...). Reference [Keene, 1989] expresses this worry in clear terms (section 5.8, p.111 : *"Guidelines on controlling the generic dispatch"*). While the declarative technique lets the user *"predict the order of methods without looking at the code in the bodies of the methods"*, the imperative technique is *"in a sense (...) a violation of modularity"* and should thus be used *"only when that power is truly necessary"*. Unfortunately, *"some programs"* cannot be written *"without resorting to the imperative technique"*.

As explained in [Borron, 1995b] [Borron, 1996d], COP gets rid of this common non modular OOP construct. The combination mechanism thus gets fully declarative instead of being partially declarative (method qualifiers) and

---

6 Here, we suppose the inherited dimensions are all distinct.

partially imperative (**super** or **call-next-method**). No need to look inside method bodies[7] to understand how methods are ordered at run-time. Sophisticated method combinations are nevertheless possible. For example, one can used **:before**, **:after** and **:around** method qualifiers to mimick the CLOS standard method combination. (See Appendix E for more details.)

4.2.3 no arrays

[Mills-Linger, 1986] is an apparently indirectly related paper. It claims for avoiding arrays and pointers and constraining collections to sets, stacks and queues. This claim parallels the one for avoiding GOTO's and using disciplined control structures in place. *"As an evidence of this approach, a system of programs of some 20 000 lines without arrays"* (worth an equivalent program of 95 000 lines) was developed. It proved much easier to check than it would have been otherwise and *"operated for a year with no errors detected"*. The authors also claim that this data access discipline can be taken adavantage of by an efficient compiler. Our proposal makes clear that arrays are quite disturbing for the programmer who is used to (too many states to handle, hence a cognitive load which may be the cause of errors). While the use of arrays is not theoretically forbidden (decompositions and conjuctions made them possible if adapted to indices), our methodology certainly discourage their use. We think this is a good thing and, given the above cited successful experiment, we do not intend to propose indexed constructors.

4.3 A visual formalism

4.3.1 Introduction

In 1977, the largest existing Smalltalk program was *"Pygmalion, a computer program to model and stimulate creative thought"* using a visual interface [Smith, 1977]. His author motivated it in length, quoting –among others– a survey about the creative methods of mathematicians done in 1945 [Hadamard, 1945] : it appears that the mental pictures of such distinguished scientists were visual and/or kinetic.

Einstein, replying to the survey, wrote : *"The words of the language, as they are written or spoken, do not seem to play any role in my mechanism of thought. The physical entities which seem to serve as elements in thought are certain signs and more or less clear images which can be 'voluntarily' reproduced and combined... Taken from a psychological viewpoint, this combinatory play seems to be the essential feature in productive thought –before there is any connection with logical construction in words or other kinds of signs which can be communicated to others. The above-mentioned elements are, in any case, of visual and some of muscular type. Conventional words and other signs have to be sought for laboriously only in a secondary stage, when the mentionned associative play is sufficiently established and can be reproduced at will"*.

Jacques Hadamard, in charge of the survey, concluded : *"My mental pictures are exclusively visual. (...) Practically all [the polled mathematicians] (...) avoid not only the use of mental words but also, just as I do, the mental use of algebraic or any other precise signs. (...) The mental pictures (...) are most frequently visual, but may also be of another kind –for instance kinetic"*.

[Larkin-Simon, 1987] acknowledges this fact and shows why it is advantageous to use diagrams. They display important informations that are otherwise only implicit in textual representation and very costly to compute from it. A study reported in [Scanian, 1989] comforts this view. Hence, our intent to provide a visual formalism and to tune it so as to fit cognitive aspects in the best possible way.

4.3.2 Color graphs

In our view, the most important cost in traditional OOP code is due to the impliciteness of states (cf. subsection 4.2.1). The color graph formalism evacuates this problem. Central to the programming environment, it can be used at editing time as well as at debugging time. This uniformity is an important advantage for the user and can be taken advantage of for an integrated simulation tool (see next subsection). At present, much work remains to be done for designing a nice and useful programming environment. In this respect, an analyis in terms of cognitive dimensions [Green, 1989] [Green, 1990] will certainly be beneficial.
As part of the tuning of the formalism, it appeared useful to study the respective merits of connectedness vs. insideness, i.e. the gains and drawbacks due to transforming color graphs into "color higraphs", a variation of the higraph formalism proposed in [Harel, 1988] and related papers. Hence the paper [Borron, 1996a], devoted to the color graph visual formalism and to the study of its variations. Even if the color graph formalism presently appears stable in it basic features, experiments and expected feedbacks on our proposal are likely to induce a better tuning of supplementary features (like super-blends, negative transitions, constrained subtrees : see [Borron, 1996a]) ; the design of the programming environment should beneficiate from experiments and feedbacks too.

---

[7] of primaries and around-methods

4.3.3 Other representations

Beyond personal tastes and habits, a formalism may better express certain structures inherent in the underlying data while masking others. We thus do not exclude other useful representations. Class hierarchies offer an obvious example. Our report on inheritance [Borron, 1996d] also proposes additional representations for methods (dimension diagram, invocation state diagram). As noted by ([Chatel-Détienne, 1994], section 5.4) about a study concerning the Smalltalk language : *"It is important to provide visualisation of programs which support (...) various strategies [used by experts] : a representation of the message passing graph for the procedure-centred strategy, and a representation of the conceptual schema for the object-centred strategy and the function-centred strategy"*. This excerpt corresponds to our experience and also to a remark previously made in [Détienne, 1990a] concerning a particular language for an object-oriented data-base : the author proposes (p. 72) to articulate representations respectively with the code structure, the data flow and the control flow. (The point here is not about the medium, visual or not, but about the type of information to be shown.) One can also relate the remarks made here with the tools evoked in subsection 3.1.

4.4 A simulation capability

Color graphs support simulations. This capability seems important from several points of view. A color graph acts as a model that a programmer (or program designer) may animate in his/her own mind. A color graph may also be animated at run time and at editing time : in this latter case, depending on the informations provided by the programmer, the animation may be more or less automatic. We propose to extend the automatic simulation capability to hand-made actions : this should help the user to explore and evaluate possible interactions between objects. By storing these actions, the underlying system will provide to the programmer a useful extension to his/her own memory. These actions will not be lost once the simulation is completed, but can be –on user's demand– transformed into actual code, thus providing an alternative to the standard writing of method bodies. (This proposition is much like the capability of dynamically defining macros under the Emacs editor.) The visual aspect is supposed to help programmers (at least, some of them).

This proposition directly derives from [Guindon, 1990.b] as an attempt to overcome the users's difficulties described in the following excerpt (p. 339) : *"(...) designers found it difficult to simulate the interactions between components of the system, the behavior of a component if it extended over many steps, or the behavior of a subsystem calling centrally embedded susbsystems. To help mental simulations, designers often resorted to diagrams. However, because diagrams were a poor medium to represent changes in location and time, they were not sufficient to prevent all simulation breakdowns*[8]*."*

According to this author, simulations play an important role in high-level software design [Guindon, 1990.a] : designers rely heavily (1) on *"problem domain scenario simulations [for] the discovery of new requirements and of partial solutions in various points and abstraction levels in the solution decomposition"* (p. 288) and (2) on *"simulations of their design solutions"* so as to discover various inconsistencies and incompletenesses (p. 291). Concerning the first type of simulations, Raymonde Guindon cites [Larkin-Simon, 1987] as we do in the preceding subsection ; concerning the second type of simulations, she stresses the role of notations and notes that *"the mental simulations were shallow, that is, most were restricted to one level of abstraction and one subsystem (...) [given] the severe limitations the working memory poses on the processing of embedded structures"*. She confirms an observation made earlier by [Kant-Newell, 1984] : *"there were two types of simulations : executions with specific tests cases and symbolic executions using variables instead of specific cases"*. Among an initial list of problem solving and design heuristics proposed by Raymonde Guindon, appears the following : *"(3) Identify system functions that can be performed nearly independently and divide the system into corresponding subsystems"*. One may note the color graph formalism matches well this third heuristic through the support of (nearly independent) dimensions (cf. Appendices C and D.). Following [Guindon, 1990.b] and [Adelson-Soloway, 1988], [Burkhardt-Détienne, 1995b] also stresses the importance of the simulation activity (section 4) and relates simulations to the more general framework of the relationship between mental/situation models (considered as equivalent terms by the two authors) and analogy (section 3.1). Hence, we consider as an important aspect the development of a powerful and easy to use simulation tool.

4.5 A practical programming support

Although the goal of this paper concerns the cognitive aspect of COP programming, it is probably worth to mention the above benefits are obtained without loosing in efficiency or reliability.

4.5.1 efficiency

Method dictionaries are attached to colors and pigments. A color or pigment change has a cost : a pointer change. On the other hand, due to the precise specification of transition destinations, a COP implementation will only evaluate the

---

[8] Note the observation was made about the lift control problem : this problem involves concurrency, a feature not supported –for the moment– by COP.

conditions that really need to be evaluated (thus saving unnecessary tests an OO implementation may have to make because methods do not specify a destination) ; a careful implementation may also produce optimized code by taking into account the partitionning property of the destinations of a selection (one test is meant to succeed).

4.5.2 reliability

By construction, a color graph prevents from the use of non applicable micro-methods (for example, the *surface* method of the *Circle* class is not applicable to an unitialized *Circle* instance). To obtain the same result in traditional OOP, the programmer must program in a defensive way or make sure (by pre-compiling states in his/her own mind) that a method is systematically applicable in the application in question. This advantage of COP vs. OOP is true when specifying a program for the first time ; it is even more valuable when a program gets modified (constraints are explicit in the program itself instead of being (or, more probably, having been) explicit in the programmer's head only).

5. CONCLUSION

Color graphs appear to have distinct advantages for abstracting essential properties of classes and capturing informations that were left implicit in traditional OOP. The formalism structures the method space and externalizes the control flow. It is expected to be attractive for three reasons : from a language point of view, as a structuring mechanism for reusable software ; from a programming environment point of view, as a basis for a friendly and powerful visual interface with simulation capabilities ; from a cognitive point of view, as a better support of the [opportunistic] programming process.

Our interest in cognitive psychology cannot replace theoretical and experimental expertise of scientists of the domain. We are clearly interested in estimating our proposal from a cognitive point of view using, for example, "cognitive dimensions". The tools (programming environment) as well as the notation (language) should benefit from such an analysis. Conversely, given the challenge it represents for mastering complexity, COP may well become —after OOP— a field of reflexion and experiments for cognitive psychologists (concerning OOP, such a goal is apparent in [Green-Gilmore-Blumenthal-Davies-Winder, 1992]) : on one hand, COP splits up the code even more than OOP; on the other hand, it offers a new level of structuring .

Acknowledgements

Thanks to Francoise Détienne for commenting on the remarks we made in [Borron, 1995a] [Borron, 1995b] about cognitive aspects.

Bibliography

[Adelson-Soloway, 1988] B. Adelson & E. Soloway. *"A model of software design"*. In M.T.H. Chi, R. Glaser & M.J. Farr (Eds.). The nature of expertise. pp. 185-208. Lawrence Erlbaum Associates. 1988.

[Aksit-Bergmans, 1992] Mehmet Aksit & Lodewijk Bergmans. *"Obstacles in Object-Oriented Development"*. In OOPSLA'92 Proceedings. October 18-22, 1992.

[Bellamy-Carroll, 1990] Rachel K.E. Bellamy & John M. Carroll. *"Case-based reuse"*. Research Report RC 16119 - IBM Research Division, T.J. Watson Research Center, Yortown Heights, NY 10598. (Also in Proceedings of the ACM SIGPLAN Symposium : Object-Oriented Programming Emphasizing Practical Applications.) Sept. 1990.

[Böcker-Herczeg, 1990] Heinz-Dieter Böcker & Jürgen Herczeg. *"Browsing through program execution"*. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Human-Computer Interaction - Interact'90. Elsevier. pp. 991-996. 1990.

[Bobrow et alii, 1988] Daniel G. Bobrow , Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales & David A. Moon. *"CLOS"*. X3J13 Document 88-002R. June 1988.

[Borron, 1995a] Henry J. Borron. *"Colored-Object Programming : (1) Describing Interfaces"*. In GL'95 Proceedings. (Huitièmes Journées Internationales sur le Génie Logiciel et ses Applications.) November 15-17, 1995.

[Borron, 1995b] Henry J. Borron. *"Colored-Object Programming : (2) Concrete and Abstract Implementations"*. In GL'95 Proceedings. November 15-17, 1995.

[Borron, 1996a] Henry J. Borron. *"Colored-Object Programming : About the visual formalism"*. (First version in January 1996.) Research Report 2879, april 1996.

[Borron, 1996b] Henry J. Borron. *"Colored-Object Programming : Color graphs, a visual formalism for synthesizing the behaviour of objects"*. (First version in February 1996.) Research Report 2876, april 1996.

[Borron, 1996c] Henry J. Borron. *"Colored-Object Programming : mixin and derivation, two conjoint concepts for a rigorous handling of independent supplementary behaviours"*. (First version in February 1996.) Research Report 2877, april 1996.

[Borron, 1996d] Henry J. Borron. *"Colored-Object Programming : Inheritance by dimensions"*. (First version in October 1995.) Research Report 2878, april 1996.

[Borron, 1996e] Henry J. Borron.*"Colored-Object Programming : Ergonomic and cognitive issues"*. May 1996. To be published in ERGO-IA'96 Proceedings (october 1996).

[Borron, 1996x] Henry J. Borron.*"A two-pass effcient and monotonic linearization algorithm"*. In preparation. INRIA Sophia-Antipolis. 1996.

[Burkhardt-Détienne, 1995a] Jean-Marie Burkhardt & Françoise Détienne. *"An empirical study of software reuse by experts in object-oriented design"*. Interact'95. June 1995.

[Burkhardt-Détienne, 1995b] Jean-Marie Burkhardt & FrançoiseDétienne. *"La réutilisation de solutions en conception de programmes informatiques"*. (In French.) Revue de Psychologie Française. Numéro spécial pour l'ergonomie cognitive. 1995.

[Carroll-Mack, Carroll, 1985] J.M. and Mack, R.L. *"Metaphor, computing systems and learning"*. International Journal of Man-Machine Studies, 22, 39-57. 1985.

[Carrol-Mack-Kellog, 1988] Carroll, J.M., Mack, R.L., dand Kellog, W.A.. *"Interface metaphors and user interface design"*. In M. Helander (ed.), Handbook of Human-Computer Interaction, Elsevier Science Publishers, North Holland, pp. 67-85. 1988.

[Carroll-Kellogg, 1989] John M.Carroll & Wendy A. Kellogg. *"Artifact as theory-nexus : hermeneutics meets theory-based design"*. In K. Bice and C. Lewis (Eds.), Proceedings of CHI'89. Austin. Special Issue of the SIGCHI (Computer-Human Interaction) Bulletin. pp. 69-73. April-May 1989.

[Chatel-Détienne, 1994] Chatel & Détienne. *"Expertise in object-oriented programming""*. ECCE7. Bern, 5-8 sept 1994.

[Cointe, 1988] Pierre Cointe, personal comunication.

[Curtis, 1990] Bill Curtis. *"Empirical studies of the software design process"*. In D. Diaper, D. Gilmore, G. Cockton & B. Shackel (Editors). Human-Computer Interaction. Interact'90. Elsevier. pp. xxxv-xl. August 1990.

[Dahl-Dijkstra-Hoare, 1972] Ole-Johan Dahl, Edsger.W. Dijkstra & C.A.R. Hoare. *"Structured Programming"*. A.P.I.C. Studies in Data Processing no. 8, Academic Press. 1972.

[Davies, 1991.a] Simon P. Davies. *"Characterizing the program design activity : neither strictly top-down nor globally opportunistic"*. Behaviour & Information Technology. Vol. 10. No 3. pp. 173-190. 1991.

[Davies, 1991.b] Simon P. Davies. *"The role of notation and knowledge representation in the determination of programming strategy : a framework for integrating models of programming behavior"*. Cognitive Science. Vol 15, 547-572. 1991.

[Descartes, 1637] René Descartes. *"Discours de la méthode pour bien conduire sa raison et chercher la vérité dans les sciences (...)"*. Jean Maire (Ed.). 1637.

[Détienne, 1990a] Françoise Détienne. *"A cognitive ergonomics approach for the evaluation of an object-oriented programming system : the example of the O2 system"*. (In French). Ergo-IA'90. Sept. 1990.

[Détienne, 1990b] Françoise Détienne. *"Difficulties in designing with an object-oriented language : an empirical study."* In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Human-Computer Interaction - Interact'90. Elsevier. pp. 971-976. 1990.

[Dijkstra, 1976] Edsger.W. Dijkstra. *"A discipline of programming"*. Prentice-Hall Series in Automatic Computation. Prentice-Hall . 1976.

[Ducournau et alii, 1992] R. Ducournau, M. Habib, M. Huchard & M.L. Mugnier. *"Monotonic conflict resolution for inheritance"*. In OOPSLA'92 Proceedings. October 18-22 1992. pp. 16-24.

[Ducournau et alii, 1994] R. Ducournau, M. Habib, M. Huchard & M.L. Mugnier. *"Proposal for a monotonic multiple linearization"*. In OOPSLA'94 Proceedings. October 23-27 1994. pp. 164-175.

[Fischer, 1987] Gerhard Fischer. *"Cognitive view of reuse and redesign"* IEEE Software. pp. 60-72. July 1987.

[Fitter-Green, 1979] M. Fitter & T.R.G. Green. *"When do diagrams make good computer languages ?"*. International Jounal of Man-Machine Studies. Vol.. 11. pp. 235-261. 1979.

[Goldberg-Robson, 1983] Adele Goldberd & Daniel Robson. *"Smalltalk-80 : The Language and its Implementation"*. Addison-Wesley. 1983.

[Green, 1982] T.R.G. Green. *"Pictures of programs and other processes, or how to do things with lines"*. Behaviour and Information psychology. Vol. 1, no 1, pp. 3-36. 1982

[Green, 1989] T.R.G. Green. *"Cognitive dimensions of notations"*. In A. Sutcliffe and L. Macaulay (Eds.), People and Computers (Vol 5). pp.443-460. Cambridge University Press. 1989.

[Green, 1990] T.R.G. Green. *"The cognitive dimension of viscosity: a sticky problem for H.C.I"* In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Human-Computer Interaction - Interact'90. Elsevier. pp. 79-85. 1990.

[Green-Gilmore-Blumenthal-Davies-Winder, 1992] T.R.G. Green, D.J. Gilmore, B.B. Blumenthal, S. Davies & R. Winder. *"Towards a Cognitive Browser for OOPS"*. In International Journal of Human-Computer Interaction. Volume 4, no1. pp.1-34. 1992.

[Green-Sime-Fitter, 1981] T.R.G. Green, M.E. Sime and M.J. Fitter. *"The art of notation"*. In Computing Skills and the Computer Interface. pp. 221-251. M.J. Coombs & J.L. Alty (Eds). Academic Press. 1981.

[Guindon, 1990.a] Raymonde Guindon. *"Knowldge exploited by experts during software system design"*. International journal of Man-Machine Studies, Vol. 33, pp. 279-304. 1990.

[Guindon, 1990.b] Raymonde Guindon. *"Designing the design process : exploiting opportunistic thoughts"*. Human-Computer Interaction, Vol. 5, pp. 305-344. 1990.

[Hadamard, 1945] Jacques Hadamard. *"The psychology of invention in the mathematical field"*. Dover Publications, New York. 1945.

[Harel, 1987] D. Harel. *"Statecharts : a visual formalism for complex systems"*. Science of Computer Programming. Vol. 8, no 3, pp. 231-274. June 1987.

[Harel, 1988] D. Harel. *"On visual formalisms"*. Communications of the ACM, Vol. 31, no 5, pp. 514-530. May 1988.

[Harel et alii, 1987] D. Harel, A. Pnueli, J.P. Schmidt, R. Scherman. *"On the formal semantics of statecharts"*. (Extended abstract). In proceedings of the second IEEE symposium on logic in computer science (Ithaca). pp. 54-64. IEEE Press. June 1987.

[Hoc, 1988] Jean-Marie Hoc. *"Towards effective computer aids to planning in computer programming : theoretical concern and empirical evidence drawn from assessment of a prototype"*. In G. C. van der Veer, T.R.G. Green, J.M. Hoc & D. Murray (eds.). Working with computers : Theory versus Outcomes. Academic Press. 1988.

[Jackson, 1975] M. Jackson. *"Principles of program design"*. Academic Press, London. 1975.

[Kant-Newell, 1984] E. Kant & A. Newell. *"Problem solving techniques for the design of algorithms"*. Information Processing and Management. No 28. pp. 97-118. 1984.

[Keene, 1989] Sonya E. Keene. *"Object-oriented progamming in Common Lisp. A programmer's guide to CLOS"* Addison-Wesley in association with Symbolics Press. 1989.

[Kleyn-Gingrich, 1988] Michael. F. Kleyn & Paul C. Gingrich. *"GraphTrace - Understanding object-oriented systems using concurrently animated views"*. In proceedings of OOPSLA'88 Conference (San Diego). SIGPLAN Notices. Volume 23, no 11. pp.191-205. September 1988.

[Lange-Moher, 1989] Berth M. Lange & Thomas G. Moher, T.G. *"Some strategies of reuse in an object-oriented programming environment"*. In K. Bice and C. Lewis (Eds.), Proceedings of CHI'89. Austin. Special Issue of the SIGCHI (Computer-Human Interaction) Bulletin. pp. 69-73. April-May 1989.

[Larkin-Simon, 1987] Jill H. Larkin & Herbert A. Simon. *"Why a diagram is (sometimes) worth ten thousand words"*. Cognitive Science. Volume 1, no 1, pp. 65-100. January-March 1987.

[Lewis-Henry-Kafura-Schulman, 1991] John A. Lewis, Sallie M. Henry, Dennis G. Kafura & Robert S. Schulman. *"An empirical study of the object-oriented paradigm and software reuse"*. In proceedings of OOPSLA'91 Conference on Object-Oriented Programming Systems and Languages. pp. 184-196. 1991.

[Lieberman, 1986] Henry Lieberman. *"Using prototypical objects to implement shared behaviour in object-oriented systems"*. In proceedings of OOPSLA' 86 (Portland). SIGPLAN Notices, vol 21, no 9, pp. 214-223. 1986.

[Linger-Mills-Witt, 1979] Richard C. Linger, Harlan D. Mills & Bernard I. Witt. *"Structured Programming : Theory and Practice"*. Addison-Wesley, Reading, Mass. 1979.

[Meyer-Baudoin, 1978] Bertrand Meyer, Claude Baudoin. *"Méthodes de Programmation"*. Collection de la Direction des Etudes et Recherches d'Electricité de France. Eyrolles. 1978.

[Mills et alii, 1987] Harlan D. Mills et alii. *"Principles of Computer Programming : A mathematical approach"*. Allyn and Bacon, Rockleigh, N.J., 1987.

[Oquendo et alii, 1993] Flavio Oquendo, Françoise Détienne, Nando Gallo, Anne Kastner & Alessandra Martelli. *"The SCALE Project : building PCTE-based process-centred environments for supporting large and fine grain reuse"*. In proceedings of the 2nd international conference on PCTE (PCTE'93). Paris. November 17-18 th 1993.

[Nassi-Schneiderman, 1973] I. Nassi & B. Schneiderman. *"Flowchart techniques for structured programming"*. SIGPLAN Notices, 8 (8), pp. 12-26. 1973.

[Papert, 1980] Seymour Papert. *"Mindstorms. Children, computers and powerful ideas"*. Basic books, 1980

[Papert, 1981] Seymour Papert. *"Jaillissement de l'esprit. Ordinateurs et apprentissage."*. Flammarion, 1981

[Pennington, 1987] Nancy Pennington. *"Stimulus structures and mental representations in expert comprehension of computer programs"*. Cognitive Psychology. Vol. 19. No 3. pp 295-341. July 1987

[Petre-Winder, 1988] Marian Petre & Russel Winder. *"Issues governing the suitability of programming languages for progamming tasks"*. In D.M. Jones and R. Winder (Eds.), People and Computers IV (Vol 6). pp. 199-214. Cambridge Unversity Press. 1988.

[Rist, 1986] Robert S. Rist *"Plans in programming : definition, demonstration and development"*. In Empirical Studies of Programmers. E. Soloway & S. Iyengar (eds.). Ablex. 1986.

[Rist, 1989] Robert S. Rist *"Schema creation in programming"*. Cognitive Science, 13, pp. 389-414. 1989.

[Rist, 1990] Robert S. Rist *"Variability in program design : the interaction of process with knowledge"*. In International Journal of Man-Machine Studies. No 33. pp-305-322. 1990.

[Roberts-Wei-Winder, 1990] G.A. Roberts, M. Wei & R.L. Winder. *"The Solve object-oriented programming system for parallel computers"*. Technical Report SPAN-WP-10-Deliverable-28. University College London. CS Department. 1990.

[Scanian, 1989] David A. Scanian. *"Structure Flowcharts outperform pseudocode : an experimental comparison"*. IEEE Software. September 1989.

[Smith, 1977] David Canfield Smith. *"Pygmalion. A computer program to model and stimulate creative thought"*. Interdisciplinary Systems Research no 40. Birkhaüser Verlag (Basel & Stuttgart). 1977.

[Soloway-Erlich, 1984] Soloway & Erlich. *"Empirical studies of programming knowledge"*. IEEE Transactions on Software Engineering. SE-10, 5. pp. 595-609. 1984.

[Springer-Friedman, 1989] G. Springer & D.P. Friedman. *"Scheme and the Art of Programming"*. MIT Press. 1989.

[Steele, 1990] Guy L. Steele. *"Common Lisp : the language"* . 2nd edition, Digital Press, 1990.

[Strom-Yemini, 1986] Robert E. Strom & Shaula Yemini. *"Typestate : A Programming Language Concept for Enhancing Software Reliability"*. In IEEE Transactions on Software Engineering, Vol. SE-12, no1. pp. 157-171. January 1986.

[Sutcliffe-Maiden, 1990] Alistair Sutcliffe & Neil Maiden. *"Software reusability : delivering productivity gains or short cuts."* In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.), Human-Computer Interaction. Interact'90. Elsevier. pp. 895-901. 1990.

[Woodfield-Embley-Scott, 1987] Scott N. Woodfield, David W. Embley & Del T. Scott *"Can Programmers Reuse Software ?"*. IEEE Software. pp. 52-59. July 1987.

APPENDIX A : GETTING RID OF EXPLICIT TEST CONTROL STRUCTURES

A.1 Historical perspective

A.1.1 Object oriented programming

COP is a refinement of OOP. But what is OOP ? If one single idea was to be retained from the latter, we would rather say that it is the responsability given to each data structure to know what piece of code is to be run when asked to do something. Before that, the code was decoupled from the data structures. For example, in PASCAL, any piece of code (a "procedure" or a "function") has a priori access to all data-structures : any data structure, may a priori be traversed for consultation or modification. In OOP, each data-structure (termed "object") is attached a dictionary of the code to be run ("method") when asked to do something (when "sent a message"[9]).

Having this view in mind, it is quite immediate to think about avoiding to repeat the same methods and memory representations whenever possible. Hence, the idea of sharing. This idea first leads to specify in one place (a "class"[10]) the description of objects having the same memory representation and the same methods. These objects ("instances" of their class) differ by the values they hold in their memory representation[11].
Once object descriptions are organized into classes, it does not take a long time to realize that the idea of sharing can be pushed further : it is recurrent that one description is almost like one or several other ones. Hence, the idea of deriving this description from others and associating it an increment to slightly complement or modify the behaviour obtained from the "inherited" classes. Systematically used, this concept of "inheritance" organizes classes into a hierarchy, a tree (simple inheritance) or a lattice (multiple inheritance). Smalltalk [Goldberg-Robson, 1983] is the prototype of this sort of languages. It features simple inheritance[12]. When sent a message, a Smalltalk object first consults the dictionary of its class ; if a method convenes, this method is executed ; otherwise, the object consults its superclass dictionary, etc ; if no method is found, an error message is issued[13]. (The memory representation used for an instance of a class is obtained by concatenating the local memory representation with all the inherited ones.)

Compared to traditional imperative programming, one may thus view object orientation as the consequence of two simple re-organizations (not fundamentally different from those made by attentive secretaries concerning the documentation of their bosses) : first one consists in sytematically grouping representation and code pieces that work together into one folder ; second one, in organizing the resulting folders into a hierarchical classification[14].

A.1.2 Colored object programming

COP is designed to push further the basic principle of OOP exposed above, i.e. to give data-structures the responsability of operations they can support. Several converging reasons motivate this decision. Let's give one. ,

When compared to structured programming [Dahl-Dijkstra-Hoare, 1972] [Nassi-Schneiderman, 1973] [Dijkstra, 1976] [Linger-Mills-Witt, 1979] [Mills et alii, 1987], one interesting consequence of OOP is that it partly eliminates the presence of test control structures from the source code.

Consider, for example, a drawing you are making on your favorite portable computer. Suppose you move a group of objects on the screen : these objects need to be redisplayed in their final position. Internally, the group is implemented as a list, each element of the list being either a line, a rectangle, a curve, a string, ... The implementer is faced with the following problem : each element of the list is to be redisplayed yet its type is not known in advance. To accomplish

---

9 "message sending" pertains to the Smalltalk wording. In Smalltalk, a message has but one "receiver". This modelling is perfectly acceptable. Yet, it gets uneasy when several objects do participate in the selection of the piece of code to be run (ex. asking a rectangle to draw itself using a given pattern could alternatively be done by asking the pattern to fill in the rectangle : as a matter of fact, both the pattern and the rectangle intervene). A more general approach, examplified by CLOS, considers a "generic function call" instead of a "message sending" : such a call may involve several "specializers" instead of a single message receiver.

10 An alternative modelling exists which potentially considers any object as a "prototype" for making new objects. See [Lieberman, 1986] for more details.

11 i.e. in their "instance variables" (Smalltalk wording) or "slots" (CLOS wording)

12 CLOS [Bobrow et alii, 1988] is an example of language supporting multiple inheritance.

13 This "class based" modelling is the dominant one in OOP. For the moment, COP is "class based" ; it has not yet been adapted to the "prototype based" scheme.

14 An important goal of OOP is reuse. To facilitate it, object are considered as black-boxes (no direct access is given to the memory representation from the outside : a message is the necessary medium) ; in addition, objects should be simple : because complex records (like those that may be built in PASCAL or C...) are difficult to reuse, only simple ones are considered.

this in a traditional (imperative) way, the programmer uses a test control structure (a CASE statement, for example) so as to jump to appropriate code for lines, rectangles, curves, strings, ... after testing the current element's type. Each possible case should be mentioned : the programmer is thus required to somehow compile programs in his/her head before coding them for the computer (hence, a cognitive load with all its possible consequences, notably errors).

In an OOP program, this CASE simply disappears : each element of the loop is sent a *display* in turn ; when receiving it, this element (i.e. a window, a rectangle, a curve, a string, ...) dynamically looks for an appropriate method in its class (and superclasses) as explained above. No test control structure is any longer used for making decisions depending on types (on "classes" to be more exact). This is an important point in practice because the programmer's task is simplified while the code gets more robust to changes : in traditional programming, adding a new type (ex. : *Oval*) means the CASE control structure is to be searched and its contents is to be updated ; in OOP, the programmer is free from this burden.

As implicitely mentioned above, not all control structures were eliminated from the OOP source code. To remove extra ones, we need to reuse the same trick, i.e. to exhibit a sub-typology among objects of a same class. Distinguishing instance states is then a natural idea.

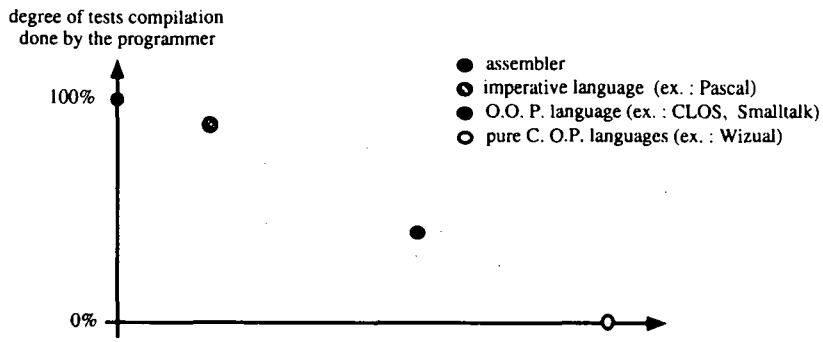


Figure A.1.

A quite interesting consequence appears immediately : because the sub-typology enables all explicit test control structures (like IF, CASE, WHEN,..) to be removed from method bodies, all explicit loop control structures (like WHILE, REPEAT, DO...) may also be removed (once loops are systematically implemented in a recursive form) without leaving any explicit test control structure. Thus, method bodies get purely sequential[15].

A.2 Aesthetic and practical motivations

This approach is attracting for an aesthetic reason as well as for practical ones.

A quest for systematic symmetry, monotonicity, .simplicity, etc., the aesthetics in programming is a notion difficult to talk about, yet extremely important in practice for a programmer (at least, it should be) and for a language designer : *"Simplicity is nowhere more practical than in programming, where the bane is complexity. When just the right abstraction for a problem has been found, it may be a thing of beauty. "* [Springer-Friedman, 1989], pp. 3-4. Experience shows the aesthetic motivation is an excellent guide in computer science (as it is in other disciplines : for example, the quest —and discovery— of an atomic subparticle because of an otherwise lack of symmetry in a group structure). The sole aesthetic reason may, in our view, justify the development of the idea.

But practical reasons cannot be neglected. Consider, for example, the editing of method bodies. If imperative statements are permitted, a visual syntax-directed editor is awkward in its use since it participates to two styles of programming : the imperative one (statements) and the object oriented one (message sendings). Note that a textual syntax-directed editor will mask the differences and thus the awkwardness.

A.3 Cognititive motivations

From a cognitive point of view, the sequentiality of method bodies changes the representation one may have of programs. Students are told that programming basically requires three types of control structures : the sequence, the test and the loop. Except in rare cases[16], this is done using an imperative language as an example. (Is an imperative language simpler for learning programming concepts ??) This first exposure to programming is somewhat critical[17] :

---

15 This is only a possibility. One can easily imagine an impure COP language that still allows explicit loop and test control structures. (The past has demonstrated the successful emergence in the software industry of impure languages, even for OOP.)

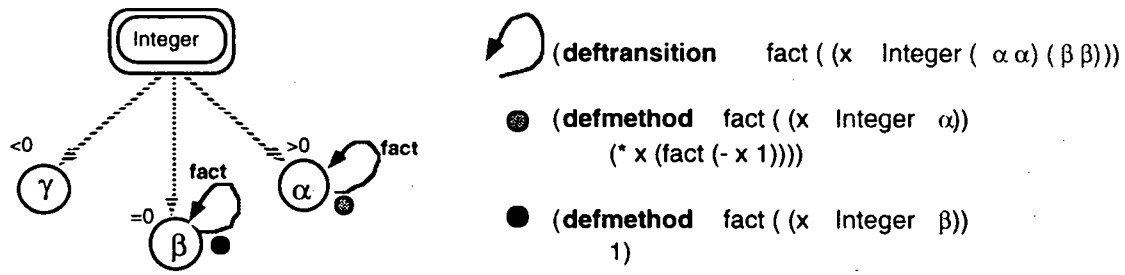
16 As far as we know, in some CS departments, students are exposed to Lisp without prior imperative programming experience.

17 yet, fortunately, not as critical as the first exposure of Konrad Lorenz's grey geese to a parent substitute!

as noted in ([Détienne, 1990a], §4.4.2) and ([Détienne, 1990b], §4.3.3), programmers not experienced in OOP tend to transfer their (imperative) experience to OOP ; even, programmers experienced in OOP remain somewhat influenced by their first (imperative) experience [Détienne, 1990b] ; we also know a graduate student who proudly announced that he never writes a recursive function but always implements it using an explicit stack ! In COP, the sequences, tests and loops still exist, yet in a different manner for the last two. As explained above, more immediate concepts are typologies and recursivity. In fact, typologies appear as the focus of the programmer's activity : once identified, programming consists simply in sending messages (recursivity is so natural that it is transparent).

Because, pure COP prones systematic recursivity, let's comment this point a bit. In our view, it is far better to teach recursivity as a basic mechanism (prior to any exposure to explicit loop imperative control structures) as it is done in some computer science departments[18] : a recursive program is simple to state ; what is not simple is to emulate it in our own mind : in fact, that's the job of the computer! Explicit loop control structures are easy to use in simple cases (and recursivity too) ; yet, when cases are getting more complex, they get unmanageable (recursivity is the only practical way). In our view, programmers who are first exposed to explicit loop control structures may well tend to systematically rely on this mechanism : such an attitude is usually not a problem since all common languages provide these explicit loop control structures ; when encountering complex cases (too complex to be treated in practice with explicit loop control structures), these programmers reluctantly turn to recursivity ; being not used to it, they encounter difficulties (not carefully distinguishing between several incarnations, for example) and they likely debug they program by emulating its execution in their own mind. As a consequence, these programmers associate recursivity and difficulty. As it may happen that they asked to teach programming to others, they may well transfer their bad understanding and negative opinion. Having informally discussed this point with a few cognitive psychologists, it appears that their opinions diverge on the cognitive load due to the use of recursivity vs. imperative control structures. A comparative study avoiding possible pitfalls would be quite interesting.

In COP, instead of the sequence, test and loop explicit control structures, the programmer faces a typology of clesses, a sub-typology of states and pure sequences of code. In such an environment, the rule of thumb is thus to precisely distinguish each case and attach it a simple (sequential) piece of code. Reducing the complexity at each step is not less natural than moving down along a ladder : one first identifies each rung ; then, step after step, he/she moves one leg and the other. For example, to implement the famous *factorial* method, the programmer may distinguish the following cases under the *Integer* class : (1) zero ; (2) positive ; (3) negative. In the first case, the body of the method is *"1"* (it evaluates to one ; the result of the evaluation is returned) ; in the second case, it is "*self* * (*self* -1) *factorial*" (meaning "consider the integer just lower to the receiver's value, send it the *factorial* message, multiply the answer by the receiver's value and return the result") ; in the third case, the user may provide a body for explicitely signalling an error : *"self error: 'not valid' "* [19] ; the user may also specify the transition *fact* exists for all integers except the negative ones (see next figure). Note this style of programming is possible in OOP, certainly more elegantly when individual methods are supported as in CLOS with the *(eql form)* type of specializer. Further note that the fusion of OOP individual and regular methods is representative of the aesthetic improvements brought by COP.



Figures A.2 & A.3.

In COP, the sequentiality of method bodies also eases the programmer's work. Basically, the effort for expliciting states is beneficial to the programmer because the underlying system is able to take advantage of an otherwise unrevealed structure : for example, the condition attached to a state needs to be specified only once by the COP programmer whereas it possibly has to be managed several times in more traditional programming (hence, possible errors) ; the revealed structure also grasps constraints that are automatically taken into account by the underlying system, thus relieving the programmer from this burden. This new structure is itself of direct interest to the programmer as an abstract layer : it explicits states and relations between them (note that OOP methodologies often suppose a concept of states whereas OO languages most usually do not support it [Aksit-Bergmans, 1992]) ; it also expresses how messages can be ordered ; on the other hand, the context for defining a method is narrowed (methods are sequential and are tinier than usual in OOP) : the attention of the programmer is thus moore easily focused and errors are more unlikely than usual.

---

[18] We were told a few years ago this does not seem to pose a problem to students [Cointe, 1988].

[19] Here, we use a Smalltalk-like syntax : *self* represents the message receiver.

APPENDIX B : A CLEAR DISTINCTION BETWEEN THE SPECIFICATION AND IMPLEMENTATION LEVELS

B.1 The specification level

Given a class of objects, the specification level describes only states and transitions between these states. Each transition depicts the possible occurrence of a message in a certain state of the receiver as well as the effect of this message, i.e. the new state of the receiver (this description corresponds to single dispatch[20] but it can easily be generalized to multiple-dispach[21]).

States and transitions are described in a multi-dimensional space (N≥1). In the simplest case (one dimension), each state is represented by a single node (termed a **color**) and each transition by a single **arc** between two colors. Each color is given a name and is attached a **condition**. A color may be **basic** or **ephemere**, in which case it represents the ORing of two or more other colors (destination colors). An ephemere color and its destination colors instantiate a **construct** termed a **selection**. The current state of a given instance is represented by a **token** moving in the color graph according to the triggered transitions. Such a color graph in a single dimension space is termed a **c-graph**.

As an example, let's consider a *Stack* instance. When just created, it is *empty* : its current state is represented by a token in a node (say, color 1) that is attached the condition *empty*. In this state, the considered instance can only be sent a *push:* message. Once this first *push:* has occurred, the instance is *not empty* : the *push:* transition has moved the instance token to a second node (say, color 2) that is attached the condition *not empty*. Subsequent *push:* or *top* messages in color 2 do not change the instance color. A *pop* transition may also occur in color 2. After it, a *Stack* instance may be either *empty* or *not empty*. To model this uncertainty, the *pop* transition is specified as flowing from color 2 to an ephemere color (say, color 3) the condition of which ORs the conditions of color 1 (*empty*) and color 2 (*not empty*). A *pop* transition thus moves the instance token from color 2 to color 3. When in color 3, the *empty* **testing transition** is automatically triggered by the underlying system to decide which node, color 1 or color 2, actually reflects the current instance state. When the decision has been made, the token is moved to the actual destination : this is formalized as the triggering of a **reflex transition** flowing from the ephemere node. One reflex transition exists per selection destination. A reflex transition fires when the condition of its destination gets true. Reflex transitions are also used in a static way for **local inheritance**, i.e. inheritance inside one color graph, a quite important role indeed. An example is given here by the *empty* transition which also exists in color 1 and in color 2 : in color 1 (resp. color 2), the *empty* transition yields *true* (*false*) and the instance state remains unchanged. The *empty* transition in color 3 is **inherited** in colors 1 and 2 along the reflex transitions flowing from color 3.

The next figure shows the *Stack* color graph. Nodes are represented by circles with a number inside (the name of the node) ; (regular) transitions, by solid arrows ; reflex transitions, by dashed arrows. The condition attached to a node is represented in italics. The *empty* transitions in color 1 and 2 are not shown since they are inherited from color 3. Usually, the *empty* transition in the ephemere color 3 is itself not shown since it can be easily inferred from the existence of conditions *empty* and *not empty* respectively in colors 1 and 2. For the same basic reason, the *t* (*true*) condition in color 3 is usually not shown too (it ORs the *empty* and *not empty* conditions). The small bubble inside color 1 represents a token : in this case, it depicts an *empty* instance.

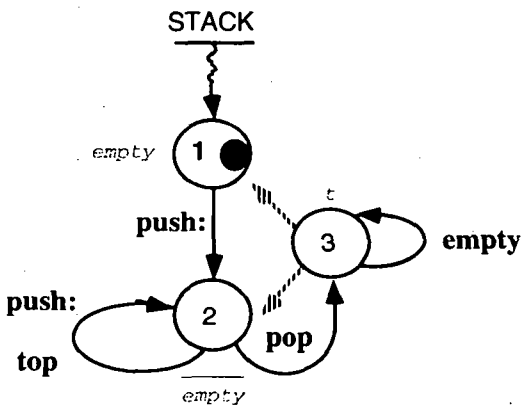


Figure B.1.

[20] The single dispatch case corresponds to the usual "message" metaphor illustrated by Smalltalk. A single object, termed the receiver, is ask to perform a given action m.

[21] The multiple dispatch case corresponds to "generic function calls" in CLOS. Several arguments are required instead of a single one as in the case of a message.

In more complex cases, a N-dimensions space (N≥2) is used to describe states and transitions. For each dimension, one or several nodes (termed **pigments**) may be defined. Besides the selection, two new constructs exist : the **decomposition** which transforms a color into a set of N pigments ; the **conjunction** which groups two or more pigments into one node termed a **blend** (this blend happens to be a color when recursively grouping N pigments). Blends are useful in certain circumstances (for simplifying the expression of transitions), but they always can be not used at all. All constructs (selection, decomposition, conjunction) are conceptually composed using reflex transitions. For more details, refer to [Borron, 1996a] [Borron, 1996b].

B.2 The implementation level

An implementation associates a set of memory representations and methods to a color graph (hence an augmented color graph). A memory representation is necessary for differentiating states ; methods are necessary to perform transitions between states. A same color graph may be implemented in different ways, i.e. according to different sets of memory representations/methods. As for regular transitions, memory representations and methods are inherited along reflex transitions inside a same color graph (local inheritance).

In practise, a memory representation can be associated to each color or pigment ; a pre- and a post-method, to each transition (the **pre-method** at the transition source ; the **post-method** at the transition destination). Pre- and post-methods are collectively called **micro-methods** (or, more loosely, methods).

In our drawings, pre- and post-methods are represented using two small circles : a white one at the source ; a black one at the destination (see next figure). A memory representation is depicted by a pattern inside the node in question. (A white pattern corresponds to an absence of specification.)

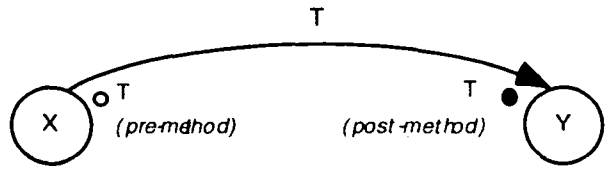
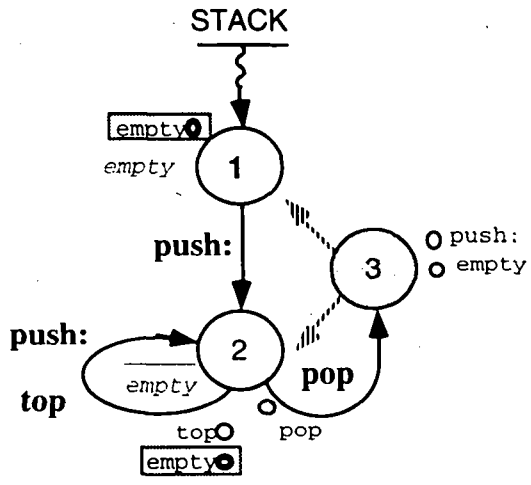
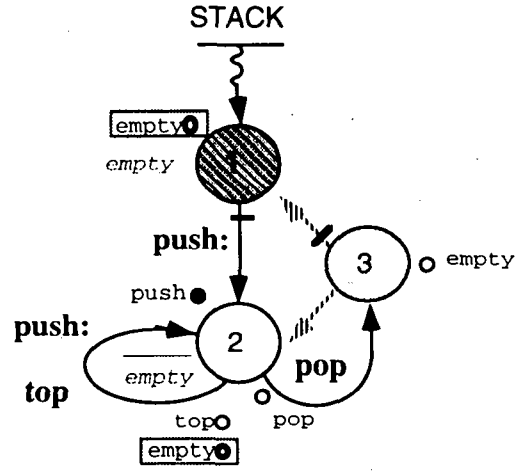


Figure B.2.

Next two figures represents two different implementations of the *Stack* specification.

The first one corresponds to a standard implementation : a same memory representation (one slot for holding a list of elements) is used in all possible states. This memory representation being common , it is specified only in color 3 and is thus inherited in colors 1 and 2 via the reflex transitions outcoming from color 3. All methods (*empty*, *push:*, *pop*, *top*) could also be defined in color 3 only, for being inherited in other colors if necessary. As a result of a certain discipline, we attached the *pop* and *top* methods (pre-methods in fact) to color 2, and only attached the *push:* and *empty* methods to color 3 : the *empty* method is necessary for implementing the *empty* testing transition of color 3 ; the *push:* method is factorized in color 3 even if not used in this color (it is used in colors 1 and 2). The *empty* method of color 3 is a priori inherited in colors 1 and 2 ; in fact, it is **masked** by the *empty* **constant methods** automatically generated by the underlying system in these colors (these constant methods are represented inside grey bordered rectangles).





Figures B.3 & B.4.

The second figure corresponds to a non-standard implementation based on the idea that no slots are really needed in the *empty* color since no elements need to be held. A specific memory representation is thus specified for color 1. In

addition, the pre-method is void (the pushed element cannot be stored as long as no slot exists to refer to it) : the post-method does the job of storing the instance ; because it may also implement the circular *push:* transition in color 2, this post-method is in fact specified as implementing both *push:* transitions. The two small bars represents an automatic change of representations.

APPENDIX C :
HIGHER MODULARITY VIA MULTIPLE DIMENSIONS
1) SPECIFICATION LEVEL

In the preceding appendix, we describe the *Stack* c-graph using a single dimension space. Yet, the proposed formalism (as well as the machinery behind it) supports more dimensions, an important aspect in practise. It corresponds to one of the problem solving and design heuristics proposed by Raymonde Guindon in [Guindon, 1990.a] : *"Identify system functions that can be performed nearly independently and divide the system into corresponding subsystems"*.

C.1 First Example

A p-graph is a color graph that exhibits two or more dimensions (N≥2). Whereas a c-graph (N=1) describes states as points along one same axis, a p-graph decomposes a state onto several axes (one axis per dimension).

Next figure shows one example. It is is derived from the source code of the *Person* example given in section 4.2 of [Chambers, 1993][22]. *Person* instances are differentiated into *male* and *female* categories according to the *sex* ; *child*, *teenager* and *adult* categories, according to the *age*. In addition, both criteria are used to distinguish *boy* and *girl* categories.

In terms of COP, two dimensions are used (N=2) : **pigments** 2 to 5 are described along the first one ; pigments 6 to 12, along the second one. Color 1 is the initial state : it is **decomposed** according to the two dimensions (pigments 2 and 6). **Blends** 13 and 14 result from two **conjunctions**. They may be termed colors since they group a pigment of each dimension. Pigments 2, 4 and 5 (resp. 6, 10, 11, 12) are **basic** pigments along the first dimension (resp. along the second one) ; pigments 3, 7, 8, 9 are **ephemere** pigments. Given an instance, its current state is marked by two **mini-tokens**, one per dimension : in the figure below, the instance which is represented in in state (2, 10).

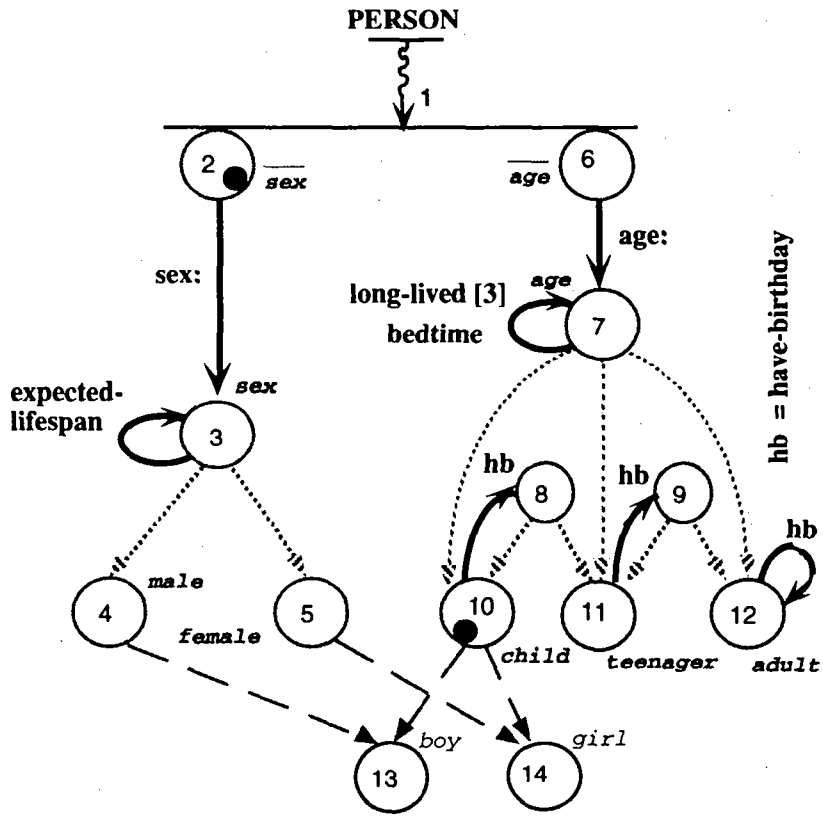


Figure C.1.

22 For the purpose of the demonstration, a slight difference exists to create an instance of *Person* : the cited paper proposes a *make-person* method with values for *sex:* and *age:*. The corresponding creation transition is not represented in the figure.

Most transition names are self explanatory. Messages *sex:* and *age:* are used for **initialization**. The circular *expected-lifespan* transition is valid for a *male* or *female* instance (the returned value depends on the *sex*), yet it is attached to the ephemere node 3 for factorization purpose. For similar reasons, *bedtime* is also represented by a circular transition attached to the *age* pigment. The *have-birthday* message is kind of special since, by incrementing the *age*, it makes an instance stay in its pigment (ex : *child*) or move to the next one (ex: *teenager*) except when in the *adult* pigment. No symmetry exists that can be taken advantage of. To model this situation, two **selections** are used (for the *have-birthday* transitions outcoming from the *child* and *teenager* pigments) plus one circular transition (for the *have-birthday* transition outcoming from the *adult* pigment). This makes pigments *child*, *teenager* and *adult* to inherit from two or three nodes. *Long-lived* — which tells if a given instance has gone farther its *expected-lifespan* — is represented by a **constrained** circular transition attached to the *age* pigment. This is because both the *sex* and the *age* should be known : one condition is captured in having the transition attached to the *age* pigment ; one, in having a **clause** mentioning the *sex* pigment (pigment number 3). (The opposite choice is perfectly acceptable.) *Long-lived* is valid for any fully initialized state, among which *boy* and *girl* : the notation we used avoids the enumeration of all these states (6 stable ones). Note that the *boy* and *girl* conditions attached to blends 13 and 14 induce the existence of a *boy* and a *girl* testing transitions valid for all fully initialized *Person* instances (not only in states *boy* or *girl*).

The proposed formalism really expresses a decomposition into a N-dimensions cartesian space. Yet, this space cannot usually be drawn as such. Next figure examplifies it in the case of the *Person* p-graph. As a matter of fact, three dimensions (N+1) are used instead of two to enable a simpler representation of selections. (Refer to [Borron, 1996a] for a different representation of clouds of basic points.) Note the double circles correspond to initial pigments.

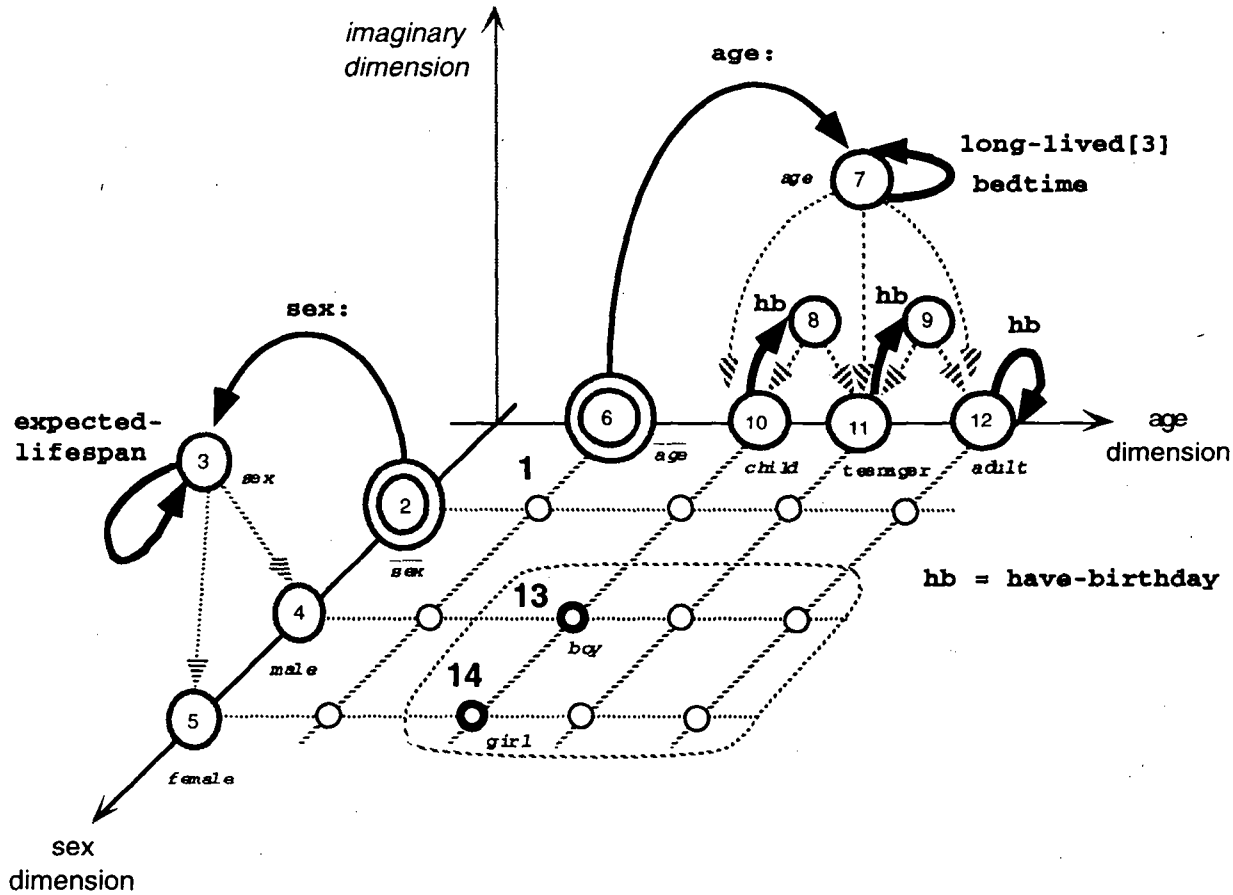


Figure C.2.

This *Person* example clearly illustrates what are "nearly independent" dimensions (here, *sex* and *age*). Were the *expected-lifespan* transition, as well as the *boy* and *girl* blends, not existing that the *sex* and *age* dimensions would be totally independent.

As a matter of fact, the independent part of each dimension can be extracted from *Person* and stored in an independent class (hence, classes *Sex* and *Age*). This being done, the *Person* p-graph is obtained by the **composition** of the two extracted classes (now termed **superclasses**) and by the addition of an increment. Next figure illustrates this in a schematic way. The figure afterwards expresses the same idea in the imaginary space of the above figure C.2 : the *Sex* and *Age* c-graphs respectively appear on the *sex* and *age* axis. Reference [Borron, 1996d] describes class inheritance as a generalization of local inheritance.

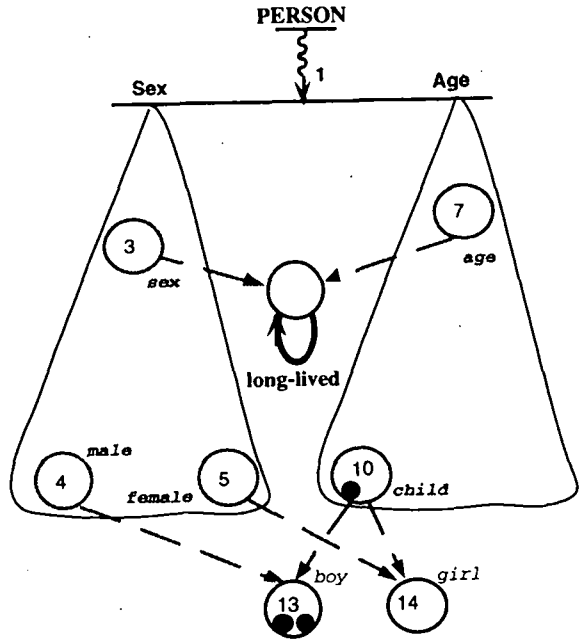


Figure C.3.

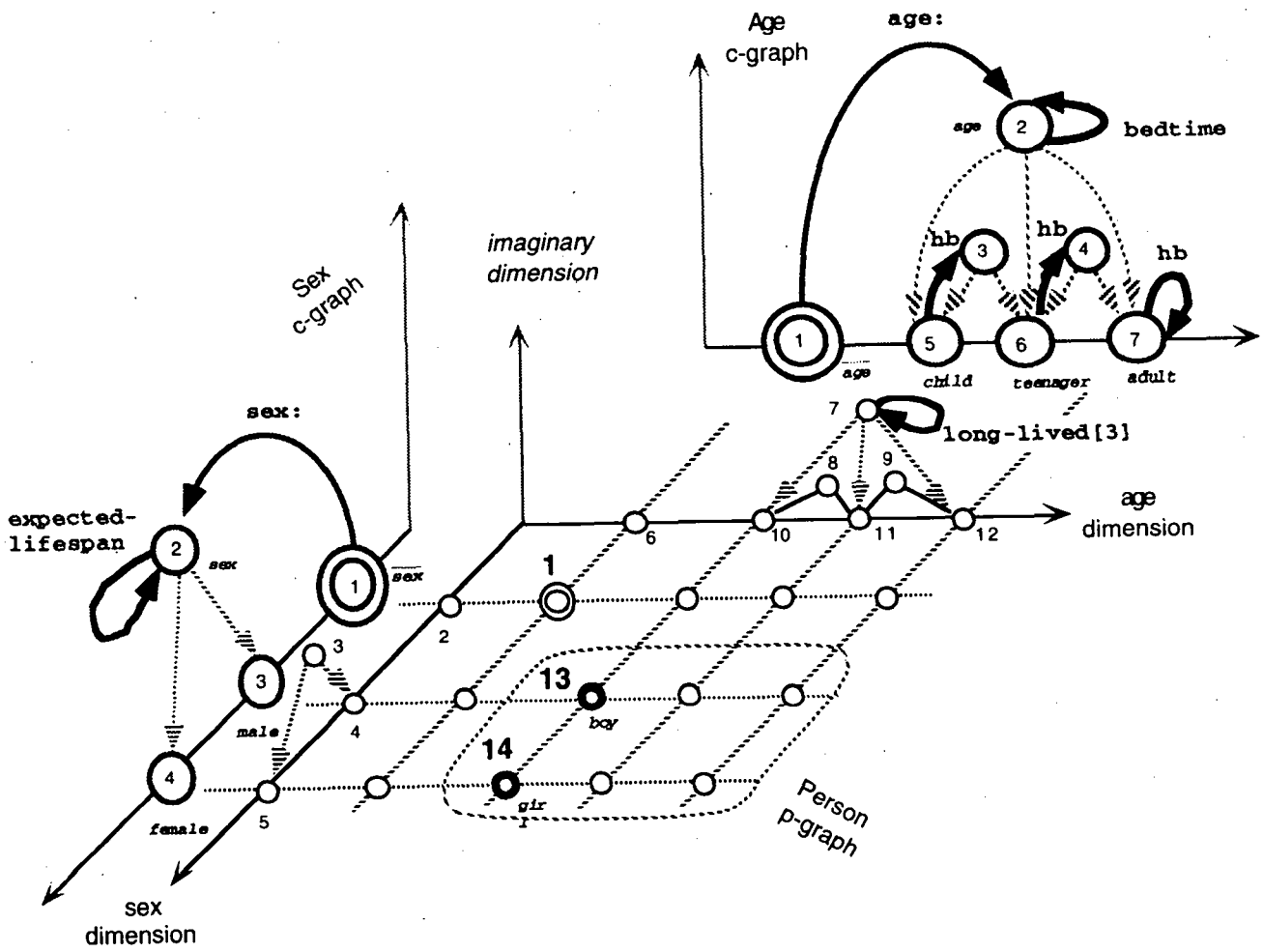


Figure C.4.

C.2 Second Example

Let's now consider the (less easy) case of a *Bounded-Stack* instance. Obviously enough, a *Bounded-Stack* instance behaves almost like a *Stack*.. It should thus be possible to get a description of the former by changing a bit the behaviour of the latter.

As shown previously (Appendix B), the color graph of a *Stack* exhibits three colors. Its behaviour is altered only in one color :

— when a *push:* message occurs in color 2 (*not empty*), a test should occur that possibly moves the instance state to *full*. In *full*, the *push:* transition is marked as **unwilling**. (This means the usual effect of a *push:* message is cancelled (redefined) ; however, the transition still exists to respect the usual constraints on transitions. For more details, see [Borron, 1996b] ;

— the *Stack* instance behaviour is unchanged in color 1 and 3.

Because the **composition** operator is fairly inappropriate in this case, a specialized operator is used. Termed the **derivation**, it acts as a composition only in one state (here, color 2) of the base color graph (here, *Stack*). It adds dimensions —here a single one : *bounded*— only in this point.

Next figure shows *Bounded-Stack* as a derivation of *Stack* using the *Bounded* mixin. A **mixin** capture an independent supplementary behaviour. (The design of mixins is elaborated in [Borron, 1996c].) Being reusable, a mixin adopts fairly general names, hence some necessary renaming : here, the *Bounded* mixin[23] uses *put:* and *get* in place of *push:* and *pop* . (A few checkings are also done, but this involves too much details for such an article.) The figure afterwards shows the equivalent c-graph obtained after expansion (one can check it is efficient).

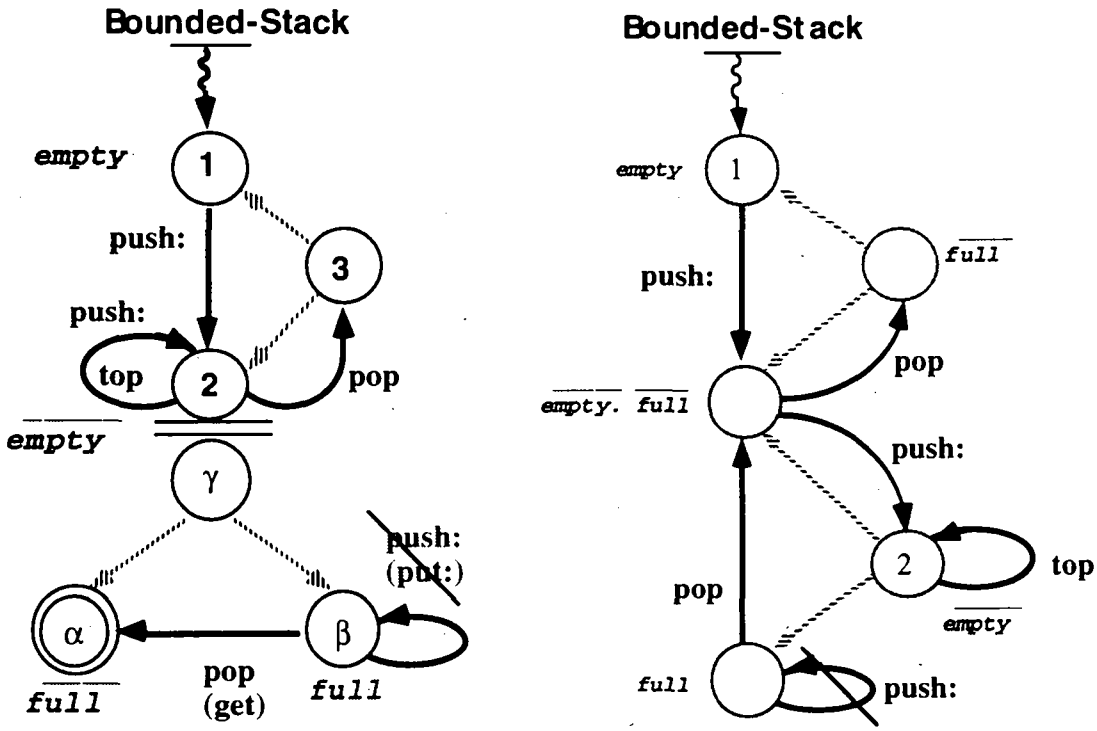


Figure C.3.

Next figure explicits the behaviour of a *Bounded-Stack* instance using the *stack* and *bounded* dimensions. (The *Bounded* mixin is exactly the same than in the above figure, yet without the conventional rules for simplifying its presentation.) Let's consider the*Stack* mini-token. This one moves along the horizontal axis. When present in color 2, another mini-token is created : this one is the *Bounded* mini-token ; it moves along the vertical axis erected in color 2. This mini-token is created in γ. γ is the source of a selection on α (*not full*) and β (*full*). The *Bounded* mini-token automatically moves to *not full* because *not full* is declared as being the initial node of the selection. When the *Bounded* mini-token is in *not full*. a **starred** *pop* message makes the test *not empty* to be checked. If the condition is not verified. the *Bounded* mini-token disappears ; given the result of the test, the *Stack* mini-token moves to *empty*. If the condition is verified. the *Bounded* mini-token stays in *not full*. A *push:* message makes it move to the selection root.

---

[23] This mixin supposes the *bound* (maximum number of elements in the *Stack* instance) is strictly greater than 1. To remove this restriction. a parameterized mixin is used (see [Borron, 1996c]).

From there it goes automatically either to *full* or to *not full*. In *full*, besides the special behaviour in case of a *push:* message, a *pop* message will make the *Bounded* mini-token to move back to *not full*..

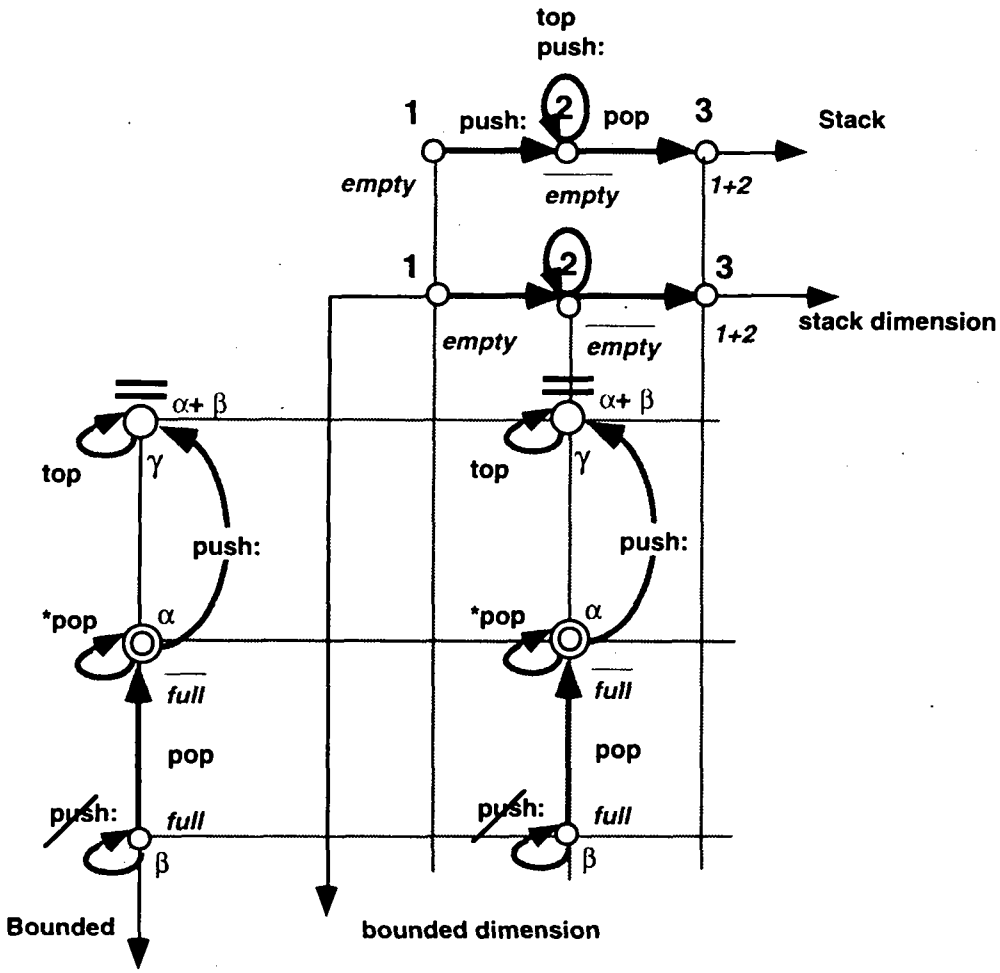


Figure C.4.

The constraints concerning the *Stack* and *Bounded* mini-tokens express that the *stack* and *bounded* dimensions are "nearly independent". The derivation construct manages these constraints while making the assembly of the base and mixin color graphs simpler than a constrained p-graph and more modular than a c-graph. The example may appear kind of difficult, but it should be retained that the existence of the two companion concepts derivation+mixin brings a modular solution to a probably non trivial problem (we are not aware of any other solution).

APPENDIX D :
HIGHER MODULARITY VIA MULTIPLE DIMENSIONS
2) IMPLEMENTATION LEVEL

D.1 Memory Representations

As already expressed in Appendix B, section 2, a memory representation can be specified for each color (in a c-graph) or pigment (in a p-graph). Thus memory representations may be made specific in a much more easy way than in traditional OOP. The example given in the same section shows a specific memory representation could possibly be chosen for a *Stack* instance when *empty* (since no elements need to be held, no slot is necessary) and an other one when *not empty*. These specific memory representations may be changed independently. Change of representations are done automatically when necessary. This malleability can be taken adavantage of to progressively pass from simple data-structures to more complex ones yet still keeping the same functionalities : this can be used, for example, when bootstraping the COP underlying system itself, or for making an application smoothly evolve from a poorly efficient implementation (ex. : prototype) to an efficient one.

Class inheritance for memory representations is quite simple. Given an instance of class *C0* in a certain state, it is done on a per dimension basis. For each dimension, a class-precedence-list[24] (cpl, for short) exists : it is obtained by linearizing the list of classes belonging to the hierarchy of *C0* and impacting the dimension in question (classes in this list are ordered from most specific to least specific). For example, considering an instance of *Person*, the list of classes impacting the *age* dimension are *Person* and *Age* (in this order). The memory representation for the considered instance of C0 is btained by an appropriate combination of the specifications of memory representations collected along the cpl of each involved dimension. Abstract dimensions like *object* of class *Object* or *bounded* of mixin *Bounded* are not involved. The combination may be particular to a language. It may also be changed via a meta-object protocol.

Next figure illustrates this point. Dimensions *dx0* and *dx1* are both involved. The memory representatiion will combine the memory representations obtained for dx0 and for dx1. The memory representation for *dx0* may itself combine the two specifications that are found (as done in CLOS).

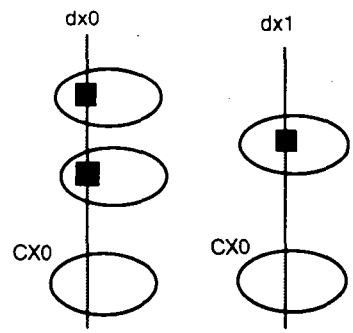


Figure D.1.

This proposal simply extends what is done in CLOS. In this language, the specifications of memory representations are are listed according to a single cpl, the class-precedence-list of *C0* obtained by linearizing the hierarchy of classes rooted in *C0*.

D.2 Methods

In this part, we do not consider the existence of qualifiers like **before:**, **after:**, ... (the next appendix takes care of them). Once again, the existence of dimensions enables a high level of modularity. For pedagogical purpose, we progressively address each level of complexity (single dispatch, multiple dispatch) and make a comparison with what is done or could be done in OOP (CLOS, an OOP language which already attains a high degree of modularity, is again our reference).

---

[24] Here, we used the CLOS vocabulary. Note a cpl exists for each dimension (in CLOS, only one exists per class).

D.2.1 Single Dispatch

a) in OOP

Let's suppose a message *m* is issued. Be *C0* the class of the receiver. Methods are listed according to the cpl of *C0*. (The way they are combined is expressed in the next appendix.)

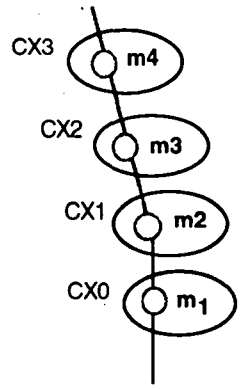


Figure D.2.

b) in COP

When a message *m* is sent to an object of class *C0*, valid methods also depend on the receiver's state. The search is made according to the cpl of each involved dimension. Here, a dimension is involved when the *m* transition is defined along this dimension. (For example, for a *Person* instance, the *age* and *sex* dimensions are both involved when searching methods named *long-lived*.) The result is an **invocation sequence diagram** listing methods along all involved dimensions and possibly along other dimensions that are employed (by the *m* methods) but not involved (by the *m* transitions). (For example, the *print* transition is defined for the sole *object* dimension, but the *print* methods generally use all the dimensions.)

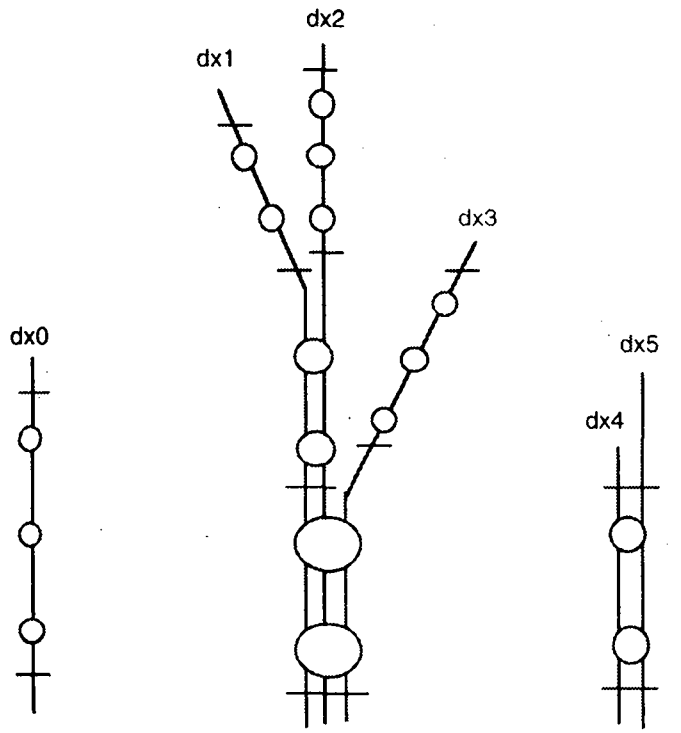


Figure D.3.

If a certain condition holds[25], this invocation sequence diagram is a list of trees. Each tree is due to one or several dimensions that are initially parallel (they traverse the same classes), then progressively diverge. (See figure.)

---

25 This condition is termed the "regularity ". A hierarchy is regular vs. its methods named *m*, if all *m* methods of degree≥ *K* that exist along a same dimension do satisfy the same *K* dimensions [Borron, 1996d]. In case of multiple dispatch, the dimensions of all arguments are to take into account. Regularity vs. methods, although not automatic, is obtained in practice.

As a matter of fact, compared to the original ordering of methods when recursively found along each dimension, this tree is possibly the result of a simple re-ordering[26] : this one privileges more specialized methods vs. less specialized ones ("prevalence of combined items" rule). Next figure illustrates that : method *m2* is initially ordered after *m'1* and *m"1* ; because m2 is more specialized than the other two (it is based on two dimensions instead of one), it is finally ordered before them. This enables the expression of modularity by successive refinements at different granularity levels: method *m1* refines method *m2* ; method *m'1* refines method *m1* ;... ; method *m"1* refines method *m"2* ; ... This basic scheme may be refined by masking in a rather sophisticated way : for example, *m1* may me declared as masking methods valid for both dimensions *dx1* and *dx2* (i.e. *m2*) while not masking methods valid for a single dimension.

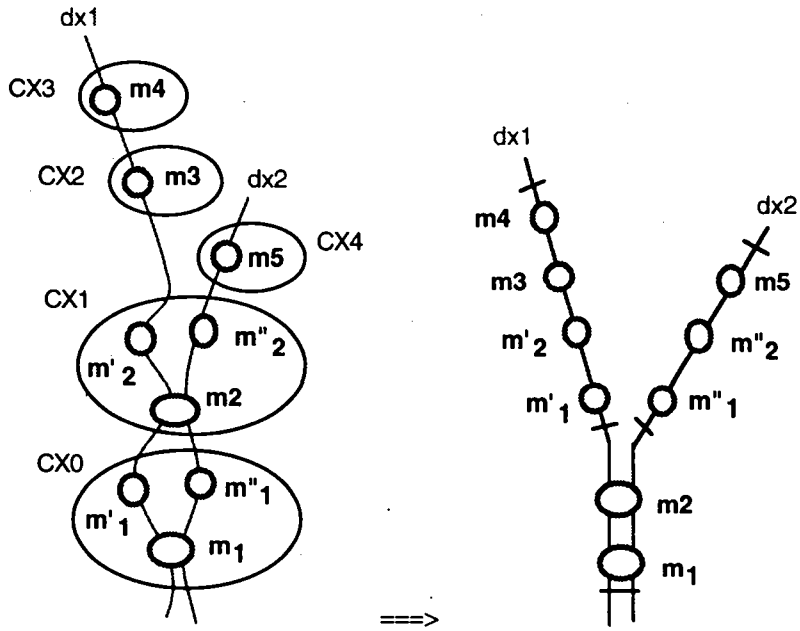


Figure D.4.

D.2.2 Multiple Dispatch

a) in OOP

Let's suppose a generic function call *m* is issued with a number of required arguments x, y,... of class C_{X0}, C_{Y0}, ... The *m* methods that satisfy the required arguments are primarily ordered according to the cpl of the first required argument class ; when several methods are attached to the same class in this first cpl, the cpl of the second required argument class is used to sort these methods ; if still tied, the cpl of the third required argument class is used in turn ; etc. For example, in the next figure, methods *m1*, *m2*, *m3*, *m4* and *m5* are ordered according to the cpl of *CX0* ; moreover, the first two methods are ordered according to the cpl of *CY0*. (The way these mehods are combined is expressed in the next appendix.)

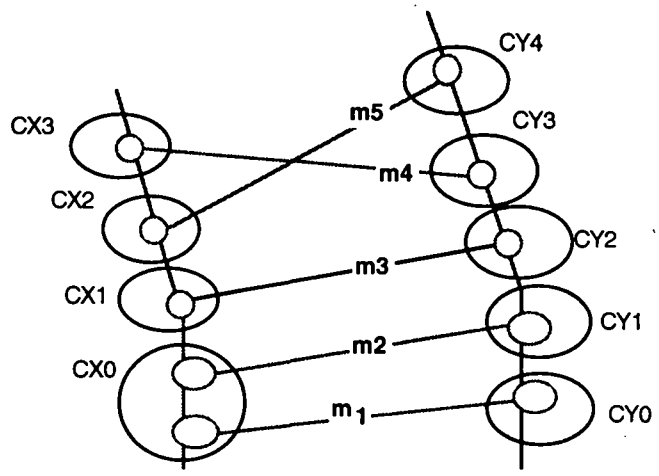


Figure D.5.

b) in COP

[26] Note that when two dimensions do not agree on the ordering of two methods, the ordering along the first dimension is selected. (An analogous rule exists in CLOS when, in the case of multiple dipatch, two cpl do not agree on the ordering of two methods.)

When considering a single required argument, the structure of the invocation sequence diagram corresponds to what was described in case of single dispatch (see D.2.1.b).

Due to global regularity, a number of topological properties can be stated [Borron, 1996d]. Next figure illustrates them. First of all, to each group of the *dxi* dimensions (first argument) corresponds a group of the *dyi* dimensions (second argument). For example, the tree made by the *dx1* and *dx2* dimensions is associated to the tree made by the *dy1*, *dy2*, and *dy3* dimensions. Even if these two trees do not contain the same number of dimensions, they are isomorphic : they both have three parts (a trunk and two branches). The number of nodes on each part (trunk included) is the same in the two trees (two on each trunk, two on each left branch, one on each right branch). There is one crossing due to methods *m4* and *m5* : the classes of their arguments are not found in the same order in the *CX0* and *CY0* hierarchies. No crossing may exist between parts which are not in correspondence.

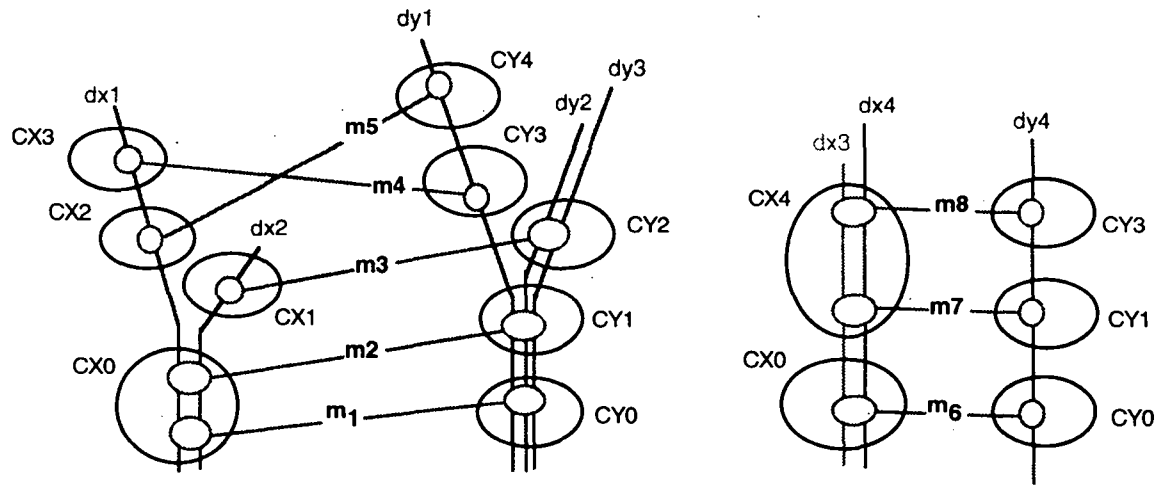


Figure D.6.

Given these topological properties, the OOP approach is generalized : methods are primarily sorted according to the ordering defined by the first required argument, the ordering defined by the other required arguments being used only for breaking ties if any. An example is given in the above figure with methods *m1* and *m2*.

APPENDIX E :
A PURELY DECLARATIVE LANGUAGE

E.1 Problem

A question which is somewhat perturbing in traditional OOP is that it mingles declarative and imperative programming techniques for method combination.

The declarative technique consists in attaching qualifiers (like **:before,** or **:after**) to method declarations and using these qualifiers as a basis for method combination : present in CLOS, this technique is (unfortunately) not quite frequently supported by OO languages.

To be general, the imperative technique is due to the OO construct used for calling, from inside a given method body, a method having a same name but normally masked (for example, because it is defined in a superclass of the class in which this method body exists). Such a construct (ex. : **super** in Smalltalk ; **call-next-method** in CLOS) allows reuse while preventing infinite looping, but it is non modular. Sonya Keene, a member of the CLOS committee, expresses her worry about it in clear terms[27]. While the declarative technique lets the user *"predict the order of methods without looking at the code in the bodies of the methods"*, the imperative technique is *"in a sense (...) a violation of modularity"* and should thus be used *"only when that power is truly necessary"*. Unfortunately, *"some programs"* cannot be written *"without resorting to the imperative technique"*.

As shown below, our proposal takes adavantage of the existence of pre- and post-methods ; it consists in running the pre-methods in bottom-up order, post-methods in top-down order and (if necessary) in attaching to the header of micro-methods (or, possibly, transitions) the indication 'masking should occur' in the form of a keyword (**:masking**), exactly like the **:before** or **:after** keywords. No **call-next-method** (nor **super**) is any longer necessary. By default, masking is done vs. all methods ranked after the masking method along the same dimension(s) ; but, it may also be specified as being done only vs. the methods valid for exactly the whole set of same dimensions.

E.2 Sophisticated Combinations without the Send-Super construct

Our proposal can be tuned so as to mimick –without the harmful **send-super** construct– the mechanisms supported by traditional OOP, for example the sophisticated CLOS ones.

In COP, instead of a simple list of classes for each argument, we have a list of trees (cf. the preceding appendix), each tree being due to one or several dimensions that are initially parallel (they traverse the same classes), then progressively diverge. So, to ease the initial comparison, we first express the basic idea for declarativeness supposing a single dimension per class hierarchy (i.e. per required argument) . This restriction is removed in subsection E.3.

E.2.1 Basic idea

Let's mimick the different cases of method combinations in CLOS

a) The standard method combination

This frequent combination deals with **:before**, **:after**, primary and **:around** methods. Original methods of a CLOS program will generally be splitted up in two parts according to the class colors and pigments. Certain **call-next-method** invocations may vanish in the process.

The solution we propose applies to the bodies of the resulting methods :
- **:before** methods are represented as pre-methods, **:after** methods as post-methods ;
- primary and **:around** methods are represented as pairs of pre- and post-methods.

---

27 [Keene, 1989], section 5.8, p.111 : *"Guidelines on controlling the generic dispatch"*.

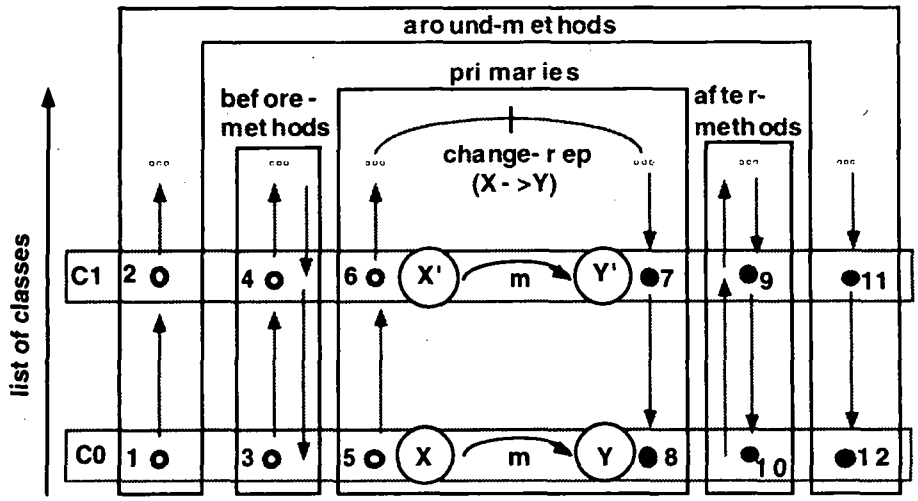


Figure E.1.

The above figure illustrates our proposal (the numbers give the order of method executions).

Note the relative ordering of methods in CLOS (ex. : the running of **:before** methods from most specific to least specific or the running of **:after** methods from least specific to most specific) appears as a mere consequence of our more general mechanism (pre-methods in ascending order, post-methods in descending order).

See [Borron, 1995b] for the specification of details (extra data control to respect the intricacies of the **call-next-method** semantics as well as the consequence of method splitting about temporaries.)

b) Other method combinations

Besides the standard method combination, CLOS offers a number of other built-in method combination types (simple ones, like **progn** or **max**). In these, the **:around** and primary roles are recognized. The former role may use **call-next-method** (and **next-method-p**) exactly as in the standard method combination : the above analysis[28] thus applies. The latter role can't use **call-next-method** (hence, a simpler treatment than with the standard method combination) ; order of primaries is defaulted to **:most-specific-first** but can be changed to **:most-specific-last** : obviously, such an effect can easily be obtained in our approach by specifying the primary methods as being pre- or post-methods.

A user is also given two possibilities to define other method combination types. The first one, termed "short form", leads to the considerations just evoked (the simple built-in method combination types act as if they were defined using this form). The second one, termed "long form", provides great flexibility. From the perspective of the current discussion, this form leads to the considerations already evoked, either about the standard method combination type (which can be defined using it) or about the order of methods.

E.2.2 Generalization

Two method combinations are considered. The first one, termed **[combination]**, combine methods of a same degree in a same group of dimensions (like *m1* and *m2* in figure D.4). It is like the CLOS combination along a cpl. The second combination, termed **{combination}**, combines items found along two or more diverging branches. As a matter of fact, the items to be [combined] (resp. {combined}) may themselves result from a {combination} (resp. [combination]).

Using this notation, the result of figure D.4 (right part) is the following : *[m1 m2 [[m'1 m'2 m3 m4] [m"1 m"2 m5]]]*. The result of figure D.6 is : *{[m1 m2 [[m5 m4] m3]] [m6 m7 m8] }*.

The interpretation of the [combination] is different in case of pre-methods (ascending order is the default : methods are processed from left to right between the brackets) and post-methods (descending order is the default : methods are processed from right to left between the brackets).

---

28 Primary or **:around** methods are typically made of a first part making a number of computations, followed by a **call-next-method** the result(s) of which is (are) used in a second part. The pre-method corresponds to the first part ; the post-method, to the second. More complex schemes may appear in traditional OOP (presence of several instances of **call-next-method** instead of a single one, possibly under the control of test or loop statements). In the COP view, these extra controls are obtained using colors at an appropriate level.

When qualifiers are used, the independence of dimensions allows various —and a priori equivalent— formulas to be used for combining the whole set of methods. The default is to combine in order the combined methods resulting from: the combination of all around pre-methods, the combination of all before methods, the combination of all primary pre-methods, the combination of all primary post-methods, the combination of all after methods, the combination of all around post-methods. This generalizes the CLOS standard combination. For other method combinations, the remark made in E.2.1.b applies. (For an in depth study, see [Borron, 1996d], subsection 10.3.)

APPENDIX F :
MAKING LINEARIZATION EASY TO PREDICT

F.1 Problem

Inheritance is an interesting feature. Linearization is a simple and systematic way to handle it, at least as a default mechanism. Yet, it can be surprising by its results. A keyword in this matter is thus prediction.

This aspect has been the motivation for **monotonicity**, a property identified in [Ducournau et alii, 1992]. This property is obtained when, for any hierarchy *HC0*, the cpl in a superclass *Si* of *C0* is a sublist of the cpl in *C0*, i.e. is the list of classes obtained by removing from *LC0* all classes that do not belong to the hierarchy *HSi* rooted in *Si*. When a linearization algorithm is monotonic, the cpl (global behaviour) in class *C0* can be predicted from the cpl (global behaviour) in the sole superclasses *S1*... *Sn*, plus *C0* (increment behaviour). This property is supported by the linearization algorithms respectively proposed in [Ducournau et alii, 1994] and in [Borron, 1996x].

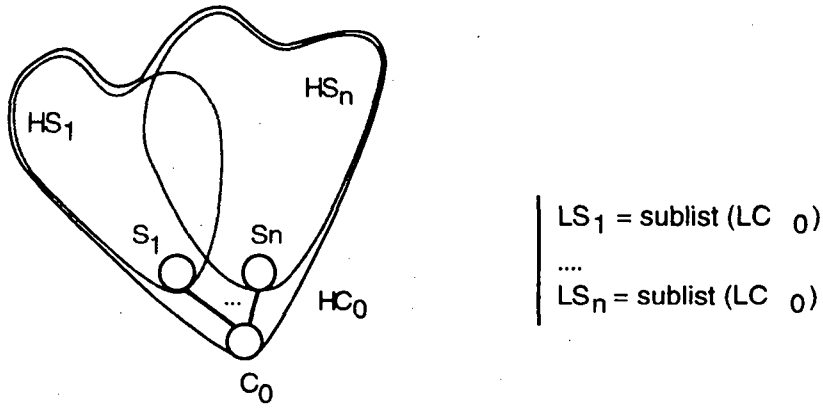


Figure F.1.

An additional property is desirable for COP : **congruency** (cf. [Borron, 1996d]. It is obtained when the cpl of the sub-hierarchy *HC0(C)* (hierarchy rooted in *C0* and with summit *C*) is a sublist of the hierarchy rooted in *C0*, whatever *HC0* and the summit *C*. Besides enabling a fast computation of the cpl of each dimension, this property also expresses a form of regularity : the ordering for the dimension(s) introduced in class *C* depends solely on the classes impacting these dimension(s)

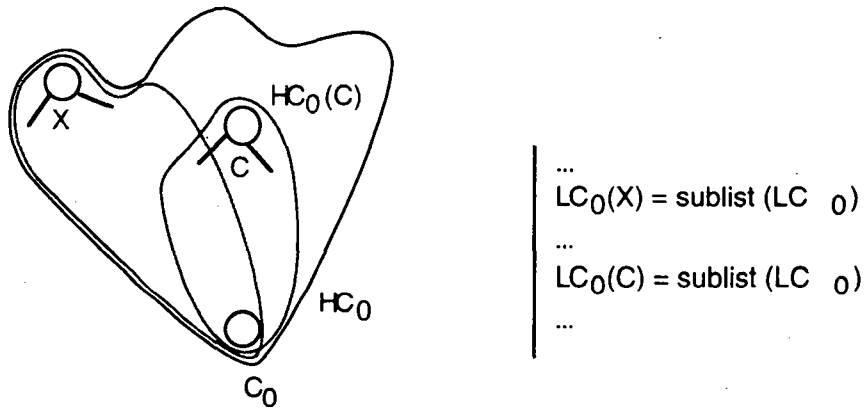


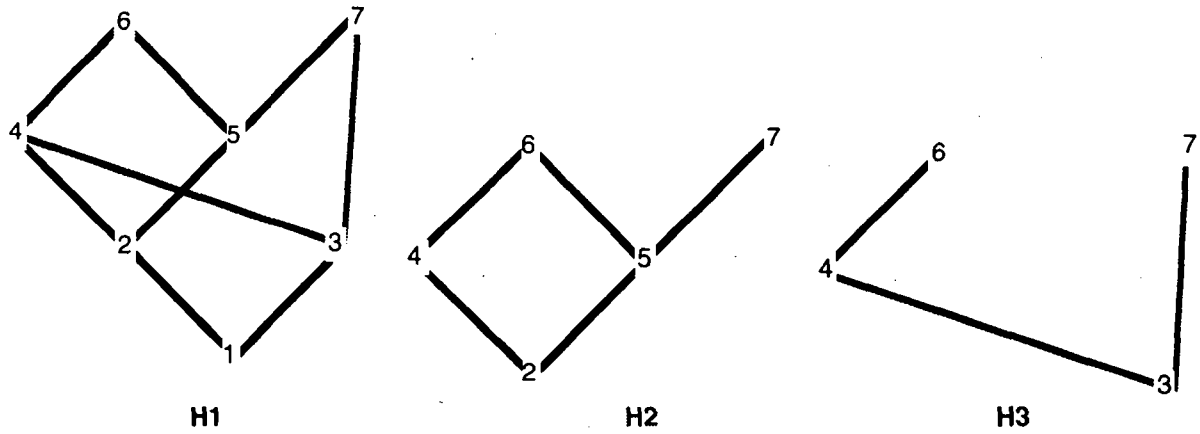
Figure F.2.

However, as demonstrated in [Borron, 1996d], congruency and monotonicity are antagonistic properties : a linearization algorithm cannot be congruent and monotonic for all hierarchies if it lists nodes blindly (i.e. if it lists nodes along a branch in the same way be this branch leading to a converging node or not). Thus, three solutions were envisaged. The last one was retained as being the most helpful for the programmer.

F.2 Solution 1

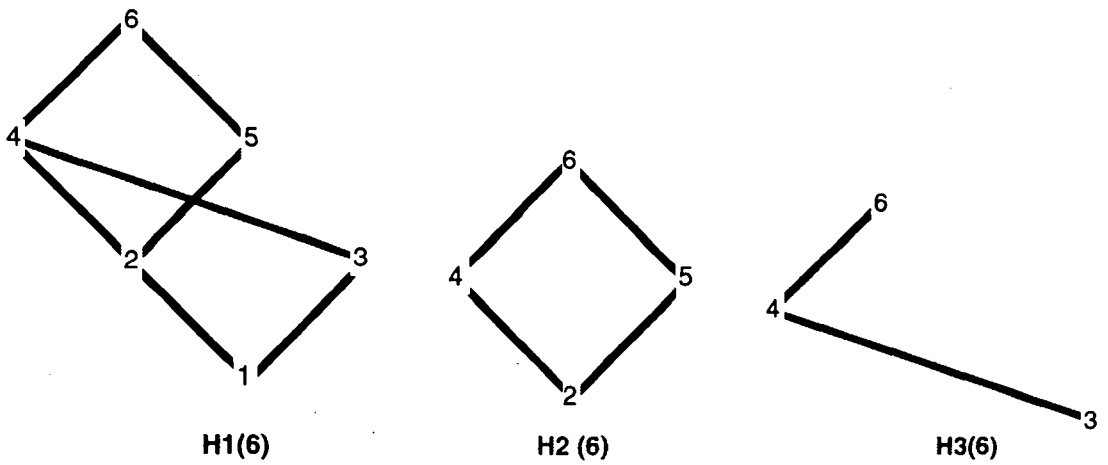
It consists in using a congruent linearization algorithm, for example and notably the LOOPS one (this one is demonstrated to be congruent in [Borron, 1996d]). Under this hypothesis, the cpl for any dimension existing in *C0* is easily obtained from *LC0*. However, in this case, monotonicity is not guaranteed : thus, the cpl in *C0* cannot be predicted from the cpls of the superclasses of *C0* ; and the cpl for a given dimension of *C0* cannot be predicted from the cpls of the superclasses of *C0* for the same dimension.

Next figure illustrates this. Here, the cpl in node 1 is *L1 = (1 2 5 3 4 6 7)*. It cannot be predicted from the cpls in node 2, i.e. *L2 = (2 4 5 6 7)*, and in node3, i.e. *L3 = (3 4 6 7)*. (*L2* and *L3* are poorer in informations than *H2* and *H3* ; the lost information appears essential for determining *L1*.)



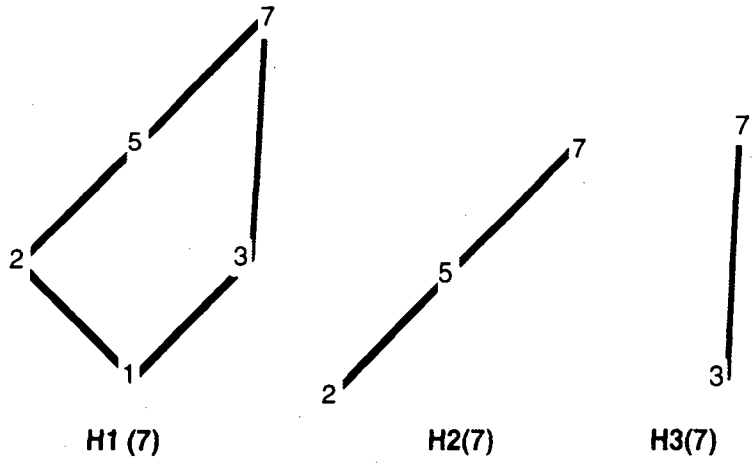
Figures F.3, F.4 & F.5.

Concerning the dimensions that are introduced in class 6, the ordering of interest concerns the sub-hierarchy *H1(6)*. This ordering can be computed directly : *L1 (6) = (1 2 5 3 4 6)*. Since congruency is known to exist, *L1 (6)* can be obtained from *L1* by restricting this one to the sole classes that exist in *H1(6)*, i.e. by removing class 7 : *(1 2 5 3 4 6 7)*. However, because monotonicity is not guaranteed, the user cannot predict the ordering of a sub-hierarchy like *H1(6)* from the ordering of *H2(6)* and *H3(6)*, the sub-hierarchies with summit 6 and rooted in the superclasses of class 1. Here, *L2(6) = (2 4 5 6)* and *L3(6) = (3 4 6)*. (*L2(6)* is not a sublist of *L1(6)* : class 5 is not ordered in the same way in both lists.)



Figures F.6, F.7 & F.8.

A similar study can be made vs. the sub-hierarchy *H1(7)*. In this case, results are not provoking any surprise : monotonicity happens to be observed.



Figures F.9, F.10 & F.11.

Next figure graphically illustrates the study made so far vs. (the dimension(s) introduced by) class 6, as well as the similar study that would be made vs. class 7.

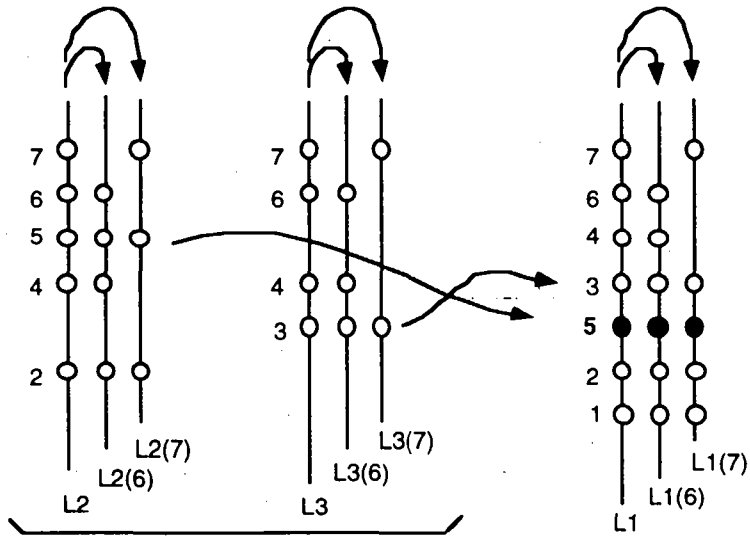


Figure F.12.

With a congruent algorithm, the problem is thus that the user should consider the whole hierarchy each time : the behaviour of the ancestor classes of *C0* cannot be abstracted in the sole superclasses of *C0*. The ordering in *C0* may be a surprise vs. the orderings previously obtained (in the superclasses of *C0*). On the opposite, the good point is that the ordering for each dimension always parallels the ordering obtained for the whole hierarchy (no surprise).

F.3 Solution 2

F.3.1 Properties

Given a monotonic linearization algorithm (cf. [Ducournau et alii, 1994] or [Borron, 1996x]), this solution consists in computing *LC0* and, for each class *C* introducing a dimension, in restricting *LC0* to the classes belonging to *HC0(C)*. Be *$\mathcal{L}$C0(C)* this ordering.

Because the linearization algorithm is monotonic, *LC0* may be obtained incrementaly (from the cpl of each superclass of *C0*). The cpl for each dimension is easily computed. Yet, because congruency cannot be guaranteed in all cases, this *$\mathcal{L}$C0(C)* ordering may well be different from *LC0(C)*, i.e. from the one obtained by directly linearizing the hierarchy *HC0(C)*. In other words, the $\mathcal{L}$ ordering cannot be predicted by isolating the classes impacting the dimension in question.

Yet, this $\mathcal{L}$ ordering happens to be monotonic. Since the linearization algorithm is monotonic, *LSi* is a sublist of *LC0* (if *LC0* exists). Since *$\mathcal{L}$Si(C)* is obtained from *LSi* by removing a number of classes, it is a sublist of *LSi* ; in the same

way, *LC0(C)* is a sublist of *LC0*. Thus, *ℒSi(C)* is a sublist of *LC0(C)* (they order the same classes in the same way ; and all classes in *ℒSi(C)* are part of *LC0(C)*). This property is valid for any superclass *Si* of *C0*. If the orderings differ for *LSi (C)* and *LSj(C)* (with $i \neq j$), then *LC0* does not exists (the linearization algorithm rejects *HC0*). We call this property **ℒ monotonicity**.

F.3.2 Example

Let's consider the same example as above. Here, *L1* is equal to *(1 2 3 4 5 6 7)*. For the dimension(s) introduced by class 6, we use the restriction of *L1* to the classes of *H1(6)*. Be it *ℒ*1(6) = *(1 2 3 4 5 6 7) = (1 2 3 4 5 6)*. Similarly, for the dimension(s) introduced by class 7, we use *ℒ*1(7) = *(1 2 3 4 5 6 7) = (1 2 3 5 7)*.

While the ordering *ℒ1(6)* used for the sub-hierarchy *H1(6)* happens to be equal to *L1(6)*, the ordering *ℒ1(7)* used for the sub-hierarchy *H1(7)* is not equal to *L1(7)*, i.e. to *(1 2 5 3 7)*. This is an unexpected result for a programmer attempting to compute the ordering for dimensions originating from class 7 by isolating the sub-hierarchy *H1(7)* : the ordering selected for such dimensions does not depend solely on the classes impacting these dimension(s). The reason for this is known : because the linearization algorithm is blind and monotonic, it is not systematically congruent.

Yet, this solution presents a good point : the *ℒ* ordering vs. each dimension is monotonic.
— First, *L2 = (2 4 5 6 7)*. Thus, *ℒ2(6) = (2 4 5 6 7) = (2 4 5 6)*. Similarly, *ℒ2(7) = (2 4 5 6 7) = (2 5 7)*.[29] *ℒ2(6)* is a sublist of *ℒ1(6)*, and *ℒ2(7)* is a sublist of *ℒ1(7)*.
— Second, *L3 = (3 4 6 7)*. From which we compute *ℒ3(6) = (3 4 6)* ; and, similarly, *ℒ3(7) = (3 7)*.[30] *ℒ3(6)* is a sublist of *ℒ1(6)*, and *ℒ3(7)* is a sublist of *ℒ1(7)*.

Next figure graphically illustrates this study.

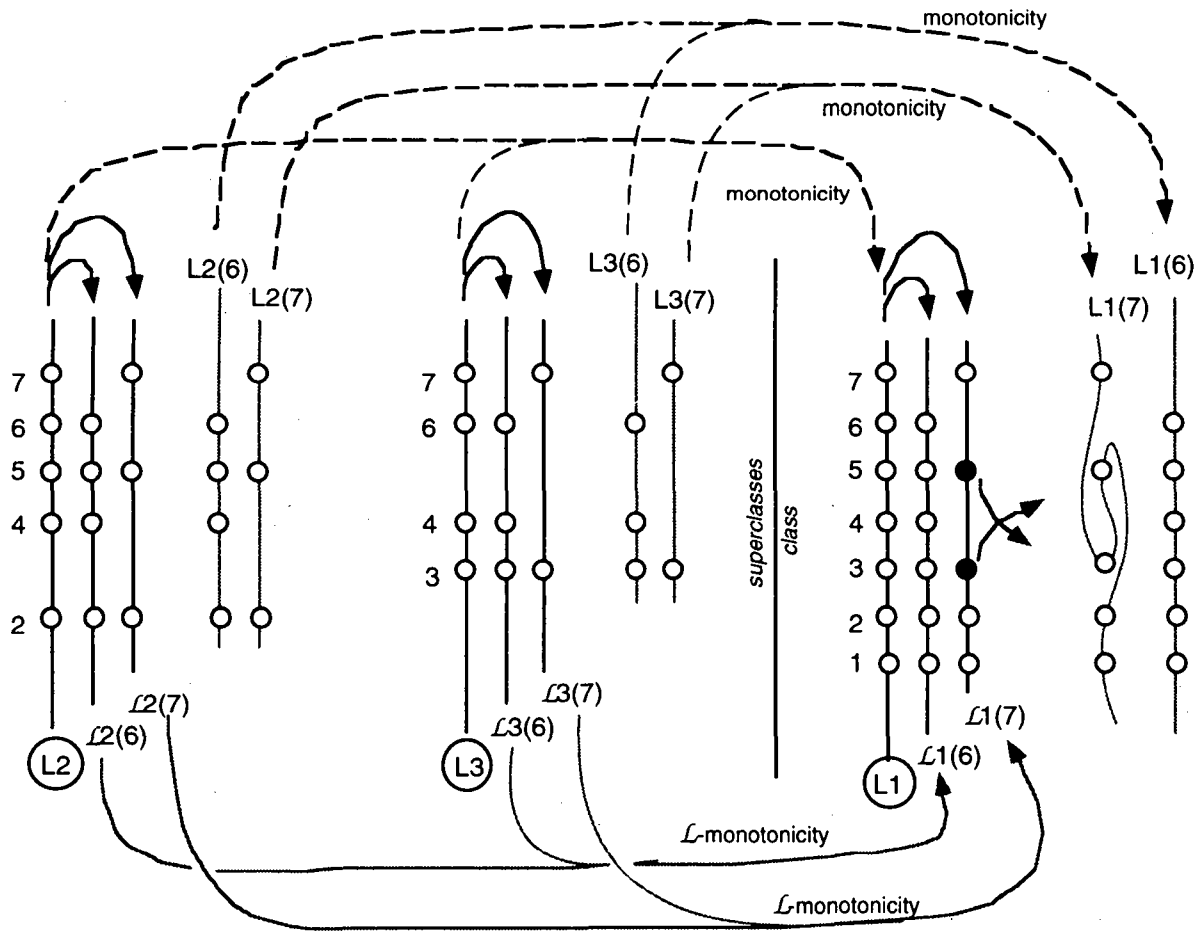


Figure F.13.

---

[29] Note congruency is verified in these cases : *ℒ2(6)* happens to be equal to *L2(6)* ; *ℒ2(7)* to *L2(7)*.

[30] Congruency is also verified in these cases : *ℒ3(6)* happens to be equal to *L3(6)* ; *ℒ3(7)* to *L3(7)*.

F.4 Solution 3

The previous solution being not totally satisfactory, a third one is now considered. It consists in a monotonic linearization dimension per dimension. In other words, for each class *C* that introduces one or more dimensions d_{Ci} in a hierarchy *HC0* rooted in *C0*, the ordering which is considered for these dimensions is *LC0(C)* where *L* is monotonic. The various cpls obtained this way for various *C* may not order two same classes in the same way. Yet, for each dimension d_{Ci}, the order is monotonic : the cpl in each superclass *Si* abstracts the hierarchy *HSi(C)* ; on that sole basis, the user may predict the order in *C0* .

In our example, this yields the following orderings :

— dimension(s) introduced in class 6 : *L2 (6) = (2 4 5 6). L3(6) = (3 4 6). L1(6) = (1 2 3 4 5 6). L1(6)* can be obtained directly from the linearization of *H1(6)* : the ordering depends only on the classes impacting the dimension(s) in question. It can also be obtained from the orderings (global behaviours) first abstracted in 2 and 3, i.e. from *L2(6)* and *L3(6)*) (see next figure) ;

— dimension(s) introduced in class 7 : *L2 (7) = (2 5 7). L3(7) = (3 7). L1(7) = (1 2 5 3 7). L1(7)* can be obtained directly from the linearization of *H1(7)*, or from the *L2(7)* and *L3(7)*.

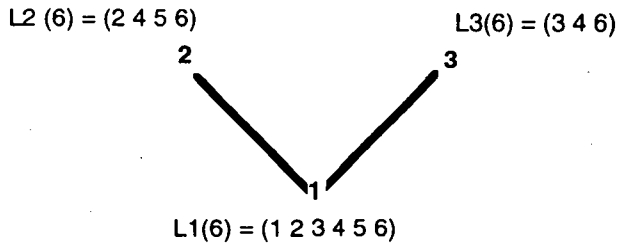


Figure F.14.

F.5 Conclusion

Because it is intrinsically less open to surprises than the two previous ones, the latter solution is a priori retained even if it is less efficient. Yet, it appears interesting to experiment the second solution for a cognitive evaluation. If satisfactory to users, this solution may be prefered to the third one because of its efficiency. (Our inheritance algorithms cope with both linearizations.)

In both cases, determining under what conditions *LC0(C) = LC0(C)* is useful : were the second solution being chosen, this defines when the user will not be surprised by *LC0(C)*, i.e. when the sub-hierarchy *HC0(C)* can safely be displayed to let the user think about it and possibly move methods around ; were the third solution being chosen, this draws up when *LC0(C)* can be computed efficiently using *LC0(C)* in place.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine - Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes - IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes - 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 (France)
Unité de recherche INRIA Rocquencourt - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249 - 6399